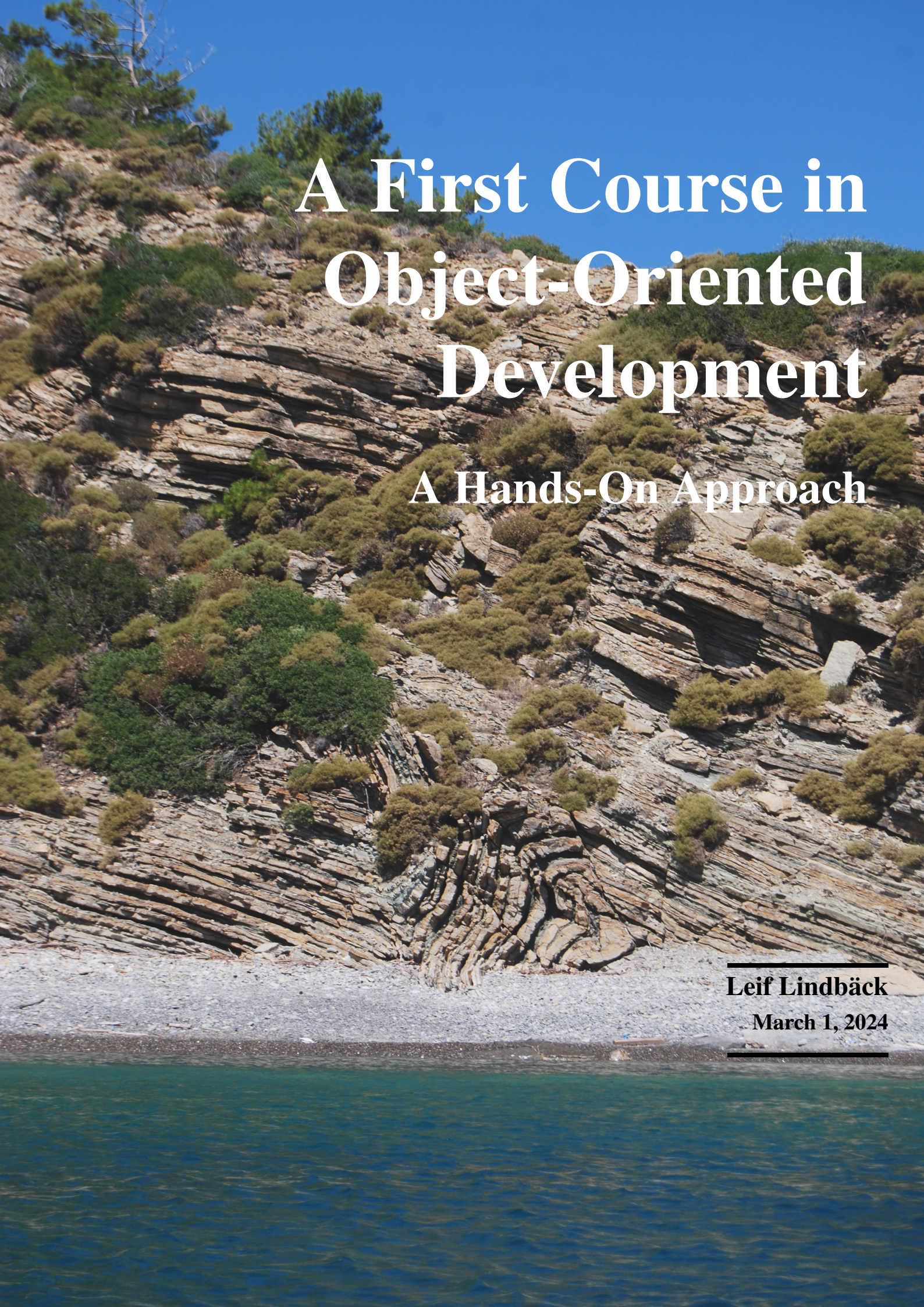


A First Course in Object-Oriented Development

A Hands-On Approach

Leif Lindbäck

March 1, 2024

Revision History

Date	Description	Page (at time of revision)
2016-02-25	First published version. Sections 6.6-6.10 and chapter 7 are not yet written.	All
2016-04-01	Fixed typos	Section 5.6
2016-04-05	Class `Amount` was kept, by mistake, after it had been stated that it should be removed.	29-31
2016-04-05	Small changes to make the text clearer.	69-71
2016-04-05	Improved table layout	26
2016-04-15	*Create* stereotype was missing in some classes in design class diagrams.	64, 68, 71, 75, 77
2016-04-15	Added section 6.6	103-118
2016-04-15	Added section 6.7	118-119
2016-04-15	Moved testing to a separate chapter	120
2016-04-15	Improved layout of table 6.1	81
2016-06-02	Clarified description of naïve domain model.	31
2016-06-07	Added chapter seven.	130-149
2016-06-07	Split chapter eight.	150-153
2016-10-25	Added exceptions and association name to UML cheat sheet.	159-163
2016-12-06	Added chapter eight.	150-174
2016-12-06	Changed code reference to GitHub repository.	280
2016-12-19	Changed license.	ii
2017-01-27	Added chapter nine (not yet completed)	176-205

Date	**Description**	**Page** (at time of revision)
2017-01-27	Removed the *Further Reading* sections	2, 14, 21, 40, 81, 150
2017-01-27	Removed references to specific courses from preface	iii
2017-01-31	Added singleton pattern to section 9.4	204-207
2017-02-23	Added section 9.3	183-192
2017-03-09	Added template method pattern	215-220
2017-03-10	Added warning not to model parts as attributes in domain model.	34-35
2017-03-14	Changed legacy time to java.util.time	91, 92, 117, 142, 143, 145
2017-03-20	Fixed typos	17, 21
2017-04-11	Added composite pattern	214-218
2017-04-27	Wrong caption to figure 7.16	143
2017-04-27	Wrong code in figure 7.17	144
2017-05-17	Attributes were, by mistake, not private in listing 7.17	144
2018-03-15	Proofread and clarified chapter four	20-39
2018-03-15	Proofread and clarified revision history, license and preface.	i-iv
2018-03-16	Proofread and clarified chapters one to three	2-18
2018-04-12	Proofread and clarified chapter five	40-80
2018-04-17	Proofread and clarified chapter six	81-120
2018-04-17	Added instructions for creating unit tests with IntelliJ.	134-135
2018-04-19	Proofread and clarified chapter seven.	121-151
2018-05-02	Proofread and clarified chapter eight.	152-177
2018-05-03	Added interface symbols to sequence and communication diagrams in UML cheat sheet.	234, 236
2018-05-03	Corrected erroneous javadoc comments in listings.	94, 173
2018-05-04	Proofread and clarified chapter nine.	178-227

Date	**Description**	**Page (at time of revision)**
2018-05-04	Added cover page.	
2019-03-07	Clarifications of domain model.	24, 30, 34
2019-03-21	Clarifications of the layer pattern.	57
2019-03-21	Clarifications of the DTO pattern.	59
2019-03-21	Clarifications of domain model usage at design.	60
2019-04-02	Clarified figures 4.11 and 4.12.	32,33
2019-04-09	Fixed typo.	57
2019-04-12	Changed exceptions in communication diagrams to use asynchronous arrow.	155, 238
2019-05-06	Clarified when to use inheritance	189
2020-02-27	Changed NetBeans homepage to Apache.	354
2020-03-10	Clarified asynchronous message in sequence diagram.	155
2020-03-10	Clarified unit testing exceptions.	176
2020-03-10	Changed from JUnit 4 to to JUnit 5.	124-129, 133-152
2020-04-09	Corrected typo	78, 80
2020-08-14	Added dedication	v
2021-02-24	Fixed caption layout bug	
2021-03-12	Headline reflects section content better	189
2021-03-12	Package names now the same as in code repository	182-187, 199
2021-03-12	Reflective loading uses *Constructor.newInstance()* instead of *Class.newInstance()*	188, 215
2021-03-16	Emphasized that data must have an origin	64
2021-03-16	Emphasized when to name objects	64, 82
2021-03-16	Business logic exceptions shall not be logged	174, 175
2021-03-16	Specified package for error handling	173, 174
2021-04-23	Added missing lines to listing C.105	296

Date	**Description**	**Page (at time of revision)**
2021-04-23	Wrong arguments in message 2 in figure 5.35	73
2021-04-28	Missing argument in message 2 in figure 5.27	65
2021-07-01	Added chapter 10	231-232
2022-02-22	Log runtime exceptions in all classes	174
2022-03-08	Added exercises for chapter seven	153-163 and 364-377
2022-04-13	Removed obsolete reading instructions for chapter 1	2
2022-04-13	Removed obsolete reference to course web	378
2023-02-17	Emphasized that both return value notations in sequence diagrams are equal	23
2023-02-20	Emphasized that it's fine to call the integration layer from controller	57, 58, 59
2023-02-20	Improved explanation of difference between DTO and entity	59
2023-02-21	Changed reference to Fowler's refactoring book to second edition	86, 241
2023-02-21	Added the code smell *complicated flow control*	107-109
2023-03-03	Clarifications of common design mistakes	81-82
2023-03-03	Added missing «create» stereotype to fig 5.38, 5.39, 5.42	75, 76, 79
2023-03-16	Improved descriptions of common design mistakes.	81-84, 349-358
2023-03-30	Corrected parameter name in figure 4.18 and 4.19, parameter 'receipt' was erroneously called 'Receipt'.	38, 39
2023-04-17	Corrected confusing text about dividing responsibility between view and controller.	56
2023-10-31	References to *business logic* were missing in the index.	392
2023-10-31	Added explanation of *business logic*.	53
2024-03-01	Added hyphen to title.	cover, v
2024-03-01	Added appendix E, *missing features*.	391-395

License

To my friend Jesus, who has helped me in many difficult situations in life.

Preface

This text is a compilation of material developed during seventeen years of teaching a first course on object oriented development. Students taking the course have previously taken one 7.5 hp credit course in object-oriented programming, using Java as example language. Thus, the reader is assumed to have basic knowledge of Java programming. Important concepts, in particular objects and references, are repeated in chapter 1.

Things that are crucial to remember, but easy to miss, are marked with an exclamation mark, like this paragraph. Forgetting the information in such a paragraph might lead to severe misunderstandings.

!

Paragraphs warning for typical mistakes are marked NO!, like this paragraph. Such paragraphs warn about mistakes students have frequently made.

NO!

There are Java implementations of all UML design diagrams in Appendix C. The purpose is to make clearer what the diagrams actually mean. The analysis diagrams are not implemented in code, since they do not represent programs. There is a NetBeans project with all Java code in this book, that can be downloaded from GitHub [Code].

Contents

Part I

Background

Chapter 1

Java Essentials

It's not possible to understand this book without previous knowledge of an object-oriented programming language, preferably Java since all code listings in the entire book are written in Java. To facilitate, this chapter repeats important concepts of Java, but it doesn't not cover the whole language. All these code examples are available in a NetBeans project, which can be downloaded from GitHub [Code].

1.1 Objects

An object-oriented program consists of classes, which group data and methods operating on that data. A class represents an abstraction, for example *person*. An object represents a specific instance of that abstraction, for example the person *you*. Whenever a new person shall be represented in the program, a new object of the `Person` class is created. This is done with the operator `new`, as illustrated on lines one and five in listing 1.1.

```
1  Person alice = new Person("Main Street 2");
2  System.out.println("Alice lives at " +
3                     alice.getHomeAddress());
4
5  Person bob = new Person("Main Street 1");
6  System.out.println("Bob lives at " + bob.getHomeAddress());
7
8  alice.move("Main Street 3");
9  System.out.println("Alice now lives at " +
10                    alice.getHomeAddress());
11 System.out.println("Bob still lives at " +
12                    bob.getHomeAddress());
```

Listing 1.1 Creating and calling objects.

Two different objects, representing the persons *Alice* and *Bob*, are created in listing 1.1. Note that when Alice moves to another address, line eight, Bobs address remains unchanged, since `alice` and `bob` are different objects. The output when running the program is provided in listing 1.2, and the source code for the `Person` class is in listing 1.3.

```
1 Alice lives at Main Street 2
2 Bob lives at Main Street 1
3 Alice now lives at Main Street 3
4 Bob still lives at Main Street 1
```

Listing 1.2 Output of program execution

```
1  public class Person {
2      private String homeAddress;
3
4      public Person() {
5          this(null);
6      }
7
8      public Person(String homeAddress) {
9          this.homeAddress = homeAddress;
10     }
11
12     public String getHomeAddress() {
13         return this.homeAddress;
14     }
15
16     public void move(String newAddress) {
17         this.homeAddress = newAddress;
18     }
19 }
```

Listing 1.3 The `Person` class

A *constructor* is used to provide initial values to an object. In listing 1.3, the value passed to the constructor is saved in the object's field on line nine. Sending parameters to a constructor is just like sending parameters to a method. More than one constructor is needed if it shall be possible to provide different sets of initialization parameters. The constructor on lines four to six is used if no home address is specified when the object is created, the constructor on lines eight to ten is used when a home address is specified. Note that, on line five, the first constructor calls the second constructor, using `null` as the value of the home address.

The variable `this` always refers to the current object. The variable `this.homeAddress` on line nine in listing 1.3 is the field declared on line two, `homeAddress` on line nine is the constructor parameter `homeAddress` declared on line eight. These two are different variables.

A word of warning: *use static fields and methods very restrictively!* Static fields are shared by all objects of the class. If for example the person's address was static, all persons would have the same address. Such a program would be useless. Methods shall also not be static, since static methods don't belong to any particular object, and can thus not access the fields of an object. Static fields and methods are appropriate only in a few, very special, cases.

1.2 References

The `new` operator creates an object and returns a reference to that object. A reference can, like any other value, be stored in variables, sent to methods, etc. This is illustrated in listing 1.4, with a program where a person feeds a dog. First, a `Bowl` object is created on line three. The reference to that object is passed to the constructor of a `Person` object on line four. On line 14, the `Person` object stores the reference in the `bowl` field, declared on line 10. Then, on line five, the `main` method calls the `feedDog` method in `person`. In `feedDog`, the method `addFood` is called in the previously created `bowl`, on line 18. This shows how an object (`bowl`) can be created in one place (`main`), passed to another object (`person`) and used there.

```
1  public class Startup {
2      public static void main(String[] args) {
3          Bowl bowl = new Bowl();
4          Person person = new Person(bowl);
5          person.feedDog();
6      }
7  }
8  public class Person {
9      private Bowl bowl;
10     private int gramsToAdd = 200;
11
12     public Person(Bowl bowl) {
13         this.bowl = bowl;
14     }
15
16     public void feedDog() {
17         bowl.addFood(gramsToAdd);
18     }
19 }
20
21 public class Bowl {
22     private int gramsOfFoodInBowl;
23
24     public void addFood(int grams) throws Exception {
25         gramsOfFoodInBowl = gramsOfFoodInBowl + grams;
26     }
27 }
```

Listing 1.4 The `bowl` object is created in the `main` method and a reference to it is passed to the `person` object, where it is used.

1.3 Arrays and Lists

An ordered collection of elements can be represented both as the language construct *array* and as an instance of `java.util.List`. An array is appropriate if the number of elements is both fixed and known, see listing 1.5, where there are exactly five elements.

```
1 int[] myArray = new int[5];
```

Listing 1.5 An array has an exact number of elements, five in this case.

It is better to use a `java.util.List` if the number of elements is not both fixed and known, see listing 1.6.

```
1 import java.util.ArrayList;
2 import java.util.List;
3 ...
4   List myList = new ArrayList();
5   myList.add("Hej");
6   myList.add(3);
7   String listElement = (String)myList.get(0);
```

Listing 1.6 Any number of elements can be added to a `List`.

A `List` can contain objects of any class, listing 1.6 stores a `String` on line five and an `Integer` on line six. This means that when reading from the `List`, the type of the read element will always be `java.lang.Object`. It is up to the programmer to know the actual type of the element and cast it to that type, as is done on line seven in listing 1.6. This procedure is error-prone, it is better to restrict list elements to be objects of one specific type. This is done in listing 1.7, where adding `<String>` on line four specifies that the list may contain only objects of type `String`. Adding `<>` specifies that this holds also for the created `ArrayList`. As can be seen on line seven, elements that are read from the list are now of type `String`, and no cast is required.

```
1 import java.util.ArrayList;
2 import java.util.List;
3 ...
4   List<String> myList = new ArrayList<>();
5   myList.add("Hej");
6   myList.add("Hopp");
7   String listElement = myList.get(0);
```

Listing 1.7 A `List` allowed to contain only objects of type `String`.

When list content is restricted to one type, it is possible to iterate the list using a for-each loop, see lines eight to ten in listing 1.8.

```
1  import java.util.ArrayList;
2  import java.util.List;
3  ...
4    List<String> myList = new ArrayList<>();
5    myList.add("Hej");
6    myList.add("Hopp");
7
8    for(String value : myList) {
9        System.out.println(value);
10   }
```

Listing 1.8 Iterating a `List` with a for-each loop

1.4 Exceptions

Exceptions are used to report errors. When an exception is thrown, the method throwing it is immediately interrupted. Execution is resumed in the nearest calling method with a `try` block. This is illustrated in listing 1.9. On line 34, the `addFood` method checks if the bowl would become overfull when more food is added. If so, instead of adding food, it throws an exception (lines 35-40). This means line 42 is not executed. Instead, the program returns to the calling statement, which is on line 24, in the `feedDog` method. However, that line is not in a `try` block, which means execution returns to the statement where `feedDog` was called. That call is made on line eight, which is in a `try` block. Execution then jumps immediately to the corresponding `catch` block. This means line eleven is the line executed immediately after throwing the exception on line 35.

```
1  public class Startup {
2      public static void main(String[] args) {
3          Bowl bowl = new Bowl();
4          Person person = new Person(bowl);
5          try {
6              for (int i = 0; i < 9; i++) {
7                  System.out.println("Feeding dog");
8                  person.feedDog();
9              }
10         } catch (Exception e) {
11             e.printStackTrace();
12         }
13     }
14 }
```

```
15 public class Person {
16     private Bowl bowl;
17     private int gramsToAdd = 200;
18
19     public Person(Bowl bowl) {
20         this.bowl = bowl;
21     }
22
23     public void feedDog() throws Exception {
24         bowl.addFood(gramsToAdd);
25     }
26
27 }
28
29 public class Bowl {
30     private int gramsOfFoodInBowl;
31     private static final int MAX_CAPACITY = 500;
32
33     public void addFood(int grams) throws Exception {
34         if (gramsOfFoodInBowl + grams > MAX_CAPACITY) {
35             throw new Exception("Bowl is overfull. " +
36                                 "Trying to add " +
37                                 grams + " grams of " +
38                                 food when there are " +
39                                 gramsOfFoodInBowl +
40                                 " grams in bowl.");
41         }
42         gramsOfFoodInBowl = gramsOfFoodInBowl + grams;
43     }
44 }
```

Listing 1.9 An exception is thrown if food is added to the bowl when it is already full.

All methods in listing 1.9 that may throw an exception declare that, with `throws Exception` in the method declaration. This is required if the thrown exception is a checked exception, but not if it is a runtime exception. An exception is a runtime exceptions if it inherits the class `java.lang.RuntimeException`.

1.5 Javadoc

Javadoc is used to generate html pages with code documentation, like the documentation of the Java APIs at `http://docs.oracle.com/javase/8/docs/api/`. It is strongly recommended to write Javadoc for all declarations (classes, methods, fields, etc) that are not private. A Javadoc comment is written between `/**` and `*/`. The tags `@param` and `@return` are used to document method parameters and return values. See listing 1.10 for examples.

```
1  /**
2   * A person that lives at the specified address.
3   */
4  public class Person {
5      private String homeAddress;
6
7      /**
8       * Creates a new <code>Person</code>.
9       */
10     public Person() {
11         this(null);
12     }
13
14     /**
15      * Creates a new <code>Person</code> that lives at the
16      * specified address.
17      *
18      * @param homeAddress The newly created
19      *                    <code>Person</code>'s home address.
20      */
21     public Person(String homeAddress) {
22         this.homeAddress = homeAddress;
23     }
24
25     /**
26      * @return The <code>Person</code>'s home address.
27      */
28     public String getHomeAddress() {
29         return this.homeAddress;
30     }
31
32     /**
33      * The the <code>Person</code> moves to the specified
34      * address.
35      *
36      * @param newAddress The <code>Person</code>'s new
37      *                   home address.
38      */
39     public void move(String newAddress) {
40         this.homeAddress = newAddress;
41     }
42 }
```

Listing 1.10 Class with javadoc comments

1.6 Annotations

Annotations are code statements that are not executed. Instead, they provide information about a piece of source code for the compiler, JVM or something else. Annotations are usually used for properties unrelated to the functionality of the source code, for example to configure security, networking or tests. An annotation starts with the at sign, `@`, for example `@SomeAnnotation`. Annotations may take parameters, for example `@SomeAnnotation( someString = "abc")`. An example is found on line 20 in listing 1.11.

1.7 Interfaces

An interface is a contract, specified in the form of method declarations. A class implementing the interface must fulfill the contract, by providing implementations of the methods. A method implementation in an implementing class must do what is intended by the method declarations in the implemented interface, which should be clarified in a javadoc comment. Note that the interface contains only declarations of methods, there are no method bodies. Listing 1.11 shows an interface that defines the contract *Print the specified message to the log*, lines one to eleven. It also shows a class that implements the interface and fulfills the contract, lines 12-24.

```
1  /**
2   * An object that can print to a log.
3   */
4  public interface Logger {
5      /**
6       * The specified message is printed to the log.
7       * @param message The message that will be logged.
8       */
9      void log(String message);
10 }
11
12 /**
13  * Prints log messages to <code>System.out</code>.
14  */
15 public class ConsoleLogger implements Logger {
16     /**
17      * Prints the specified string to <code>System.out</code>.
18      * @param message The string that will be printed.
19      */
20     @Override
21     public void log(String message) {
22         System.out.println(message);
23     }
24 }
```

Listing 1.11 Interface and implementing class

The `@Override` annotation on line 20 in listing 1.11 specifies that the annotated method should be inherited from a superclass or interface. Compilation will fail if the method is not inherited. Always use `@Override` for inherited methods since it eliminates the risk of accidentally specifying a new method, for example by accidentally naming the method `logg` instead of `log` in the implementing class.

1.8 Inheritance

When a class inherits another class, everything in the inherited class that is not private becomes a part also of the inheriting class. The inherited class is often called *superclass* and the inheriting class is called *subclass*. This is illustrated in listing 1.12, where `methodInSuperclass` is declared in `Superclass` on line two, but called on line eleven as if it was a member of `Subclass`. Actually, it has become a member also of `Subclass`, because it has been inherited.

```
1  public class Superclass {
2      public void methodInSuperclass() {
3          System.out.println(
4          "Printed from methodInSuperclass");
5      }
6  }
7
8  public class Subclass extends Superclass {
9      public static void main(String[] args) {
10         Subclass subclass = new Subclass();
11         subclass.methodInSuperclass();
12     }
13 }
```

Listing 1.12 `methodInSuperclass` exists also in the inheriting class, `Subclass`.

A method in the subclass with the same signature as a method in the superclass will *override* (*omdefiniera*) the superclass' method. This means that the overriding method will be executed instead of the overridden. A method's signature consists of its name and parameter list. In listing 1.13, the call to `overriddenMethod` on line 16 goes to the method declared on line nine, not to the method declared on line two. Do not confuse overriding with *overloading*, which is to have methods with same name but different signatures, due to different parameter lists. This has nothing to do with inheritance.

```
1  public class Superclass {
2      public void overriddenMethod() {
3          System.out.println("Printed from overriddenMethod" +
4                             " in superclass");
5      }
6  }
```

```
7
8  public class Subclass extends Superclass {
9      @Override
10     public void overriddenMethod() {
11         System.out.println("Printed from overriddenMethod" +
12                            " in subclass");
13     }
14
15     public static void main(String[] args) {
16         Subclass subclass = new Subclass();
17         subclass.overriddenMethod();
18     }
19 }
```

Listing 1.13 `overriddenMethod` in `Superclass` is overridden by the method with the same name in `Subclass`. The printout of this program is *Printed from overridden-Method in subclass*

The keyword `super` always holds a reference to the superclass. It can be used to call the superclass from the subclass, as illustrated on line ten in listing 1.14.

```
1  public class Superclass {
2      public void overridenMethod() {
3          System.out.println("Printed from Superclass");
4      }
5  }
6
7  public class Subclass extends Superclass {
8      public void overridenMethod() {
9          System.out.println("Printed from Subclass");
10         super.overridenMethod();
11     }
12
13     public static void main(String[] args) {
14         Subclass subclass = new Subclass();
15         subclass.overridenMethod();
16     }
17 }
```

Listing 1.14 Calling the superclass from the subclass. This program prints *Printed from Subclass*, followed by *Printed from Superclass*

To declare a class means to define a new type, therefore, the class named `Subclass` of course has the type `Subclass`. When inheriting, the subclass will contain all methods and fields of the superclass. Thus, the subclass will also have the type of the superclass, the subclass in fact becomes also the superclass. This means that an instance of the subclass can be assigned to a variable of the superclass' type, see line 18 in listing 1.15. When a method is

called, as on line 19, the assigned instance is executed, not the declared type. This means the method call goes to the method declared on line ten, not to the method declared on line two.

```
1  public class Superclass {
2      public void overriddenMethod() {
3          System.out.println("Printed from overriddenMethod" +
4                             " in superclass");
5      }
6  }
7
8  public class Subclass extends Superclass {
9      @Override
10     public void overriddenMethod() {
11         System.out.println("Printed from overriddenMethod" +
12                            " in subclass");
13     }
14
15     public static void main(String[] args) {
16         Subclass subclass = new Subclass();
17         subclass.overriddenMethod();
18         Superclass superclass = new Subclass();
19         superclass.overriddenMethod();
20     }
21 }
```

Listing 1.15 Calling a method in an instance of the subclass, that is stored in a field of the superclass' type. This program prints *Printed from overriddenMethod in subclass*, followed by *Printed from overriddenMethod in subclass*

Chapter 2

Introduction

Before starting with object oriented analysis and design, it is necessary to understand how those activities fit in the software development process. This chapter gives a general understanding of different activities performed in a programming project, and explains when and why to do analysis and design.

2.1 Why Bother About Object Oriented Design?

Being able to change the front door of my house does not make me a carpenter, being able to change spark plugs of my car does not make me a car-mechanic. Similarly, being able to write a program that works when run by myself, on my own computer, does not have much to do with being a professional software developer. On the contrary, professional software development means to write code that can be maintained, changed and extended, in order to meet the user's expectations for a long period of time. This should hold even if developers working with the code leave, and new developers arrive. To write such code, we need the principles of object oriented design.

To be more specific, the goal of object oriented design is to write code that enables changing the application's behavior by changing as little code as possible, and absolutely only code performing the task that shall be changed. It shall also be possible to extend the application's functionality without having to modify existing code. To reach this goal, the code must have two important properties. First, it must be *flexible*, which means changes in one part of the code does not require further changes in other parts of the program. Second, it must be *easily understood*, structure and function shall be evident to anyone who reads the code. If the program is not easily understood, other developers might unintentionally modify it the wrong way, thereby destroying the flexibility it originally had.

2.2 Software Development Methodologies

Many software development projects have faced serious problems, for example being too expensive, being delayed or producing bad software due to bugs or lack of functionality. To remedy these problems, there are many different sets of guidelines describing how to organize a programming project the best way. Such a set of guidelines is called a *software development*

methodology. This section covers some important principles agreed on by all commonly used software development methodologies.

Software development must be iterative. During an iteration a limited amount of new functionality is developed, or existing functionality is modified, or bugs in the code are corrected, or some combination of these. What is important is that the work is completely finished when the iteration is over. Iterations shall be relatively short, typically one or two weeks. The reason for working like this is that it is only at the end of an iteration we really know the status of the program being developed. There is no point in claiming that something is almost done, either it *is* done, 100 percent ready, or it is *not* done. Each iteration is like a mini project, which contains modifying requirements on the program, analysis, design, coding, testing, integrating new code with previously developed code, and evaluating the result together with clients and/or users.

Manage risks early in the project. Code that is difficult to develop must be developed during the first iterations. If not, it will be very difficult to make a reliable time plan for the rest of the project, since we postpone work we do not really know how to perform. It might even be impossible to write the difficult code. In that case, all work done in the project before this is discovered is wasted, since either the project must be canceled or the program must be rewritten.

Be prepared for changes. It is not possible to write a perfect specification of the program before development starts. Both clients and the developers will come up with new, or changed, ideas when they see the program. Therefore, there must be a procedure for managing changing requirements. Actually, changes should be encouraged by working close to the client and regularly demonstrating and discussing the program. This way the final result will be much better and clients much happier than if developers try to oppose changes and force clients to make up their minds once and for all.

Write extensive tests, and run them often. To make it easy and quick to run tests, they should be automated. That means there should be a test program which gives input to the program under test, and also evaluates the output. If a test passes, the test program does not do anything. If a test fails, it prints an informative message about the failure. With extensive tests that cover all, or most, possible execution paths through the program with all, or most, possible input values, it is guaranteed that the program works if all tests pass. This is a *very* good situation, one command starts the test, which tells if the program under test works or, if not, exactly which problems there are. This makes it easy to change the program, the test will immediately tell if it still works after the change. Without tests, on the other hand, it will be a nightmare to change the code since there is no easy way to make sure it still works.

2.3 Activities During an Iteration

Independent of software development methodology being used, the following activities are typically performed during each iteration.

Requirements analyses is the process of identifying required functionality of the software being developed. This process can not be finished early on in the project, but must be continued in each iteration. This is because clients can not be expected to know in detail what the

program shall do, before development starts. Both users and developers will come up with new, or changed, ideas when trying early versions of the program. Therefore, it is import to work close to users and frequently discuss the functionality. In particular, each iteration shall start with discussing requirements. Requirements analyses is not covered further in this text.

Analysis means to create a model, a simplified view, of the reality in which the system under development shall operate. That model shall not describe the program being developed, but just the reality in which the program operates. The purpose is to gain a better understanding of this reality, *before* thinking about the program. Analysis is introduced in chapter 4.

Design is an activity where we reflect on the code that shall be developed and create a plan that gives a clear understanding of which classes and methods the code will contain, and how they will communicate. To write a program without a plan is as inadequate as building a house without a plan. Design is covered in chapters 5, 8 and 9.

Coding is of course the most important part of development, it is code quality alone that decides if the program works a intended. The other activities have no other purpose than to improve the quality of the code. This does not mean that the other activities can be neglected, it is impossible to develop code of high quality without carefully performing all other activities. Guidelines for writing high-quality code are covered in chapter 6.

Testing shall, as described above, be automated and extensive. Tests that are easy to execute and clearly tell the state of the program are extremely valuable. They facilitate development immensely since they make developers confident that the program works, also when changing or adding code. Testing is covered in chapter 7.

Integrate means to add newly developed code to a repository with all previously developed code, and to verify that both new and previously developed code still work as intended. The bigger the program and the more developers involved, the harder this process is and the more important that it is well defined how to do it. Extensive and automated tests help a lot. Integration is not covered further in this text.

Evaluation of code that was written during an iteration, is an important last activity of the iteration. An iteration can not be ended without demonstrating the program to the clients and gathering their opinions. The client's opinions are added to the requirements and are managed in coming iterations. This is not covered further in this text.

These are the main activities performed during each iteration, a typical iteration length is one or two weeks. However, each developer shall also have a smaller, personal iteration, which consists of designing, coding, testing and integrating. These four activities make an indivisible unit of work, coding shall never be done alone without the other three activities. Design is needed to organize the code and make sure it has the two required properties being easy to modify and being easy to understand. Testing is needed to make sure the code works as intended. Tests are also needed to show that the code still works as intended after coming iterations. Integration with other code is needed because code parts are of no use unless they work together.

2.4 Unified Modeling Language, UML

Both analysis and design result in plans. The results of analysis are plans of the parts of the reality that are relevant to the program, and the results of design are plans of the code that shall be written. These plans must contain symbols of classes, methods, etc, and to understand each other's plans we must agree on the symbols being used. To define those symbols is the purpose of the unified modeling language, UML. UML is a vast standard, this text covers only the small fraction needed to draw the plans that will be developed here.

UML defines different types of diagrams and the symbols that can be used in each of those diagrams. Here, we will use class diagrams to give a static picture of something, and sequence or communication diagrams to illustrate events following each other in time. When using UML, it is important to understand that it does not say anything about the meaning of the diagrams or symbols. For example, during analysis we use classes in a class diagram to illustrate things in the reality. During design we use the same class symbols in another class diagram to illustrate classes in an object oriented program. Thus, a class symbol can represent an abstraction in the reality, a class in an object oriented program, or any other thing we choose to let it represent. UML just defines what the symbol looks like.

Chapter 3

The Case Study

This text uses a car rental company as case study to illustrate concepts and activities. More specifically, the implemented functionality is `RentCar`, which describes what happens when a customer arrives at the car rental office to rent a car. The requirements specification follows below.

3.1 Basic Flow

The basic flow, also called *main success scenario*, describes a sequence of events that together make up a successful execution of the desired functionality, see figure 3.1.

1. The customer arrives and asks to rent a car.
2. The customer describes the desired car.
3. The cashier registers the customer's wishes.
4. The program tells that such a car is available.
5. The cashier describes the car to the customer.
6. The customer agrees to rent the described car.
7. The cashier asks the customer for name and address, and also for the driving license.
8. The cashier registers the customer's name, address and driving license number.
9. The cashier books the car.
10. The program registers that the car is rented by the customer.
11. The customer pays, using cash.
12. The cashier registers the amount payed by the customer.
13. The program prints a receipt and tells how much change the customer shall have.
14. The program updates the balance.
15. The customer receives receipt, change and car keys.
16. The customer leaves.

Figure 3.1 The basic flow of the `RentCar` case study.

3.2 Alternative Flows

An alternative flow describes a deviation from the basic flow. This requirements specification has currently only one alternative flow, figure 3.2, which describes what happens if there is no car matching the customer's wishes.

4a. The program tells that there is no such car available.
1. The cashier tells the customer that there is no matching car.
2. The customer specifies new wishes.
3. Execution continues from bullet three in basic flow.

Figure 3.2 An alternative flow for the `RentCar` case study.

Part II

Course Content

Chapter 4

Analysis

The purpose of analysis is to gain a better understanding of the reality in which the program operates. That is achieved by creating a model, a simplified view, of the reality. It is essential to understand that the model created during analysis does not describe the program being developed, but just the reality in which that program operates. This chapter shows how to create such a model, consisting of a *domain model* and a *system sequence diagram*. It also covers the UML needed for those two diagrams.

4.1 UML

This section introduces the UML needed for the domain models and system sequence diagrams drawn in this chapter. The UML diagrams used are *class diagram* and *sequence diagram*. More features of these diagrams are covered in following chapters.

It can not be stressed enough that UML does not say anything about the meaning of diagrams or symbols. For example, a UML class in a UML class diagram is just that: A UML class. It can represent something in the real world, like a chair, it can represent something in a program, like a Java class, or it can represent something completely different.

When drawing a UML diagram, the meaning of the diagram and its symbols must be defined. That is why specific diagrams have specific names, for example domain model. When a diagram is given a well-defined name, everyone knows what it depicts and what its symbols represent.

Class Diagram

A class diagram gives a static picture of something. It shows no flow or progress in time, but only what classes there are, what they contain and how they are connected to each other. There is no notion at all of time in a class diagram.

The content of a class diagram might be a snapshot showing how things look at a particular instant in time, or it might be the sum of everything that has existed during a specific time interval, or it might be everything that will ever exist. This must be defined by the diagram author.

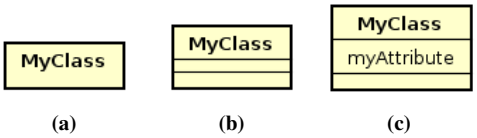


Figure 4.1 Symbols for a class in a class diagram.
(a) Without attribute and operation compartments.
(b) Empty attribute and operation compartments.
(c) With an attribute.

A class in UML represents an abstraction, a concept or idea, and is not associated with any specific instance of that abstraction. A class `Person` specifies which properties a person has. It does not say anything about the values of those properties for specific instances, like the persons `me` or `you`. A class name is always a noun in singular. Figure 4.1 shows three possible ways to draw a class in a class diagram. The first example, figure 4.1a, specifies only the name of the class, `MyClass`. The second, figure 4.1b, also specifies only the class name, but has two empty compartments below the name. The upper of these is for specifying attributes, which defines properties of instances of the class. The bottom compartment, which is empty in all classes in figure 4.1, is for operations. During analysis, there will not be any operations, therefore this compartment will always be empty in this chapter. Finally, figure 4.1c shows a class that has the attribute `myAttribute`.

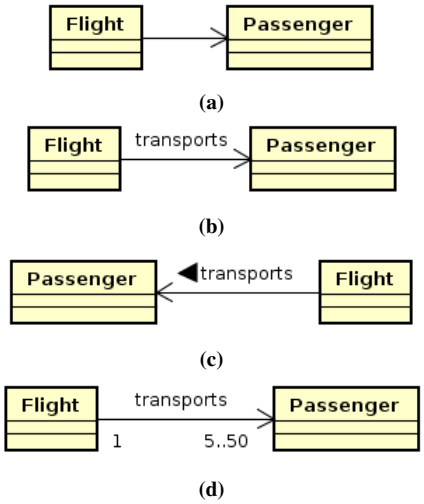


Figure 4.2 Associations
(a) Unidirectional
(b) Unidirectional named
(c) With name direction
(d) With multiplicity

Classes can have *associations* with other classes. An association between two classes means that instances (objects) of those classes are linked. If the diagram depicts classes in an object oriented program, it means that one object has a reference to the other object. If the classes depicts entities in the real world, it means that instances have some kind of relation.

Figure 4.2 shows some ways an association can be illustrated in a class diagram. Figure 4.2a shows an association with a direction. When drawn like that, with an arrow, the association exists only in the direction of the arrow, `Flight` has an association with `Passenger`, but not vice versa. There can be arrows on both ends, meaning that both classes have an association with the other class. There can also be no arrow at all, which means that direction is not considered. If there is no arrow at all, the diagram author chose not to tell the direction of the association.

In figure 4.2b, the association has a name, to clarify its meaning. If there is a name, the sequence *origin class name*, *association name*, *target class name* should make sense and convey a message illustrating the interaction of those three elements. This means the association name shall be a verb. In figure 4.2b, the message is `Flight transports Passenger`.

The black triangular arrow in figure 4.2c shows in which direction the class-association-class sequence shall be read, it does not tell anything about the association's direction. It is up to the diagram author to decide if such black triangles shall be used or not. They are most commonly used if class-association-class shall be read from right to left, or bottom up.

There are many different kinds of arrows in UML, and they all have different meanings. You are not allowed to choose any kind of arrow, they must look exactly as in figure 4.2.

Figure 4.2d tells how many instances of each class are involved in the association. In this example, there is exactly one instance of `Flight`, and five to fifty instances of `Passenger`. This means all passengers travel with the same flight, which can take a maximum of fifty passengers. Also, the flight will not take place if there are less than five passengers. It is possible to use the wildcard, $\star$, when specifying the number of instances. It means any number, including zero.

Sequence Diagram

A sequence diagram shows how instances send messages to each other. The UML term is *message*, not method call. The messages in the diagram form a sequence of events, that happen in a specified order in time.

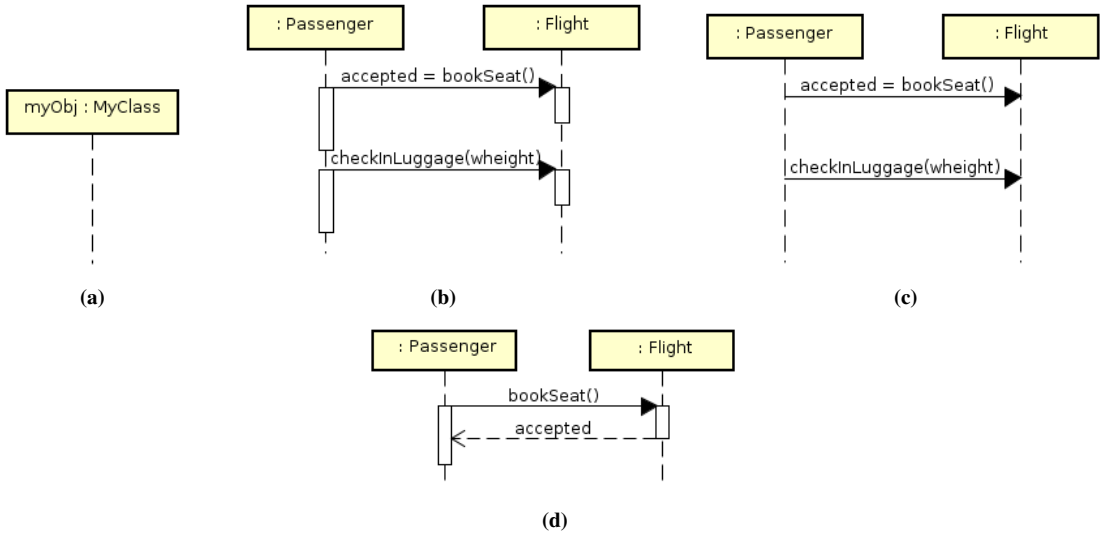


Figure 4.3 Instances and messages in sequence diagram
(a) The instance **`myObj`** of **`MyClass`**
(b) Messages with activation bar
(c) Messages without activation bar
(d) Using reply message to show return value

Figure 4.3a shows how to draw an instance. The word before the colon, `myObj`, is the name of the instance and the word after the colon, `MyClass`, is the name of the class. Both names are optional. The dashed line, called lifeline, is where messages to and from the instance are anchored. Figure 4.3b shows communication between two objects, time flows from top to bottom. The first message is `bookSeat`, which is followed by `checkInLuggage`. The message `bookSeat` has a return value, which has the name `accepted`. The message `checkInLuggage` has a parameter, which has the name `weight`. The thicker parts of the lifelines are called *activation bar*, and means the instance is active during that period in time.

If the sequence diagram depicts an object oriented program, the extent of an activation bar corresponds to execution of a method. Figure 4.3c illustrates exactly the same as 4.3b, but without activation bars. This format is preferred if it is not important to show when instances are active. Finally, figure 4.3d shows exactly the same `bookSeat` message as figures 4.3b and 4.3c, but uses a *reply message* to illustrate that a value is returned. The return value notations in figures4.3b, 4.3c and 4.3d all have exactly the same meaning. Which one to use is just a matter of taste.

Remember that different arrows have different meanings. The arrows must look exactly as in figure 4.3. Generally, most things in UML are optional, but if used they must look exactly as defined in the specification.

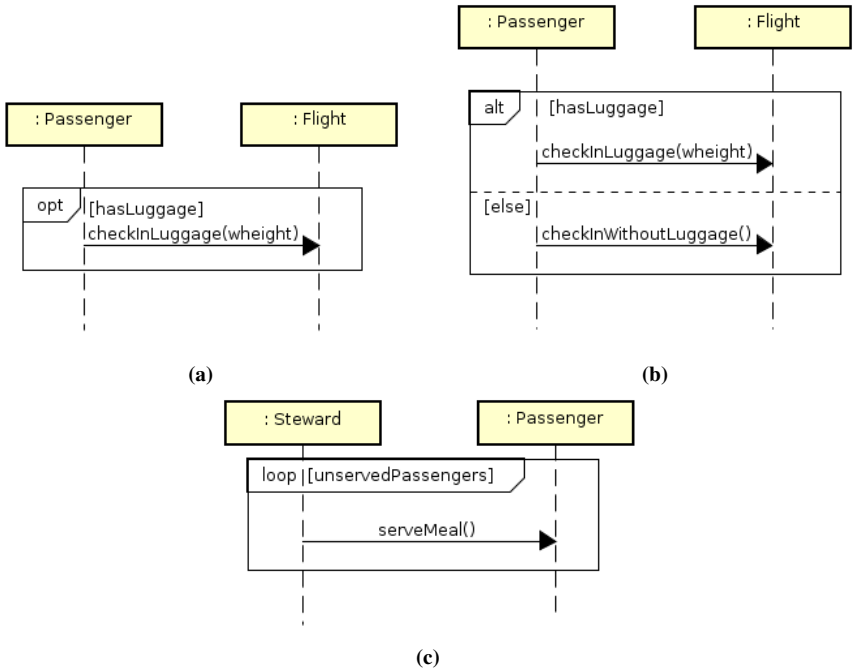


Figure 4.4 Conditions and loops in sequence diagram
(a) An `if` statement (b) An `if else` statement (c) A loop

Flow control is illustrated with *combined fragments*, which are the boxes drawn around the messages in figure 4.4. A combined fragment consists of an *interaction operator* and an *interaction operand*. The operators used here are `opt`, which illustrates an `if` statement, see figure 4.4a; `alt`, which illustrates an `if else` statement, see figure 4.4b; and `loop`, which illustrates an iteration, see figure 4.4c. The operands are the boolean expressions in square brackets. In this example, figure 4.4a says that the passenger checks in luggage if the operand `hasLuggage` is true. Figure 4.4b says that the passenger checks in luggage if `hasLuggage` is true, and checks in without luggage if `hasLuggage` is false. Finally, figure 4.4c says that the steward continues to serve meals while `unservedPassengers` is true. UML does not specify operand syntax, any text is allowed in an operand.

To avoid confusion, it is always important to follow naming conventions. UML has multiple sets of naming conventions, the conventions used here are the same as in Java. Class names are written in pascal case, `LongDisctanceFlight`; object names, attribute names, method names and variable names are written in camel case, `economyClassPassenger`, `luggageCount`, `checkInLuggage`, `unservedPassengers`.

!

Notes

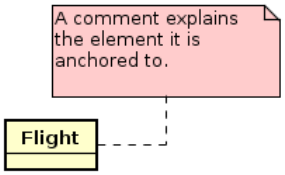


Figure 4.5 A UML comment

Both class and sequence diagrams (and all other UML diagrams) can have comments, see figure 4.5. A comment is an explaining text, a note to the reader, that is not part of any of the elements in the diagram. Comments are called *notes* in UML. A note is anchored to the element it explains with a dashed line, as in figure 4.5.

4.2 Domain Model

The *domain model*, DM, is a model of the reality (the *domain*) that shall be represented in the program under development. Remember that it does not represent the program itself, it could in fact be created by a person completely ignorant of programming. A UML class diagram is used to construct the domain model, but the elements in the DM represent things that exist in reality, not classes in an object oriented program. Therefore, it might be better to call them *entities* instead of classes.

The DM is a very good tool for discussions about the program that is being developed. It can ensure that all parties (developers, clients, users, etc) share a common view of the tasks of the program. Also, the process of developing the DM ensures that discussions about the program do take place, and that all parties develop a common nomenclature.

Another benefit of the DM is that, although it does not depict the program, it will still prove very useful when constructing the program. Since the DM is a diagram of the reality being modeled in software, the software should be quite similar to the DM to be an appropriate model of the reality. This will also make sure that class names in the software means something to domain experts.

Step 1, Use Noun Identification to Find Class Candidates

The first, and most important, step when creating a domain model, is to find as many class candidates as possible. Two complementary methods are used to find classes, noun identification and category list. It is preferred to use both methods, in order not to miss any classes.

It is far more common to have too few classes than to have too many. It is also far more problematic to have too few classes, since it is much easier to cancel existing classes than to find new ones.

!

The first method for finding class candidates, *noun identification*, means simply to identify all nouns in the requirements specification, they are all class candidates. Below, in figure 4.6, is the requirements specification for the `Rent Car` case study with all nouns in bold.

1. The **customer** arrives and asks to rent a **car**.
2. The **customer** describes the desired **car**.
3. The **cashier** registers the **customer**'s **wishes**.
4. The **program** tells that such a **car** is available.
5. The **cashier** describes the **car** to the **customer**.
6. The **customer** agrees to rent the described **car**.
7. The **cashier** asks the **customer** for **name** and **address**, and also for the **driving license**.
8. The **cashier** registers the **customer**'s **name**, **address** and **driving license number**.
9. The **cashier** books the **car**.
10. The **program** registers that the **car** is rented by the **customer**.
11. The **customer** pays, using **cash**.
12. The **cashier** registers the **amount** payed by the **customer**.
13. The **program** prints a **receipt** and tells how much **change** the **customer** shall have.
14. The **program** updates the **balance**.
15. The **customer** receives **receipt**, **change** and **car keys**.
16. The **customer** leaves.

4a. The **program** tells that there is no such **car** available.

1. The **cashier** tells the **customer** that there is no matching **car**.
2. The **customer** specifies new **wishes**.
3. Execution continues from bullet three in basic flow.

Figure 4.6 The `RentCar` scenario, with nouns in bold.

Since all words in bold are possible classes, each of them is drawn as a class in the first draft of the domain model, figure 4.7. Remember that class names shall always be in singular.

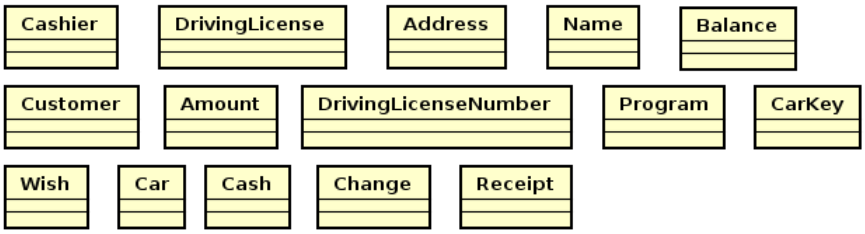


Figure 4.7 The first draft of the domain model, after noun identification

Step 2, Use a Category List to Find More Class Candidates

The second method to find class candidates is to use a *category list*. It is a table where each row specifies a category a class may belong to. The purpose is to stimulate the fantasy, thereby to help find classes that are not nouns in the requirements specification.

The purpose of the category list is *not* to sort classes. There is no point in entering classes already found during noun identification. There is also no point in spending time thinking about which row is correct for a certain class candidate.

NO!

There are many different proposals for categories. Here, the following quite short and simple set is used,

- **Transactions**, selling or buying a product or service
- **Products or services**, what is sold or bought in the transaction
- **Roles** of peoples and organizations involved in the transaction
- **Places**, maybe where a transaction is performed
- **Records of a transaction**, for example contract, receipt
- **Events**, often with a time and place
- **Physical objects**
- **Devices**, are probably physical objects
- **Descriptions** of things
- **Catalogs**, where the descriptions are stored
- **Systems**, software or hardware that is collaborating with the system for which we are creating the DM
- **Quantities and units**, for example length, meter, currency, fee
- **Resources**, for example time, information, work force

The best way to create a category list is to simply consider each row in the category list and try to imagine class candidates belonging to that category. Write down all classes that are found, at this stage it is not interesting if the class is already listed or if it is relevant. Table 4.1 is a category list for the `Rent Car` case study.

Next, all class candidates are added to the domain model, which now looks like figure 4.8.

Step 3, Choose Which Class Candidates to Keep

A question that always tends to be raised is how much it is meaningful to add to the requirements specification. For example, the `Rent Car` specification does not say anything about insurances, which seems to make it a bit far-fetched to include the classes `Insurance` and `InsuranceCost`. In a real project, this should be discussed with the customer. It might be that something is missing in the specification. Here, there is no customer, we have to decide on our own. Remember that it is much better to have too many than too few classes in the DM. Also remember that it is impossible to create a perfect model, there is a limit to how much time it is meaningful to spend. Therefore, if it is really unclear if a class shall be removed or not, just let it stay, at least for now.

Now consider figure 4.8, is there something that ought to be changed, in order to make the DM clearer?

Category	Class Candidates
Transactions	Rental, Payment, Insurance
Products, Services	Car, CarKey, Rental
Roles, People, Organizations	RentalCompany, Customer, Cashier
Places	OfficeAddress, CustomerAddress, CarPickupLocation, CarLeaveLocation
Records	RentalAgreement, Receipt
Events	Rental
Physical objects	Car, CarKey, Office
Devices	
Descriptions	RentalCondition, CarDescription
Catalogs	CarCatalog, RentalCatalog
Systems	
Quantities, units	DrivenDistance, Kilometer, FixedCost, KilometerCost, InsuranceCost, Amount, Currency
Resources	

Table 4.1 Category list for the `Rent Car` case study.

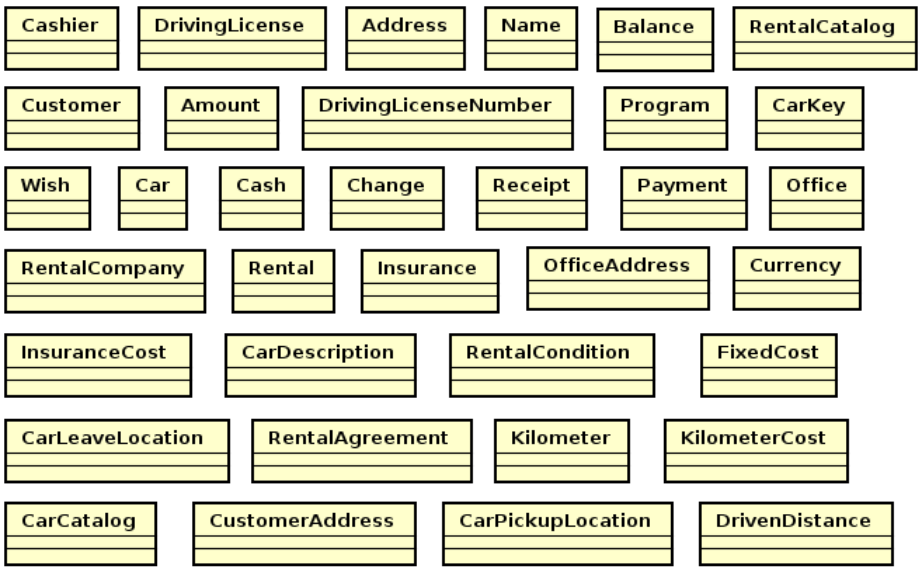


Figure 4.8 The domain model, with classes from the category list added

- The class `Address` is no longer needed, since there are the classes `OfficeAddress` and `CustomerAddress`, and no more addresses need to be specified.
- The class `Program` was created since the requirements specification stated what the

program under development should do, but should it really be included in the DM? The argument against is that the DM shall show only the reality, if `Program` is kept, we have started to think about programming. In fact, all the other classes are a model that shall be present *inside* the program. The argument for, on the other hand, is that the program is in fact present in the reality. If a person, completely ignorant regarding programming, where to write down all entities present in the rental office, the list would include *program* (or system or something similar), since the cashier obviously interacts with the computer.

This problem has no definite answer, it can be discussed endlessly. However, since this text is a first course in analysis, `Program` is removed. The unexperienced developer easily falls into the trap of modeling the program, instead of the reality, if the class `Program` is present.

That is enough for now. If there are more irrelevant classes, they can be removed later, before the DM is finalized.

Step 4, Decide Which Classes Fit Better as Attributes

An attribute is not an entity of its own, but instead a property of an entity. Some classes should not remain classes, but instead be turned into attributes. A simple, but very useful, guideline is that an attribute is a string, number, time or boolean value. A class that contain just one such value is a strong candidate to become an attribute. Another important rule is that an attribute can not have an attribute. Consider for example a class `Address`. It can be represented as a string, and is therefore a candidate to become an attribute. On the other hand, it might be convenient to split it into street, zip code and city. If that is preferred, `Address` must remain a class, to be able to contain the attributes `street`, `zipCode` and `city`. A third rule is that when it is hard to decide if something is an attribute or a class, let it remain a class. Better to have too many classes than too few.

Now consider the domain model of figure 4.8 (remember `Address` and `Program` where removed). Which classes fit better as attributes?

- `DrivingLicenseNumber` is a number (or a string), it can become an attribute of `DrivingLicense`.
- `Name` is a string. Unless it is relevant to split it into first name and last name, it can be an attribute of `Customer`. It can also be an attribute of `Cashier`, if needed.
- Should `CarKey` be an attribute of `Car`? It is true that `CarKey` can be considered to be strongly associated with `Car`, but it is not obvious that `CarKey` is a string, number, time or boolean. Therefore, it remains a class.
- `Amount` is a number, and could become an attribute of `Cash` and `Balance`. But then what about `Currency`? Is that not a string that should be an attribute of `Amount`? This is something that should be discussed with the customer, but now let's just decide we do not need to keep track of currencies. Therefore, `Currency` is removed and `Amount`

becomes attributes of `Cash` and `Balance`, and also of `Change`, `Payment`, `FixedCost`, `KilometerCost` and `InsuranceCost`.

- `OfficeAddress` and `CustomerAddress` could be attributes of `Office` and `Customer`, but according to the reasoning above about addresses, we keep them as classes. These classes should be associated with `Office` and `Customer`, respectively. That will be done below, when considering associations. However, there is no point in creating a new class for each new address. Instead, `OfficeAddress` and `CustomerAddress` are removed, and the single class `Address` is reintroduced.
- Is `CarDescription` an attribute of `Car`? No, since it most likely contains quite a lot of information, like model, model year, size, etc. All this can not be represented as a single string.
- `Kilometer` is is the unit of the quantity `DrivenDistance`. Provided there are no other units, it can be removed. `DrivenDistance` is a number, it becomes an attribute of `Rental`.
- `InsuranceCost` is a number, it becomes an attribute of `Insurance`. `KilometerCost` and `FixedCost` are also numbers, they are turned into attributes of `RentalCondition`.

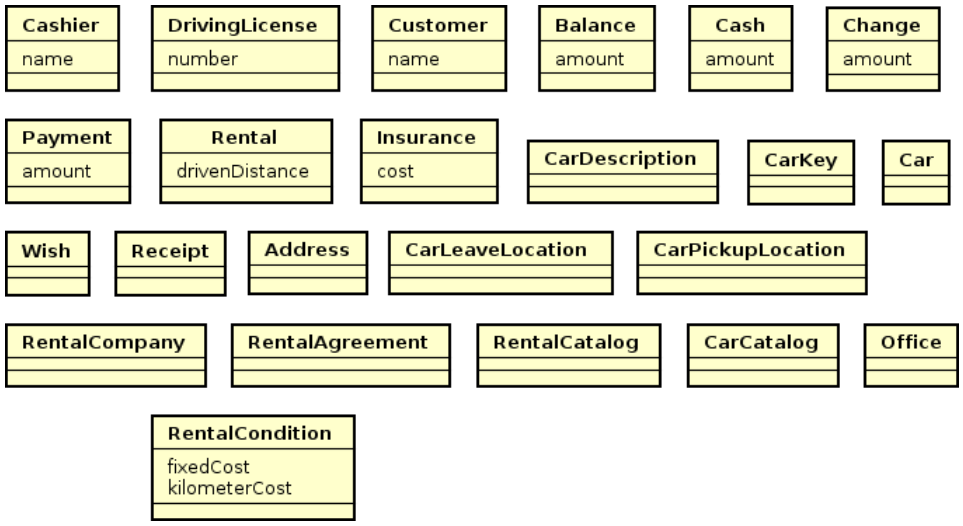


Figure 4.9 The domain model with attributes

That is enough, remember that there is no point in minimizing the number of classes. The domain model with attributes is depicted in figure 4.9.

Step 5, Add Associations

The purpose of associations in the domain model is only to clarify. Therefore, add only associations that actually do clarify the DM. It is almost always possible to find more, no matter

how many there are already. At some point, where more associations are just confusing, it is simply necessary to stop. Also, an association shall always be named, without a name it hardly clarifies at all. Try to avoid the names `has` and `hasA` since it is quite obvious that a class with an association to another class has an instance of that class. More or less all associations could be named `has`. Furthermore, the association name must begin with a verb, since the sequence `origin class name-association name-target class name` shall convey a message illustrating the interaction of those three elements.

It is strictly forbidden to create names that must be read as parts of a chain of associations, for example `Passenger checksIn Luggage at Counter`, which is class-association-class-association-class. It is impossible for the reader to now where such a sentence starts and ends, the reader would probably try to read just `Luggage at Counter`, which does not make sense.

NO!

One more guideline concerning association names is that they shall clarify the meaning of the associated classes. For example, an association named `proves` between classes `Receipt` and `Payment`, tells that `Receipt proves Payment`, which is important information.

On the other hand, if the `Receipt` class is associated with `Customer` by an association named `takes`, it does not clarify why the receipt is important. The association `Customer takes Receipt` just tells what the customer is doing. Try to avoid associations telling what users do, that is instead showed in the System Sequence Diagram, which is explained in the next section.

!

Regarding multiplicity, it is often just confusing, add multiplicity only if it clarifies the DM. Also, do not specify direction, trying to understand the direction of an association in the DM often leads to long and meaningless discussions. The DM depicts the reality, and if two entities in the reality are associated, it is almost always bidirectionally. Finally, there should be at least one association to each class. If it is hard to find an association to a certain class, or if there are different sets of internally associated classes that are not joined by associations, it is a sign that there is something wrong with the DM.

When adding associations to the DM, start with the most central ones. In the case study, which concerns a rental, those could be for example `Customer performs Rental`, `Car isRentedIn Rental`, `Payment pays Rental` and `Car isOwnedBy RentalCompany`. Then continue, following the guidelines above. The result can be seen in figure 4.10.

`Insurance`, `CarPickupLocation` and `CarLeaveLocation` were removed since they were not mentioned in the requirements specification, and the DM is becoming quite big and messy. Also `Cash` was removed. Whether to include it or not is a question of how detailed a payment record shall be. Is it of interest to know how much cash the customer gave to the cashier?

The class `Address` has no association. This is OK for classes that exist just to group data, and do not have a specific meaning but are used in many places. Examples of such classes are `Address`, `Name`, `Amount` and `Coordinate`. The reason is that the DM would be unclear if associations where added to all classes using such data containers. Instead, usage is illustrated by adding the data containers as attributes to classes using them. In figure 4.10, for example `Office` and `Customer` has an attribute `address`, showing that they use the `Address` class.

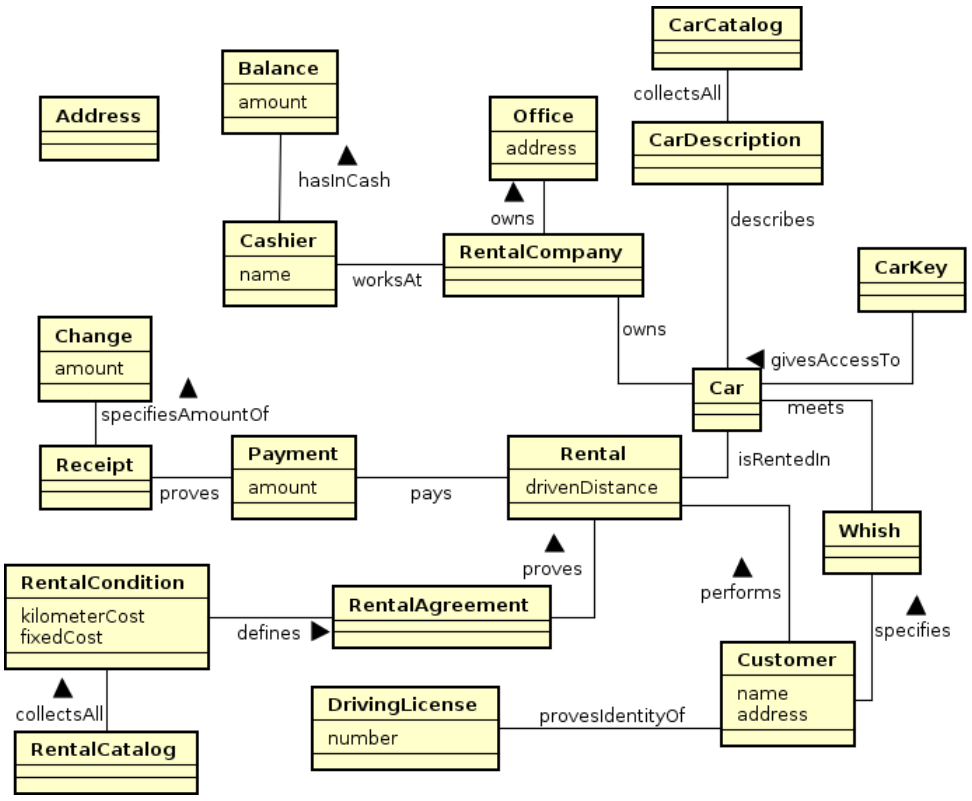


Figure 4.10 The domain model with associations

Step 6, Anything To Change?

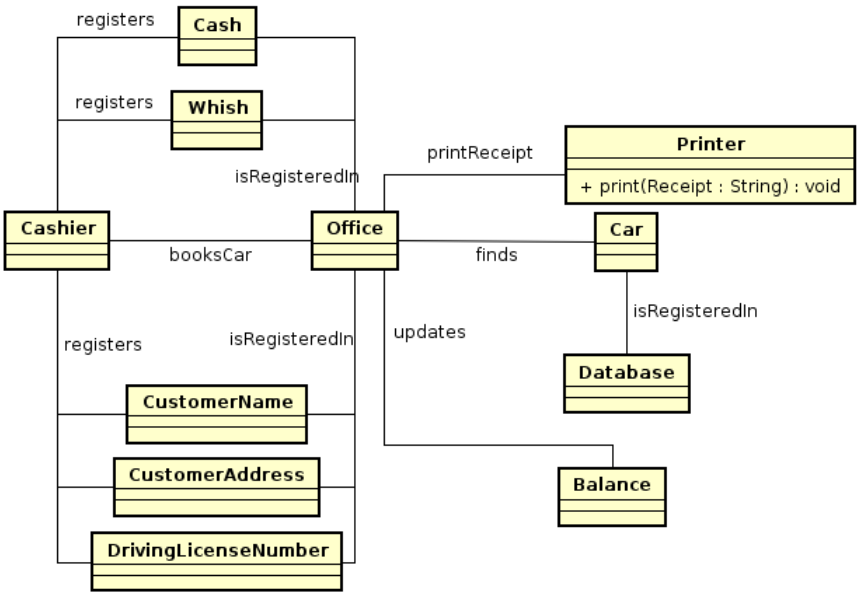
To create the domain model is an iterative process. New classes might be found while considering attributes and associations, attributes might be changed while adding associations, etc. This case study was also performed iteratively, for example was the need for `RentalCatalog` discovered when adding the association between `CarDescription` and `CarCatalog`. Therefore, it is good practice to reconsider the entire DM when done with associations. Here, there is no obvious need for changes, the DM of figure 4.10 becomes the final version.

Common Mistakes

Since creating a domain model is a matter of discussion and, at least to some extent, a matter of opinion, it might be difficult to assess the quality of the resulting DM. There are many ways to create a good model, but also many ways to create a bad model. This section explains some typical mistakes, resulting in a model of low quality. Such a model might not be plainly wrong, but is of little help to the developers.

The first, and most obvious, mistake is not to model reality, but instead regard the DM as a model of a program. This normally also means that some notion of time is assigned to the DM. Things are thought happen in a sequential order, whereas a DM (or any UML class diagram) says absolutely nothing about time or order of events. A class `Program` or `System` often becomes essential in such a "programmatic DM", but be aware that the role of the program can be assigned to any other class as well. Also, an association is considered to be some kind of method call, instead of a relation. Finally, an *actor*, which means any person or other system giving input to or receiving output from the program, becomes the user of the program. The case study has only one actor, namely the cashier. Figure 4.11 shows an example of a "programmatic domain model" where the class `Office` represents the program.

NO!

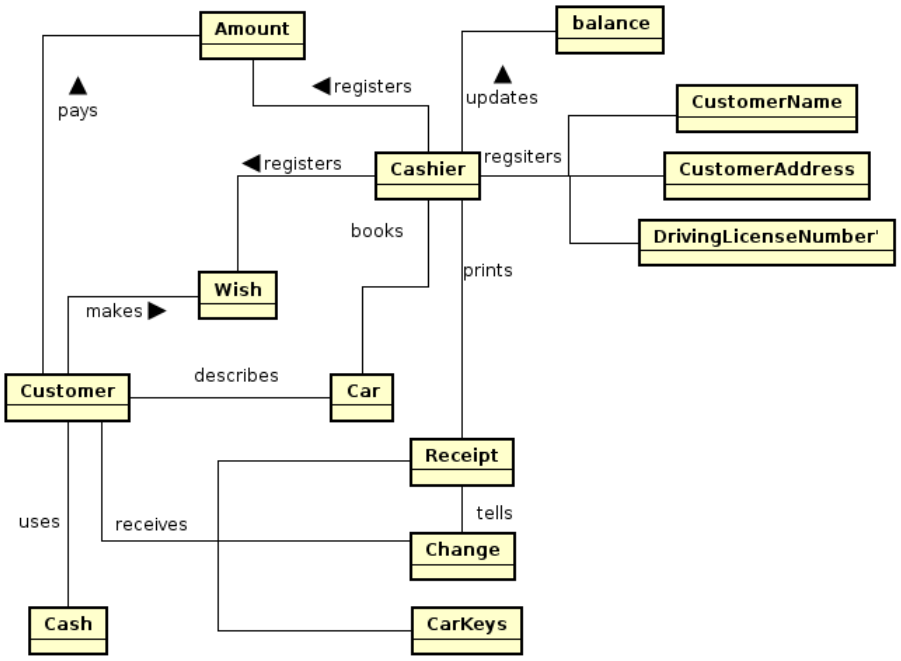


NO!

Figure 4.11 This is not a correct domain model. The modeler has tried to create a program, instead of modeling the reality in which the program acts.

Another, less obvious, mistake, is to create a DM that correctly models the reality, but does not convey any information besides what is already in the requirements specification. In such a "naïve domain model", the actors, customer and cashier in the case study, become central classes with many outgoing associations. Other classes tend to be associated only with one of the actors. This kind of DM is in fact just a visual representation of the specification. It focuses on what the actors do, modeling flow, instead of giving a static picture of what exists. This might not be completely wrong, but adds very little value to what can already be read from the requirements specification. Figure 4.12 is an example of a naïve DM, compared to the DM of figure 4.10, it does not say much. A warning sign that a DM is naïve is that the majority of associations, and the most important associations, tells what an actor is doing, for example `Customer pays Amount` or `Cashier registers Amount`. It is probably necessary to improve the DM if there are many such outgoing associations from actors.

NO!

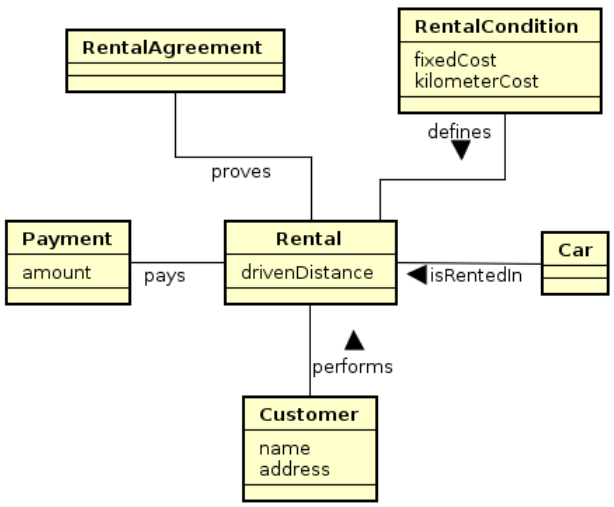


NO!

Figure 4.12 This naïve domain model does not add any extra value, or new information.

A third kind of mistake, besides programmatic and naïve DM, is creating a "spider-in-the-web" class. Such a class has associations to many other classes, while other classes have few associations, especially to classes besides the spider class. A DM with a spider-in-the-web class may still be valuable, but would probably be of higher value if associations were more evenly distributed. The spider class, with many associations, is often difficult to understand. A class with associations to `n` other classes can not be understood without thinking about the `n` other classes. The spider class will have many different responsibilities, each depending on different sets of its associated classes. Also the roles of the peripheral classes, with very few association, might be difficult to understand. They seem to be just "data containers", like primitive values, without any real role to play. It is very difficult to give a definite rule for when a class has become a spider class. A coarse guideline could be that a class should not have more than four or five associations, but that depends on the size and layout of the entire DM. A spider class is made less central by moving an association from it to another class, which is in turned associated with the spider class. As an example, consider the `Rental` class in figure 4.10. It has four associations, but had five in a previous version of the DM, where it looked as in figure 4.13. The association with `RentalCondition` was moved to `RentalAgreement`.

NO!

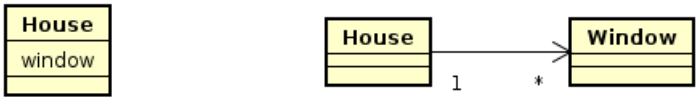


NO!

Figure 4.13 This extract of the `RentCar` case study has a class, `Rental`, with unnecessarily many associations.

Finally, remember that an attribute is a property of a class, not a part. It is a mistake to model parts as attributes. As an example, consider a house, which has windows, doors and rooms. It is not correct to model those as attributes of the house, which is illustrated by the fact that it is very hard to regard a window, door or room as a string, number, time or boolean, which was the guideline for creating an attribute. This reasoning is illustrated in figure 4.14. It is not always obvious whether something is a property or a part. For example, why is a window a part of a house, while a name is a property of a person? A rough guideline can be that the relationship is *part* when the whole actually is constructed of the part, otherwise *property* is better.

NO!



(a) (b)

NO!

Figure 4.14 In **(a)**, window is, erroneously, modeled as an attribute of house. It is better to model it as a class, as in **(b)**.

4.3 System Sequence Diagram

The *system sequence diagram*, SSD, is a sequence diagram showing the interaction between the system under development and the *actors* using it. An actor is any person or other system giving input to, or receiving output from, the system. The SSD must not show anything about the system's internal structure, the entire system must be modeled as one single object. Apart from this `System` object, there is one object for each type of actor. The messages, that is operation that actors can perform on the system, are called *system operations*. A well constructed SSD simplifies development a lot, since it shows exactly what the system can be told to do, and what response it shall give. Strictly speaking, creating an SSD is not part of the analysis, but instead belongs to gathering requirements. Here, we consider it under the analysis section since it is a preparation for program construction.

Do not confuse *system sequence diagram* with *sequence diagram*. Although the names are similar and an SSD is created using a sequence diagram, they are far from synonyms. Sequence diagram is the UML name of a kind of diagram used to illustrate how objects exchange messages, as explained above. It can be used to illustrate any kind of interaction. One specific way to use a sequence diagram is to create an SSD, which is a diagram that illustrates how actors interact with a program, and nothing else.

!

While the domain model is very much a matter of discussion, the SSD is more straight forward to create. It shall reflect the interactions of the requirements specification, no less and no more. It is common to find errors or ambiguities in the specification when constructing the SSD. In that case, the specification might have to be revised, but it is not allowed to let the SSD deviate from it.

Since the SSD shall show the interaction between the system and its actors, the first step is to define where the system ends, and which the actors are. In the case study, it is obvious that we are not developing the cashier, but we are developing the thing the cashier interacts with. Thus, the cashier becomes the actor. Then, look at the requirements specification and identify what the cashier can tell the system to do, and how it responds. The specification is repeated here, in figure 4.15, for the sake of convenience.

Bullets one and two do not contain any interaction between the actor (cashier) and the system. Remember that it is completely uninteresting for the SSD what happens "outside" the actor. Therefore, it would be wrong to include the customer.

In bullet three, there is an interaction that shall be included. A system operation shall have a name that starts with a verb, and describes what is done. The system operation in bullet three can be named `searchMatchingCar`. The name shall not describe what happens

1. The customer arrives and asks to rent a car.
2. The customer describes the desired car.
3. The cashier registers the customer's wishes.
4. The program tells that such a car is available.
5. The cashier describes the car to the customer.
6. The customer agrees to rent the described car.
7. The cashier asks the customer for name and address, and also for the driving license.
8. The cashier registers the customer's name, address and driving license number.
9. The cashier books the car.
10. The program registers that the car is rented by the customer.
11. The customer pays, using cash.
12. The cashier registers the amount payed by the customer.
13. The program prints a receipt and tells how much change the customer shall have.
14. The program updates the balance.
15. The customer receives receipt, change and car keys.
16. The customer leaves.

4a. The program tells that there is no such car available.

1. The cashier tells the customer that there is no matching car.
2. The customer specifies new wishes.
3. Execution continues from bullet three in basic flow.

Figure 4.15 The requirements specification for the `RentCar` case study.

internally, in the system, `searchInDatabase` is therefore not an adequate name. This system operation also takes parameters, namely the customer's description of the desired car. We could write a long list with these parameters, e.g., size, price, model, desired features (for example air condition), etc. The downside of such a solution is that a lot of time would be spent deciding exactly which parameters to include. Also, if the set of parameters would change, we would have to change this system operation. And the set of parameters likely will change, as development continues and the needs the system shall meet become clearer. Therefore, it is better to use an object as parameter. This object, which can be called `wishedCar`, has attributes that define the set of possible wishes. Note that types of parameters and return values, whether primitive or classes, are not of interest in the SSD. Using an object like this, it is not necessary to decide exactly which the wishes might be. Also, if the set of possible wishes changes, that is an internal matter for the class of this object, no changes are required to the system operation.

Now the system operation and its parameters are identified. Next, it must be decided if the operation has a return value, and, if so, which return value? The answer is found in bullet four, which tells that the system gives a positive answer to the search, and in bullet five, which tells that the cashier describes the found car. Considering only bullet four, it might seem adequate to use a boolean return value, but bullet five clearly states that the return value must include a description of the found car. Therefore, also the return value can be an object, which can be

called `foundCar`. The system sequence diagram with this first system operation is depicted in figure 4.16. Note that activation bars are omitted, which is practice in system sequence diagrams. It is not relevant to know when actors and systems are active. Also, trying to decide this tends to lead to long and quite meaningless discussions.

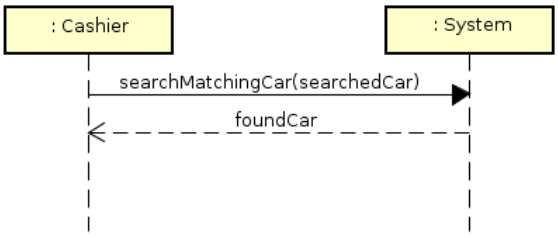


Figure 4.16 The system sequence diagram, after the first system operation has been created.

Continuing the same way, bullet six in the requirement specification does not involve any interaction with the system, neither does bullet seven. Bullet eight defines the second system operation, which can be called `registerCustomer`. An alternative name could be `registerCustomerData`, but it is usually unnecessary to include the word *data*, since more or less all operations include data in some way. The parameters of this operation are name, address and driving license number. These could very well be joined in an object, `customer`, according to the reasoning above, for the `searchMatchingCar` system operation. But since this parameter list is defined exactly in the specification, it is less likely that it changes in the future. The latter alternative is chosen, since that clearly shows that the parameters are known.

Bullet nine is a system operation, `bookCar`. It is a bit unclear if it shall take any parameters. It could be argued that it does not take any parameters, in which case the system must keep track of the car returned in the `searchMatchingCar` operation, and book that same car. It can also be argued that the car to book shall be specified in the `bookCar` operation. If so, the name of the parameter shall be the same as the name of the return value of `searchMatchingCar`, to show that it is in fact the same car. The latter alternative is chosen, mostly since the former would require us to remember that the car returned from `searchMatchingCar` must be stored, and is therefore a bit unclear.

Bullet ten is no system operation. It describes work done internally, inside the system, and does not involve any interaction with an actor. Bullet eleven also is no system operation, since it takes place only between customer and cashier. Bullet twelve is a system operation, it can be called simply `pay`, and take the parameter `amount`. The SSD now looks as in figure 4.17.

In bullet 13, a receipt is printed as a result of the `pay` operation. Does this mean `receipt` is a return value of `pay`? The answer depends on whether the printer is considered to be part of the system under development (SUD), or not. If the printer *is* part of the system, the printing is an internal matter and shall not be included in the SSD. The receipt then becomes a return value of the `pay` operation. On the other hand, if the printer *is not* part of the SUD, it becomes an external system, called from the SUD. This means there is an interaction between the SUD and an external entity, which shall be included in the SSD. The latter alternative is chosen, mainly to illustrate how such an interaction looks, se figure 4.18.

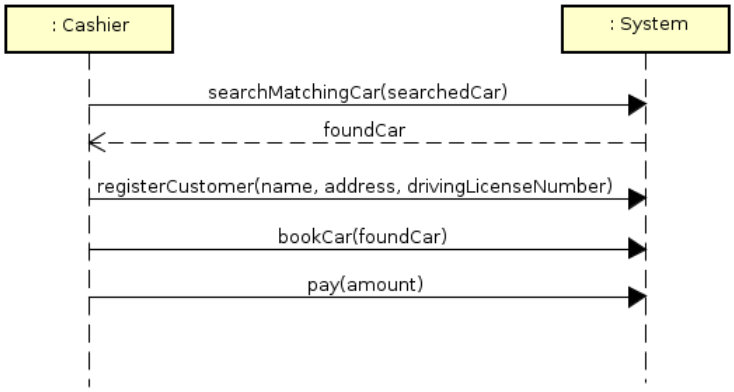


Figure 4.17 The system sequence diagram, with more system operations added.

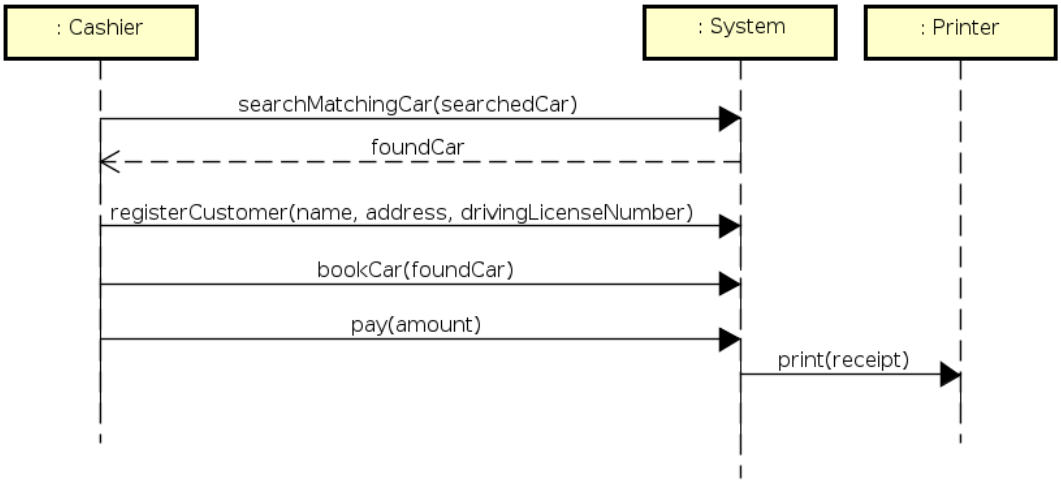


Figure 4.18 The system sequence diagram, with call to external system.

Continuing, bullets 14-16 are either internal or external, and do not imply any interaction between the SUD and its actors. Therefore, they do not generate any system operation. That concludes the basic flow, next the alternative flow is considered. The alternative flow specifies a loop, including bullets two, three and four in the basic flow. The iteration around these bullets continues until a matching car is found. Note that the specification is incomplete, it does not allow the customer to give up and leave without renting a car. Of course this must be changed in coming iterations of the development. The loop can be modeled as in figure 4.19.

There are a two things worth highlighting regarding the loop. First, the guard `noMatchingCar` is a free text boolean condition, it does not correspond to a boolean expression in the program. The condition can become true because of an action taken by the system, the actor or something completely independent of both system and actor. Second, drawing the return value `foundCar` inside the loop, as in figure 4.19, implies it can indicate both that a matching car was found and that such a car was not found. How this is done is not shown is the SSD.

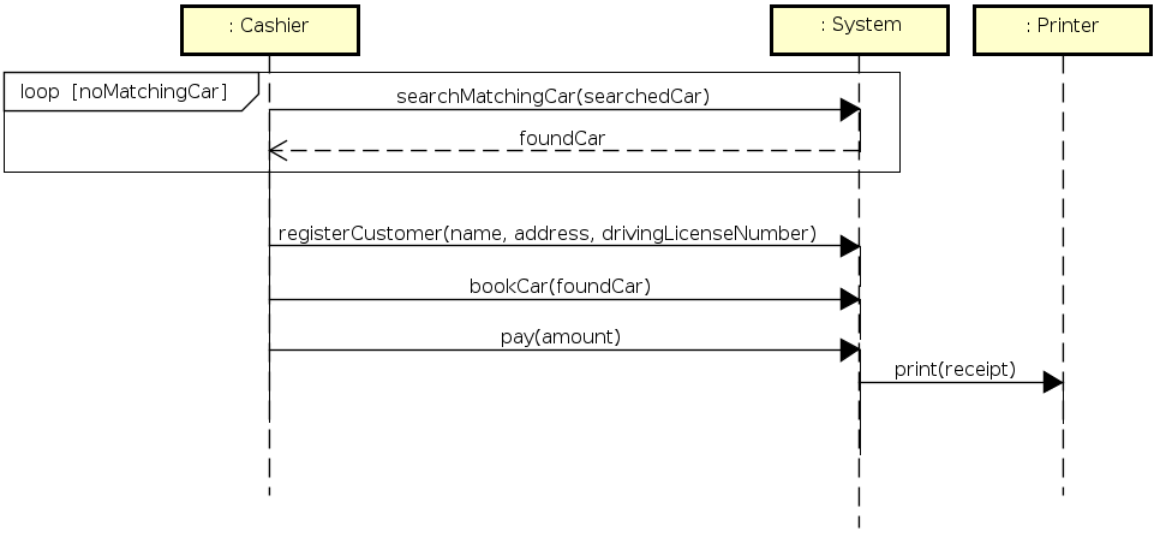


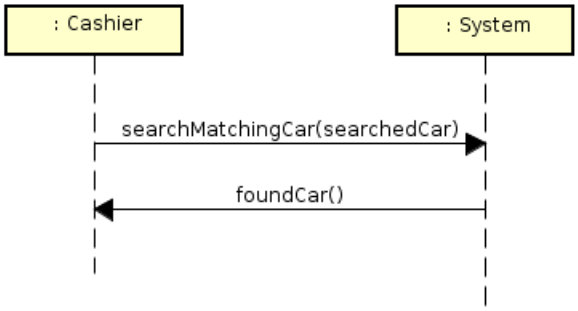
Figure 4.19 The system sequence diagram, with a loop reflecting the alternative flow.

Common Mistakes

As mentioned above, there is not so much freedom in drawing the system sequence diagram as there is in drawing the domain model. Many mistakes do not lead to a correct diagram of less value, but instead to one that is plainly wrong. Before leaving the SSD, it is therefore wise to make sure that none of the following common mistakes are made.

- Wrong kind of arrow.
- System operation, return value or parameter is missing.
- Operation name does not start with a verb.
- Operation name describes the system's internal design, for example `searchInDatabase` instead of `search`.
- Entities outside the actor are included. A typical version of this mistake would be to include an object `:Customer` in the case study's SSD.
- The object `:System` is split into more objects, showing the system's internal design. As an example, it would be wrong to include an object `:Car`, `:Rental` or `:Balance` in the case study.
- Loops or if-statements are not correctly modeled.
- External systems, like `:Printer` in the case study, are missing.
- Message from `:System` to a user, for example `:Cashier`, instead of reply message, as in figure 4.20. Such a message can be used only if the user does nothing, and the system suddenly, on its own initiative, displays something.
- To draw activation bars is not wrong, but it is discouraged since it tends to confuse, rather than clarify.

NO!



NO!

Figure 4.20 It is wrong to replace a reply message with a message from `:System` to a user, as is done here. The correct way to draw this is shown in figure 4.16.

Chapter 5

Design

The purpose of design is to plan how the code shall be written. The outcome of the design is a plan, giving a clear understanding of the classes and methods the code will contain, and of how these classes and methods will communicate. To write a program without a plan is as inadequate as building a house without a plan. The created plan, that is the design, shall guarantee that the program becomes flexible and easy to understand. Flexible means that it shall be possible to add new functionality without having to change existing code, and to change existing functionality without having to change any code besides that handling the actual functionality being changed. Easy to understand means that developers not involved in creating the program, shall be able to understand and maintain it, without rewriting anything or destroying the program structure.

The plan of the program consists of UML diagrams, therefore this chapter first covers the UML required to create a design. After that, three concepts are covered, that are necessary requirements for a design that is flexible and easy to understand. Next comes an introduction to architecture, or, more specifically, how to organize the program in subsystems. The last thing before doing the design of the `RentCar` case study, is to present a step-by-step method for design.

It is not possible to create a design of a program without understanding how the design can be implemented in code. **Make sure you fully understand sections 1.1 and 1.2 before reading this chapter.**

5.1 UML

This section introduces the UML needed for the design diagrams. Two new diagram types are introduced, *package diagram* and *communication diagram*. Also, more features of class and sequence diagrams, which where introduced in chapter 4.1, are covered.

Class Diagram

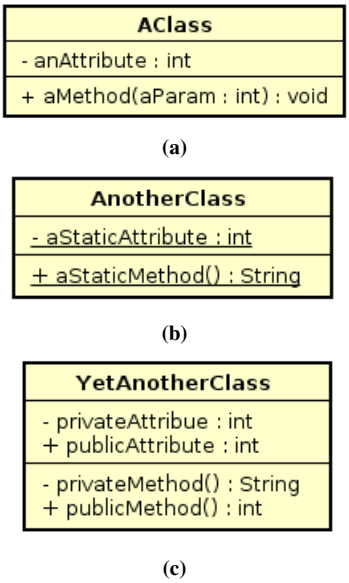


Figure 5.1 Class diagram, illustrating:
(a) method and parameter with types
(b) static members
(c) public and private visibility

A design class diagram contains features that have not been covered previously, namely method, visibility and type. Methods are declared in the lowest compartment of the class symbol. A method parameter's type is drawn after the parameter, separated from the parameter by a colon. The methods return type is drawn the same way, but after the entire method. Also the type of an attribute is drawn the same way. see figure 5.1a. Static methods (and attributes) are underlined, see figure 5.1b. The visibility of a class member (method, attribute or anything else defined in the class) defines from where that member can be accessed. For now, only two kinds of visibility are considered, *public* and *private*. *Any* code has access to a member with public visibility, while only code in the declaring class has access to a member with private visibility. The symbols + and − are used in uml to indicate public and private visibility, respectively, see figure 5.1c.

Package Diagram

The UML symbol *package* means just a grouping of something. In a class diagram of a Java program, the package symbol can mean a Java package. It can also be used to illustrate a larger grouping, like a subsystem consisting of many Java packages. Figure 5.2 shows an example of a package diagram. The dashed line means that something in `somePackage` is dependent on something in `someOtherPackage`. The diagram does not say anything about the extent or type of this dependency.

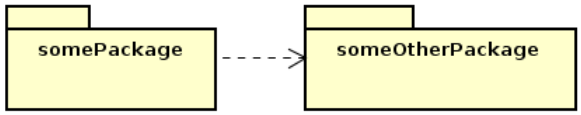


Figure 5.2 A package diagram

Sequence Diagram

This section explains some previously not covered features, needed to create design sequence diagrams. Figure 5.3 illustrates some of these. First, the call to `firstMethod` is a *found message*, which is specific in the sense that the caller is unspecified. This is normally used when the origin of the message is outside the scope of the diagram, and shall not be described in detail. The scope of the diagram is to describe what happens as a consequence of the call to `firstMethod`, not to describe when or why that call is made. Second, parameter types and return types are depicted as in a class diagram, following the parameter or method, separated by a colon.

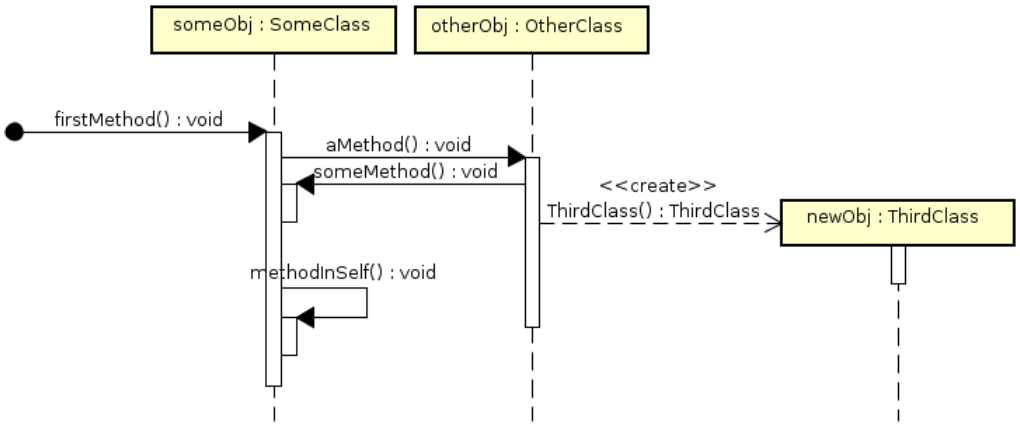


Figure 5.3 Found message, types, activation bars, message to caller, constructor

Third, note the use of activation bars, which show the duration of a method. A bar begins on the first line of a method and ends when returning from the method. There can be any number of overlapping activation bars, since execution might simultaneously be inside any number of methods in the same object. For example, `firstMethod` in the object `someObj` in figure 5.3 calls `aMethod` in `otherObj`, which in its turn calls `someMethod` in `someObj`. Since at this point `firstMethod` has not yet returned, execution is inside both `firstMethod` and `someMethod`, illustrated by the double activation bar of `someObj`.

Fourth, the call to `methodInSelf` illustrates a call where the caller and callee are the same object. Also in this case there is a double activation bar, since execution is inside both `firstMethod` and `methodInSelf`.

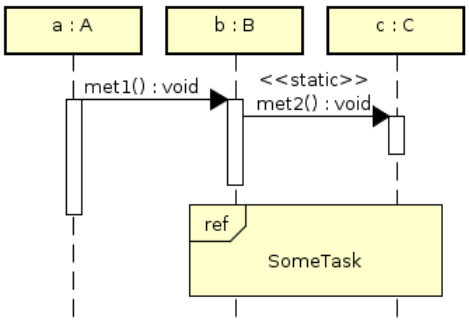


Figure 5.4 Static method and interaction use

Last, a constructor call is illustrated with the creation of `newObj`. This method must be called `ThirdClass`, since a constructor always has the same name as the class in which it is located. Also, the return type must be `ThirdClass`, since the newly created object of that type is returned by the constructor. The text «`create`» above the constructor call is a *stereotype*, which tells that the element with the stereotype belong to a certain category of such elements. Here, it says that the method `ThirdClass` belongs to the `create` category, which means it is a constructor. A stereotype can contain any text, the diagram author is free to invent new stereotypes. However, there are conventions, for example constructors have, by convention, the stereotype «`create`» . It would seem that «`constructor`» would be a more logical stereotype, but unfortunately «`create`» is used instead.

Figure 5.4 illustrates two more features of a sequence diagram. First, `met2` is a static method, which is illustrated with the stereotype «`static`» . Second, the box labeled `ref` is an example of an *interaction use*. It tells there are more method calls where the box is placed, those can be seen in a sequence diagram named `SomeTask`. A diagram should be split like this when it becomes difficult to understand, or fit in a page, because of its size.

Communication Diagram

A *communication diagram* serves exactly the same purpose as a sequence diagram, to illustrate a flow of messages between objects. Both these types of diagrams are *interaction diagrams*. Which type of interaction diagram to use is completely up to the creator, everything relevant for design can be illustrated in both types. The advantage of a sequence diagram is that time is clear, since there is a time axis (downwards), whereas a communication diagram does not have a time axis but illustrates message order by numbering the messages. The advantage of a communication diagram is that objects can be added both horizontally and vertically, whereas a sequence diagram has all objects beside each other and therefore tends to become very wide.

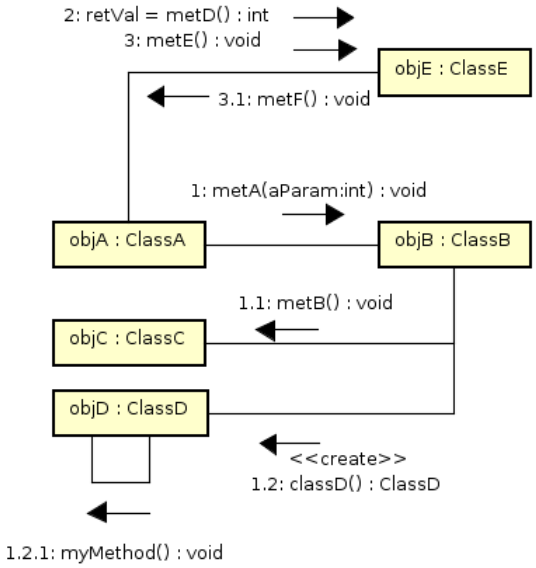


Figure 5.5 A communication diagram

Figure 5.5 shows a communication diagram. A caller and callee are connected by a line, called *link*. A message (method call) is illustrated by an arrow along the link and the name of the message (method), its parameters, types and return value. The first call, `metA`, has index 1. If a called is made from the method `metA`, it has number 1.1, as illustrated by `metB` in figure 5.5. A second call from `metA` has index 1.2 and so on. The order of execution in figure 5.5 is thus 1, 1.1, 1.2, 1.2.1, 2, 3, 3.1. If there is more than one message between the same pair of objects, they are still connected by only one link, see calls 2, 3 and 3.1. Message 1.2 illustrates a constructor call and message 1.2.1 shows a message where caller and callee are the same object.

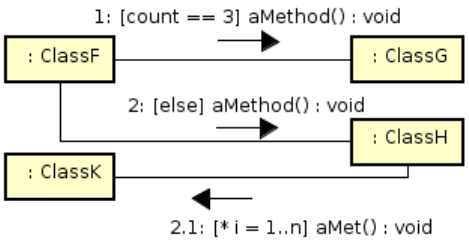


Figure 5.6 Conditional call and iteration in communication diagram

Figure 5.6 illustrates conditional calls (if statements) in messages 1 and 2, and iteration (a loop) in message 2.1. The text in square brackets, called a *guard*, specifies when a particular branch in an if statement is taken, or how many times a loop is iterated. UML does not specify guard syntax, any text or program statement is allowed. If the guard is written without an asterisk, as in messages 1 and 2, it concerns an if statement. If there is an asterisk, as in message 2.1, it is a loop. The asterisk shall be placed before the square bracket, `*[i=1..n]`, but that is unfortunately not possible in astah. Therefore, it is included in the guard statement, which is not correct UML.

5.2 Design Concepts

Much is written and said about software design in general and object-oriented design specifically, and there are many more or less complex solutions to various problems. Still, virtually all design considerations and solutions are based on a few principles. The most fundamental of those, encapsulation, high cohesion and low coupling, are covered here.

Encapsulation

Encapsulation means that irrelevant internal details are hidden. In order to use a certain item, for example a clock, it is not required to understand exactly how it works internally, as in figure 5.7. Instead, it exposes an interface with everything the user has to know. In case of the clock, this is the current time.

Figure 5.7 If you were forced to understand all the internals of a clock to tell the time, that clock would have very bad encapsulation, and would be very difficult to use. Image by FreeImages.com/Colin Adamson

To understand encapsulation in software, the concept *visibility* must be clear. The visibility of a declaration states from where that declaration is visible. For now, it is enough to understand two kinds of visibility, public and private. Public visibility, specified in Java with the modifier `public`, makes the declaration visible to all parts of the program. Any piece of code, anywhere in the entire program, can access something with public visibility, no matter what is declared or where it is declared. Private visibility on the other hand, which is specified in Java with the modifier `private`, means a declaration is visible only to code in the class in which that declaration is placed.

Encapsulation is based on the difference between *public interface* and *implementation*. The public interface is code that is visible to all other code, that is, the declarations with public visibility. The implementation consists of code *not* visible to all other code, that is, method bodies and declarations with private visibility. Something is part of the implementation when access to that something can be controlled, when it is possible to tell exactly which code can access it. Also, note that there is no "neutral" code, all code is either public

```
1  public class TheClass {
2      private int var;
3
4      public TheClass(int var) {
5          this.var = var;
6      }
7
8      public void
9        doSomething(String s) {
10         anotherMethod(s);
11     }
12
13     private void
14       anotherMethod(String s) {
15         //Some code
16     }
17 }
```

Listing 5.1 Public interface in blue italic.

interface or implementation. As a first example, consider listing 5.1, where the public interface is marked with blue italic print. The defining question is *if this code is changed, can code anywhere in the entire program be affected?* If the answer is yes, it is part of the public interface. That is why parameter and return value of the public method is marked in listing 5.1.

Continuing with slightly more complicated examples, consider listing 5.2. The `static` modifier of `PI` is part of the public interface, since `PI` might be used in a statement like `MyClass.PI`. If `static` is removed, that statement will not work. The status of the modifier `final` is quite subtle. If a non-final field is made final, code might definitely break since it will no longer be allowed to write to that field. If a final field becomes non-final, it is not obvious that code will break. However, it would be very surprising if a (previously) final field suddenly changed value. The conclusion of this reasoning is that it is safest to consider the `final` modifier to be part of the public interface.

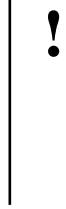
The type and name of `PI` are of course part of the public interface, but why not the value? The answer lies in the promise of this field, which is specified in the comment. Whether the value is `3.14`, `3.1416` or has some other precision, it would still fulfill its contract, to be the constant pi. If the comment had said "The constant pi with two decimals", the value would have been part of the public interface. Is this a bit of hairsplitting? Maybe, but at least it illustrates how one can reason when identifying the public interface.

The private constructor on line seven is not public interface, what matters is that it is private, that it is a constructor is of no importance. Finally, the exception list on line eleven is definitely part of the public interface. If it is changed, exception handling code might break. This holds both for checked and unchecked exceptions.

```
1  public class MyClass {
2      /**
3       * The constant pi.
4       */
5      public static final double PI = 3.14;
6
7      private MyClass() {
8          //Some code.
9      }
10
11     public void aMethod() throws SomeException {
12         //Some code.
13     }
14 }
```

Listing 5.2 Illustration of public interface, which is in blue italic print.

The point in making this distinction between public interface and implementation is that the implementation can be changed anytime, without any risk of unwanted consequences. Changing the public interface, on the other hand, is very dangerous since any code anywhere might break. That might not be a big issue in a small program with only one developer. However, in programs just slightly bigger, with more than one developer, changing the public interface immediately becomes challenging. Those whose code break will not be very happy, especially if it happens regularly or without notice. This is even more disastrous if the code is part of a published API, where it is impossible to know who is using it.



As an example, consider the two methods in listing 5.3. They both have exactly the same public interface, but implementations differ. It would be no problem at all to change between the two implementations, code that calls `multiplyWithTwo` is completely independent of whether the multiplication is done by straightforward multiplication or by shifting. The same way, it is completely safe to change any other part of the implementation, for example the name or parameter types of a private method. It could be argued that it is not allowed to change the implementation of `multiplyWithTwo` at free will, for example not to `return operand * 5`. It is true that such a change can not be made, but that is because it changes the public interface, since both name and comment become erroneous by such a change.

```
1  /**
2   * Doubles the operand and returns the result.
3   */
4  public int multiplyWithTwo(int operand) {
5      return operand * 2;
6  }
7
8  /**
9   * Doubles the operand and returns the result.
10  */
11 public int multiplyWithTwo(int operand) {
12     return operand << 1;
13 }
```

Listing 5.3 Two methods with the same public interface, but different implementations.

In conclusion, it is essential that a public interface is well designed and as small as possible. As soon as a program grows to any reasonable size, it becomes very difficult to change any part of its public interface, to say the least. Many programs suffer from strange constructs originating in public interfaces impossible to change.

High Cohesion

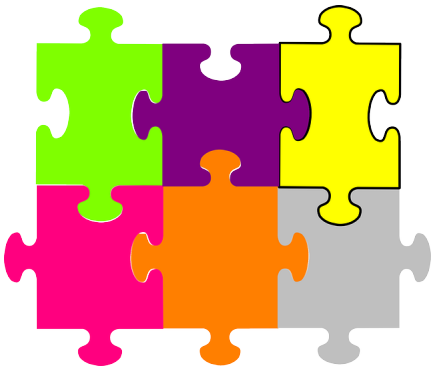


Figure 5.8 When there is high cohesion, the parts of a class fit together and create a whole that is easy to understand. Image by unknown creator [Public domain], via https://pixabay.com

Cohesion is a measurement of how well defined a class' knowledge and its tasks are, and how well they fit together. The goal is that a class shall represent one single abstraction, which is clearly identified by the class name. Furthermore, the class shall have knowledge about that abstraction, not about anything else, and perform tasks related to that abstraction, not to anything else. When this important goal is reached, the class has *high cohesion*.

Figure 5.9 shows two different designs of the same program, one with low cohesion (figure 5.9a) and one with high cohesion (figure 5.9b). In the low cohesion design, the `Employee` class has the method `getAllEmployees`, which returns a list of all employees. This means an `Employee` instance, which represents one single employee, knows all employees in the department. That is not relevant knowledge, instead, that information fits better in a `Department` class, which reasonably shall know all employees working at the department. This latter design, with higher cohesion, is illustrated in figure 5.9b. Also, the `Employee` in figure 5.9a has a method `changeSalaryOfEmployee`, which can change the salary of *any* employee, not just that particular instance. The better design, in figure 5.9b, has another version of this method, which means an instance can change only its own salary. In conclusion, in the better design, an `Employee` instance knows about, and performs operations on, only itself, while in the worse design, an instance knows about, and performs operations on, *any* instance.

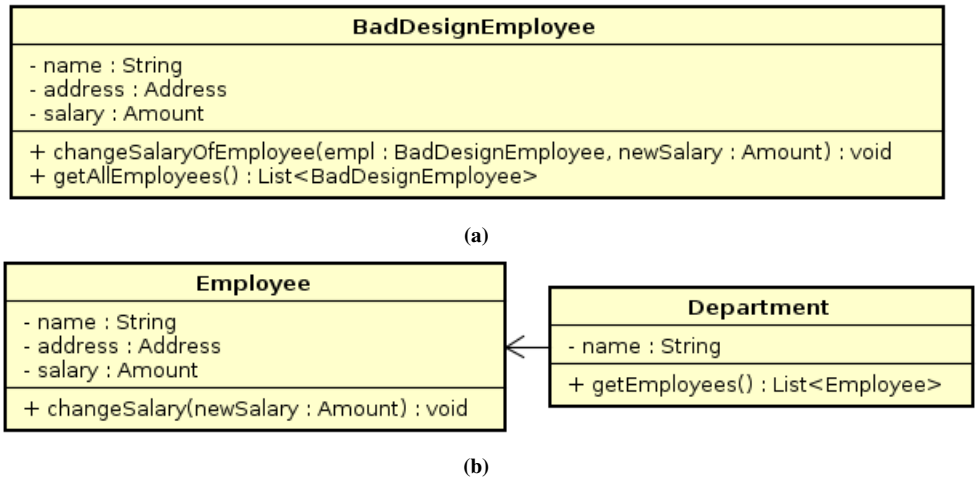


Figure 5.9 Two different designs of the same program: (a) with low cohesion (b) with high cohesion

Another example is given in figure 5.10, which illustrates a `Car` class. In the design with lower cohesion, figure 5.10a, `Car` has methods and attributes which are more related to the abstractions *radio* and *engine*. In the design with higher cohesion, figure 5.10b, those attributes and methods are moved to the new, appropriately named, classes `Radio` and `Engine`. Of course, as is the case with all designs, it can be argued that there are problems also with the designs in figures 5.9b and 5.10b. Still, those two definitely have higher cohesion than their low cohesion counterparts in figures 5.9a and 5.10a.

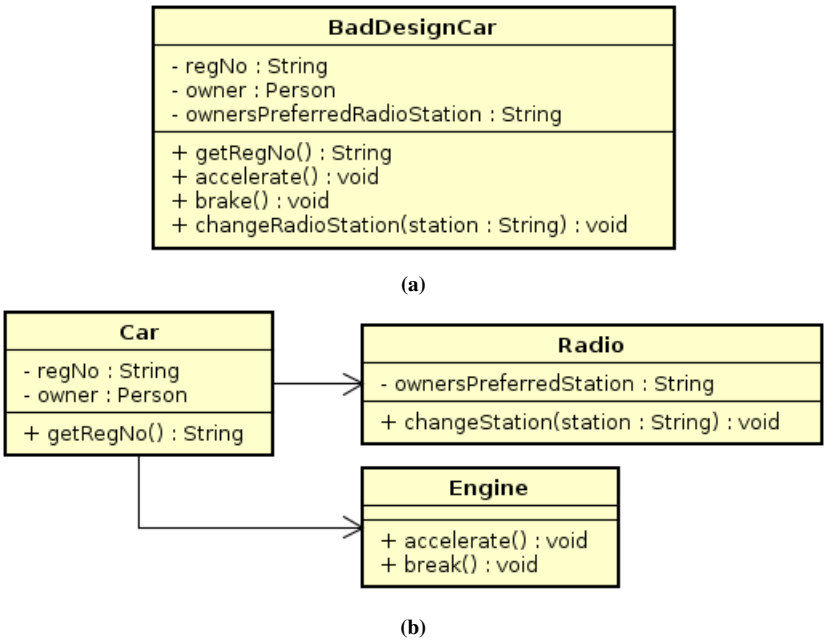


Figure 5.10 Two different designs of the same program:
(a) with low cohesion (b) with high cohesion

It is absolutely mandatory to *always* strive for high cohesion. The programmer can not relax just because the program, at some point in time, has high cohesion. As more code is added, the program will eventually get low cohesion if classes are not split. Therefore, always be on the guard for the possibility to improve the design by introducing new classes, with a clearer responsibility. If the program gets low cohesion, it will be difficult to understand, and also difficult to change, since code that is not really related will be mixed together.

!

Finally, although only classes have been discussed in this section, exactly the same reasoning applies to programming constructs of all other granularities as well. Also subsystems, packages, methods and even fields must be continuously scrutinized regarding cohesion.

Low Coupling

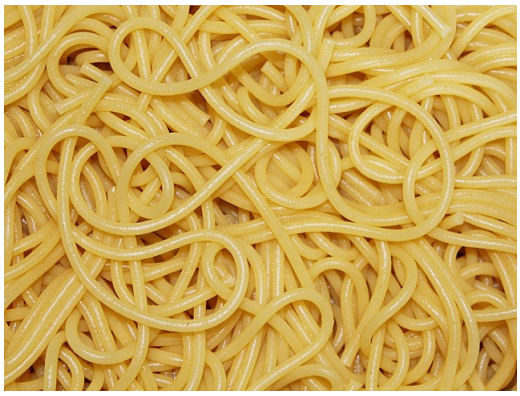


Figure 5.11 If a class diagram looks like a bowl of spaghetti, there is too high coupling. Image by Katrin Baustmann [Public domain], via https://pixabay.com

Coupling defines to which extent a class depends on other classes. What is interesting is primarily on how many other classes it depends, the type of dependency is not of great interest; whether method call, parameter, return value or something else does not matter much. Low coupling means there are as few dependencies as possible in the program. It is not possible to tell a maximum allowed number, what matters is that there are no dependencies which are not required.

The main reason to strive for low coupling is that if a class (depender) depends on another class (dependee), there is a risk that the depender must be changed as a consequence of a change in the dependee. If for example a method name is changed, also all classes calling that method must be changed. The problem is bigger the less control the developer has of the dependee. For example, it is a relatively small problem if the program is small and developed by only one person, but much bigger in a large program where developers far away might change the dependee. The severity of the problem is also defined by the stability of the dependee. The more often a class is changed, the bigger problem to depend on it. For example, it is completely safe to depend on classes in the APIs in the JDK, in the `java.*` packages, since they change extremely seldom.

Figure 5.12 shows two different designs of the same program, one with high coupling and one with low coupling. In the version with unnecessarily high coupling, figure 5.12a, `HighCouplingOrder` has a reference to `HighCouplingShippingAddress`. This is not required since `Order` can get `ShippingAddress` from `Customer`. Therefore, this reference can be omitted, as illustrated in figure 5.12b.

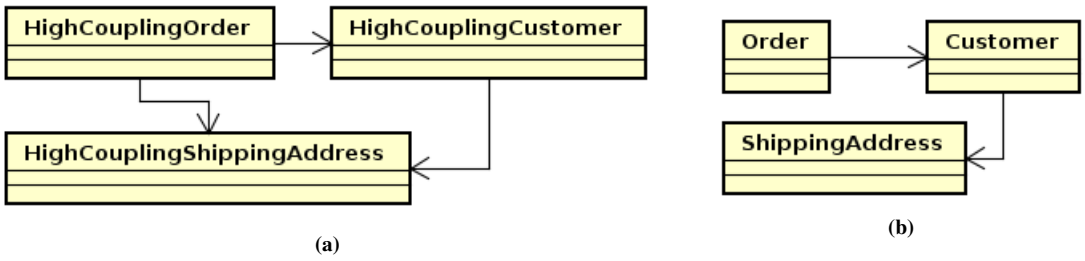


Figure 5.12 Two different designs of the same program: (a) with high coupling (b) with low coupling

Another example of unnecessarily high coupling is found in figure 5.13a, which depicts a typical "spider in the web" design, with a "spider" class that has references to many other, peripheral, classes. The peripheral classes in such a design tends to have none or very few references to other classes. The problem here is that the spider class normally becomes involved

in all operations, thereby getting messy code with bad cohesion. The peripheral classes, on the other hand, tend to become just data containers, doing nothing at all, which makes their purpose unclear. A spider in the web design can normally be improved by moving some of the peripheral classes further away from the spider class, as is done with `Guest` and `Floor` in figure 5.13b. This improved design does not have a spider class with references to all other classes, and there is not a huge set of peripheral classes without references. Note that the total number of references is the same in both designs in figure 5.13. Still, coupling is lowered in the better design, since it does not include a spider class with high coupling. Also, we get more out of the references in the better design, since there it is possible to navigate from `Booking` to `Guest`, and from `Room` to `Floor`.

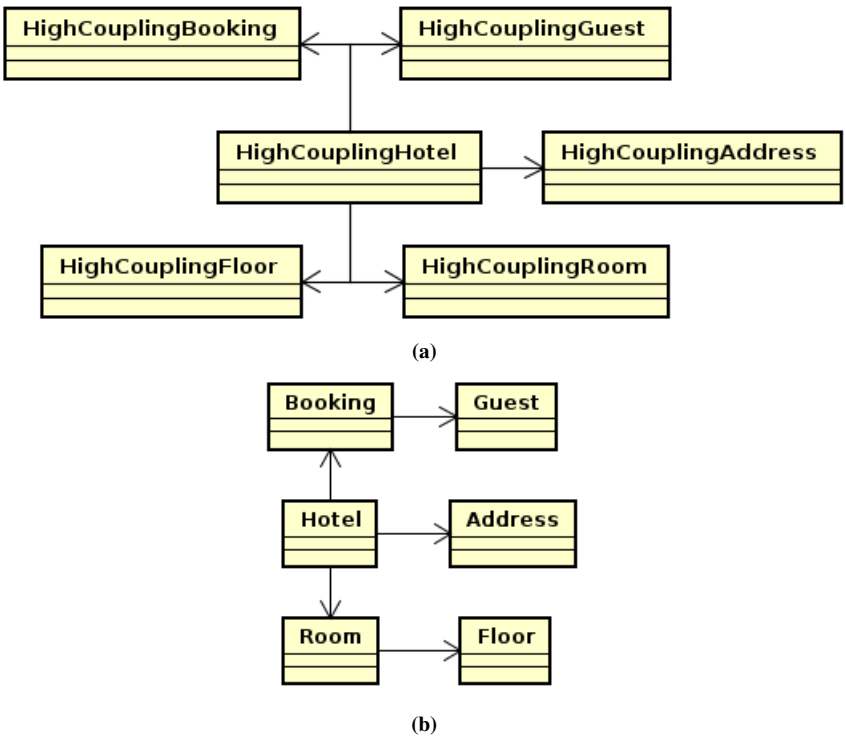
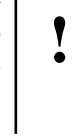


Figure 5.13 Two different designs of the same program:
(a) with high coupling (b) with low coupling

Just as is the case for high cohesion, low coupling is not something that can be achieved once and for all. It is absolutely mandatory to always try to minimize the coupling. Also parallel to high cohesion, low coupling does not apply only to classes, but to programming constructs of all granularity, for example subsystems, packages and methods.



5.3 Architecture

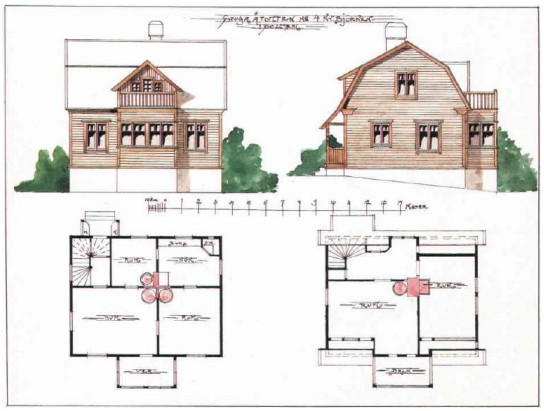


Figure 5.14 An architectural plan does not show any details of the construction. Image by Gunnar Way-Matthiesen (stockholmskällan) [Public domain], via Wikimedia Commons

The architecture gives the big picture of the system under development, it shows how the system is divided into subsystems. It tells which problems the system can solve, and where in the system each problem is solved. It does not, however, tell exactly how the problem is solved, that belongs to design and coding. As an analogy, consider the architectural plan of a building in figure 5.14. It ensures that the problem of moving between floors can be solved, since there is a stair. It does not tell exactly how to construct the stair, which materials to use, etc. And it most certainly is not a real, usable, stair. It is just a plan. Similarly, an architectural plan of a software system could ensure that for example data storage can be handled, by including a database and a class or package that calls the database. However, the architecture of the software system would not be a detailed design of the database or the calling package, and it would definitely not be an actual database or program, but instead a UML diagram or something similar.

Patterns

This section will cover *architectural patterns*, but first, let us make clear what a pattern is. A *pattern* is a common and proven solution to a reoccurring problem. Typically, developers realize that a particular problem in software development is solved many times, in different programs, but the solution is always more or less the same. If this solution works well, it is worth creating a formalized description covering the problem, variants of the solution, advantages and disadvantages of the solution, etc. This formalized description is a pattern. If it concerns architecture, it is an architectural pattern, if it concerns design it is a design pattern, and so on. A collection of patterns is like a cookbook for software development. Knowledge of patterns becomes a common vocabulary for software developers, that can be used to discuss possible ways to solve a particular problem.

Packages and Package Private Visibility

Now that we are about to divide the system into smaller subsystems, it is important to start using packages and package private visibility. This is how logical parts of the program are represented in the Java language, without it, the division into subsystems exists only in the minds of the programmers. Package private visibility means that

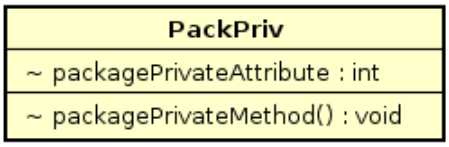


Figure 5.15 Package private visibility.

a particular declaration (field, method, class, etc) is visible only to code in the *same package* as that declaration. In UML, it is illustrated with the tilde character, see figure 5.15. In Java, it is declared by omitting visibility modifier, do not write neither `public`, nor `private` (or anything else). See appendix C.11, showing the implementation of figure 5.15. Note that package private visibility is closely related to private visibility. Both are part of the implementation and impose a strong limit on the visibility. Both make it possible to tell exactly which code can see the declaration.

The MVC (Model-View-Controller) Architectural Pattern

Before this pattern can be considered, it's necessary to understand the term *business logic*, which is a central part of the pattern. Business logic can be explained as being the code implementing business rules, which requires a definition also of the term *business rule*. The business rules define how data is allowed to be shown, deleted, created, or altered, and are rules that would apply even if there were no computers, programs or databases at all. As an example, we can consider rules for withdrawals from a bank account. A perhaps obvious rule is that the amount of money handed to the customer must be subtracted from the account balance. Other rules could be that it's not possible to withdraw more money than there's in the account, that it's not possible to withdraw a negative amount, or that only the account owner can make a withdrawal.

With this understanding of the term business logic, it's possible to look at the architectural pattern *MVC (Model-View-Controller)*. It tells that the system must be divided into the subsystems *Model*, *View* and *Controller*, to avoid mixing code doing completely different things. Without such a division into subsystems, it would easily happen that user interface code and business logic code were mixed in the same method. Say that we are coding, for example, a bank account. A straightforward solution is to have a class `Account` that has a field `balance` and methods `deposit` and `withdraw`. That is fine, such a class contains only business logic and the current state (the variable values, for example the account balance). However, we also want to present the state of the account, for example its balance, to the user. Therefore, it might seem adequate to add code handling for example a HTML user interface to the `Account` class. This, however, would be a disaster! HTML user interfaces and business rules for withdrawing money are two completely different things. Mixing them just because they both use the same data, namely the account balance, would lead to extremely low cohesion and high coupling. Low cohesion because the very same method would handle such different things as UI and business logic, high coupling because UI code and business logic code would be inseparable, placed in the same method. As a consequence, a HTML designer would have to know Java and understand the business rules, and a Java developer would have to understand the web based user interface created with HTML and CSS. Furthermore, it

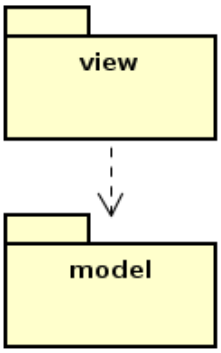


Figure 5.16 The `view` and `model` packages.

would be impossible to reuse the HTML for other web pages, not to mention the nightmare of changing to another user interface, or having multiple user interfaces to the same program. Maybe a customer using the internet bank needs a web based UI and a bank clerk needs a UI of a Java program run locally. To avoid such a disastrous mess, the MVC pattern tells us to create the subsystems `view`, containing all code managing the user interface, and `model`, with the business logic code, see figure 5.16.

Having separated the system into `view` and `model`, the two separate tasks *user interface* and *business logic* are clearly separated into two subsystems, each with high cohesion. There is, however, still a remaining problem. To understand it, let us first consider an analogy, namely to build a new school. The school will be used by a large number of people in many different roles, for example students, teachers, headmasters, IT staff. All these may have ideas about the construction that they want to communicate to the construction workers. There is also a large number of construction workers in many roles, for example carpenters, electricians, plumbers. This is illustrated in figure 5.17, which depicts two different organizations. The upper organization, figure 5.17a, is quite chaotic since anyone with an opinion about the construction is allowed to give input to any construction worker. It is easy to understand that no usable school will ever be built with such an organization. The lower image, figure 5.17b, on the other hand, illustrates a better organization. Here, there is a steering committee (green person) organizing the input. No-one talks directly with a construction worker, instead all input goes to the steering committee person, who filters the input and decides what to forward and to which worker.

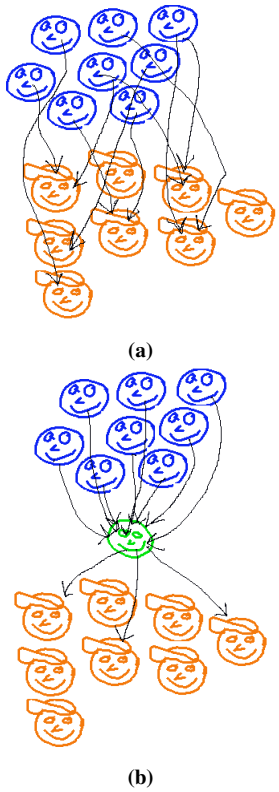


Figure 5.17 People involved in constructing a new school. The orange persons symbolize the construction workers (carpenters, electricians, plumbers, etc). The blue persons symbolize people that will use the school (teachers, students, administrative staff, etc), and therefore give directives to the construction workers.
(a) Chaotic organization without steering committee.
(b) Well-functioning organization with steering committee (green person).

The analogy to the MVC pattern is that the blue persons represent classes in the `view` package, since they give input, and the orange persons represent classes in the model, since they perform the desired work. The architecture depicted in figure 5.16 would lead to an organization of the software similar to figure 5.17a. Any object in the view would call methods in any object in the model. Such a system would have very high coupling. A change to a class in the model could affect any class in the view. Also, to change or update the view would be very difficult since it would not be clear how replacing or changing a class in the view would affect what work is actually performed in the model. The solution is to introduce a class (or some classes) corresponding to the steering committee. Such a class is called a *controller* and is placed in the third layer of the MVC pattern, the `controller`

layer. Figure 5.18 shows all three MVC layers and the `Controller` class. The controller shall contain the system operations, that is, the operations of the `System` class in the system sequence diagram, which was made during analysis. A user action, for example to click a button in the user interface, will result in one call from an object in the view to a system operation in the controller. That operation in the controller shall call the correct methods in the correct objects in the model, in the correct order. This way, the work is done in the model and it is the controller's responsibility to know which object in the model does what. The view will not have any knowledge about the model or dependence on it.

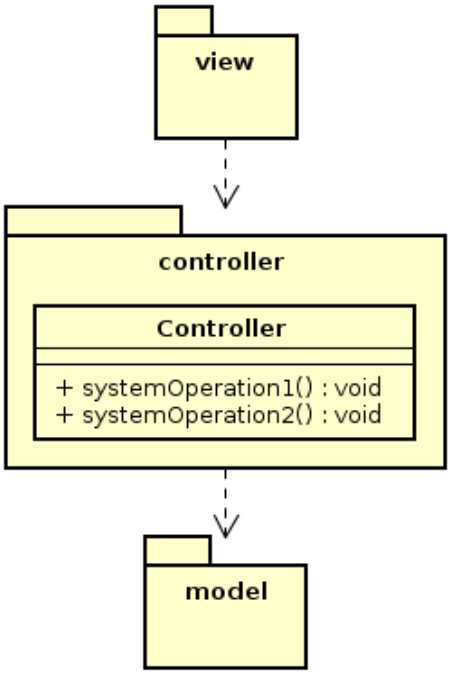


Figure 5.18 The MVC layers and the `Controller` class.

To summarize, the MVC pattern tells us to divide the system into three subsystems. The first is view, which is responsible for presenting the user interface and for interpreting the user's input. There must not be any code related to any kind of user interface outside the view. The second subsystem is controller. As stated above, the controller's responsibility is to call the correct methods in the correct objects in the model, in the correct order. The last subsystem is model, which contains the program's representation of real world entities and is responsible for the actual functionality of the system, the business logic. Figure 5.19 is a sequence diagram showing the flow from `view`, via `controller`, to `model`. The advantages of the MVC pattern are that each subsystem has high cohesion, and that there is low coupling between user interface code and business logic code since they are separated in different subsystems. The view and the model can now be developed separately, by different teams. It is in fact possible to completely replace the view, or to have multiple views simultaneously, without affecting the model in any way.

Before leaving the MVC pattern, it is worth considering interaction between the three subsystems a bit more. Regarding view and controller, a remaining question is who handles flow control between views. Suppose for example the user interface shows a list with summary information about different items. If the user clicks an item, a new view shall be displayed, with detailed information about that item. Which object knows that the list view shall be replaced by the detailed view? Flow control is the responsibility of the controller, not the view, but the controller does not know which view is currently displayed. The answer is that managing flow control between views is a task complex enough to give low cohesion to whatever class it is placed in. It is best to introduce a new class, which has exactly this responsibility. This class could be placed in either the `view` layer or the `controller` layer.

Regarding communication between view and model, it is best that, as in figures 5.18 and 5.19, there is no such communication at all. That makes these two layers completely independent, reducing coupling to a minimum. If so, the only way to send data from model to view, to

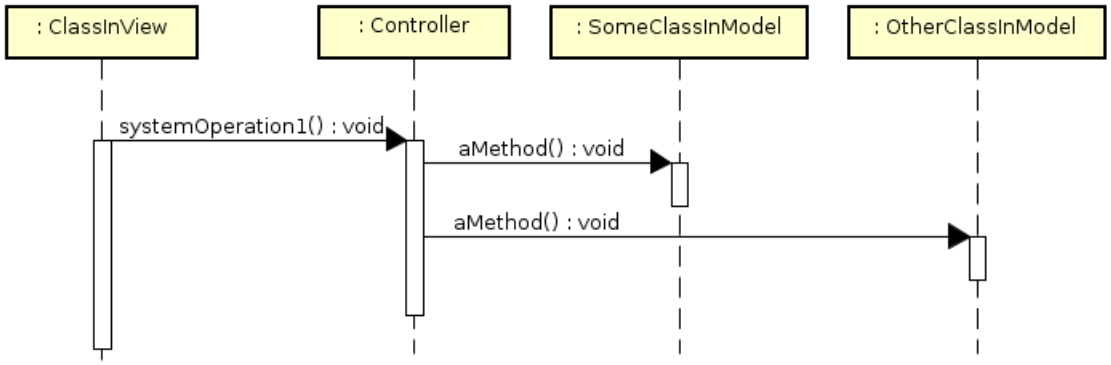


Figure 5.19 Method calls from `view`, via `controller`, to `model`.

be displayed to the user, is as return values to method calls from view, via controller, to model. If that is possible, everything is fine. Unfortunately, it is often not possible, since one method call might require many different return values. It might also be that a view shall be updated when no call to the model has been made, for example as a result of the model being updated by a call from another program, or because of a timer updating the model regularly. An option could be to add lots of getter methods to the model, and let the view use those to retrieve the required data. This solution has some big disadvantages, for example that corresponding getter methods must also be added to the controller, which will make it terribly bloated and messy. Also, the view can not know exactly when to call those getters, since it can not know when the model changes state, if the state change is not initiated by the view itself. There is an elegant solution to this problem, that will be covered later. For now, all considered scenarios will allow data to be passed from model to view as return values to method calls via controller.

The Layer Architectural Pattern

The *Layer* architectural pattern is more general than the MVC pattern. While MVC concerns the model, view and controller layers in particular, the layer pattern just says that the system shall be divided in layers. MVC solves the problem that user interface and business logic risk to be mixed. Layer applies the same reasoning, but to any two different kinds of code. Just as mixing user interface and business logic brings low cohesion and high coupling, so does mixing for example business logic and database calls. Calling a database is a separate task, in no way related to the business logic in the model. This means there shall be a separate layer dedicated to database calls. Continuing this reasoning, it is important to always be prepared to add a new layer. That must be done whenever writing code that will give low cohesion to whatever existing layer it is placed in. As an example, consider the `main` method. Its task is to start the program, which is not related to any layer mentioned so far. Therefore, yet a new layer must be introduced, whose responsibility is to start the application.

Exactly which layers there shall be in a system is a matter of discussion, and it also differs from system to system. However, all layers that have been mentioned here are often present. Those layers, depicted in figure 5.20, are `view`, `controller`, `model`, `dbhandler`

(sometimes called `integration`, responsible for calling the database), `data` (the actual database) and `startup`, which includes the `main` method and all other code required to start the application. Always strive to keep dependencies in the direction illustrated in figure 5.20, from higher (closer to the user) layers to lower (further from the user) layers. Such dependencies are unavoidable, since execution is initiated by the user. Dependencies in the opposite direction are however not needed, there is nothing forcing lower layers to call higher layers. Since such dependencies are unnecessary, introducing them means unnecessarily high coupling. Also, higher layers tend to be less stable than lower layers. For example, it is more common to change user interface layout than to change business rules, and yet less common to change entities represented in the database. Furthermore, it would be very counter-intuitive if, for example, a call from `controller` to `model`, in order to carry out some system operation, resulted in the model calling back to the controller, starting *another* system operation.

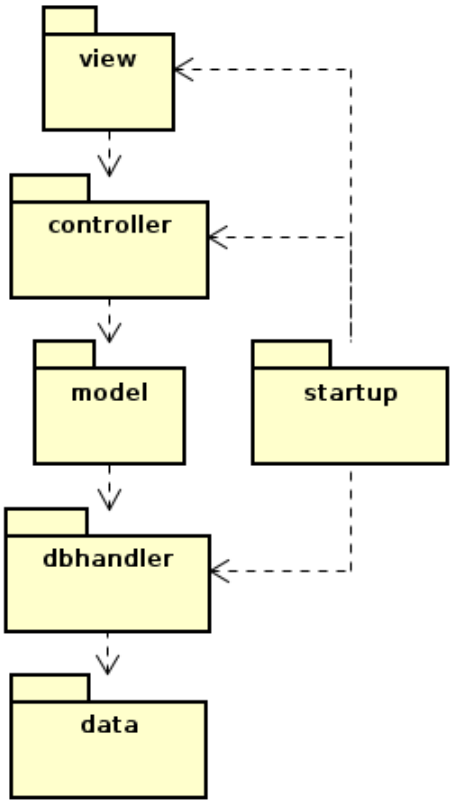


Figure 5.20 Often used layers.

Note that some layers have a particularly close relation. The `controller` layer exists exclusively because its task is to know how to call the model, and `dbhandler` exists exclusively because it encapsulates all database calls. As a consequence, `model` shall only be called by `controller` and never by any other layer, and `data` shall only be called by `dbhandler` and never by any other layer. Apart from this, layers may be bypassed. It is for example perfectly fine to call `dbhandler` directly from `controller`, instead of going via `model`.

!

To conclude, the layer pattern has important advantages. If layers are correctly designed, they form subsystems with high cohesion and low coupling. Also encapsulation applies to layers, the public interface of a layer shall be as small as possible, not revealing more than required of the layers internal workings. When encapsulation, cohesion and coupling are used to make layers independent, it becomes easy to maintain the layers and to divide development of different layers between developers. It is also easy to reuse code, since a layer can provide a well-designed public interface, callable from any code in a higher layer.

The DTO (Data Transfer Object) Design Pattern

As the number of layers increase, so does the need to pass data between layers. This often leads to long parameter lists in many methods as data is passed through the layers. Consider for example registering a new user in some community. Say that registration means to enter name, street address, zip code, city, country, phone number and email address. These are

seven string parameters that shall be passed though all layers from user interface to database, which means there will be (at least) method declarations similar to those in listing 5.4.

```
1  //In the controller layer
2  public void registerUser(String name, String streetAddress,
3                           String zipCode, String city, String country,
4                           String phone, String email) {
5  }
6
7  //In the model layer
8  public void registerUser(String name, String streetAddress,
9                           String zipCode, String city, String country,
10                          String phone, String email) {
11 }
12
13 //In the dbhandler layer
14 public void createUser(String name, String streetAddress,
15                          String zipCode, String city, String country,
16                          String phone, String email) {
17 }
```

Listing 5.4 The same method signature often appears in many different layers. This is problematic if the method has a long parameter list.

Just to make many method calls is not a problem, but the long parameter list is. First, it is difficult to remember the meaning of each parameter, especially when they all have the same type, as is the case here. Second, a long parameter list means a large public interface, and thereby a big risk that it is changed. An often used method to get rid of the parameter list is to use a data transfer object, DTO. Such an object is a just data container, without any logic. Its only purpose is to group data in the same class, see listing 5.5.

```
1
2  //The DTO
3  public class UserDTO {
4    private String name;
5    private String streetAddress;
6    private String zipCode;
7    private String city;
8    private String country;
9    private String phone;
10   private String email;
11
12   public UserDTO(String name, String streetAddress, String zipCode,
13                  String city, String country, String phone,
14                  String email) {
15     this.name = name;
16     this.streetAddress = streetAddress;
```

```
17      ...
18    }
19
20    public String getName() {
21      return name;
22    }
23
24    public String getStreetAddress() {
25      return streetAddress;
26    }
27
28    ...
29  }
30
31  //In the controller layer
32  public void registerUser(UserDTO user) {
33
34  }
35
36  //In the model layer
37  public void registerUser(UserDTO user) {
38
39  }
40
41  //In the dbhandler layer
42  public void createUser(UserDTO user) {
43
44  }
```

Listing 5.5 Here, the problematic parameter list of listing 5.4 has been removed by introducing a DTO

An obvious objection is that the long parameter list is not gone, it is just moved to the constructor of the DTO, `UserDTO`, on lines 12-14 in listing 5.5. However, it now appears only in one place. If it is changed, only one public interface is changed, not one in each layer. Also, it is now obvious that all user related data belongs together.

It is sometimes problematic to tell whether a certain class is a DTO or an actual model object, an *entity*. There was no such ambiguity in the example above, where `UserDTO` was created for the sole purpose of passing data between layers, and *DTO* was included in the class name to emphasize this. It would be more problematic if there already existed a `User` class in the model, and that class looked exactly like the `UserDTO` above. Could this `User` class be considered a DTO, or would we still have to create a `UserDTO`? There are different ways to answer that question. One way to distinguish between an entity and a DTO is that two DTO instances are equal if all their attributes are equal, but two entity instances are equal if some kind of instance id is equal, for example a bank account number. Another way to decide the difference is that an entity can change state. It has set methods and maybe also

other, more complex, business logic methods that updates its state, for example a method to withdraw money from a bank account. A DTO, on the other hand, is read-only, it has only get methods. This means it is *immutable*, none of its fields can ever change value. Also, since it can't change state, it has no history. It's just a snapshot of what something looked like at a particular instance in time. Yet another, very strict, way to make the distinction is that a class is a DTO only if it exists just for passing data between layers, and if its name ends with DTO. This was the case in the example above.

5.4 A Design Method

Finally, all the theoretical background is covered. Having sufficient knowledge about UML class, sequence, and communication diagrams; the design concepts encapsulation, cohesion and coupling; and the architectural patterns MVC and layer, it is now time to look at how to actually design a program. This section describes a step-by-step method for design, which will be used to design the `RentCar` case study in the next section.

1. **Use the patterns MVC and layer.** This means to create one package for each layer that is supposed to be needed. Exactly which layers that is, is a matter of discussion, and can not be known for certain until the design is complete. An educated guess that is valid for many programs is to use the layers depicted in figure 5.20. Having decided which layers to create, draw a class diagram with one package for each layer, and also draw the `Controller` class in the `controller` layer.

2. **Design one system operation at a time.** The system sequence diagram shall guide our design, it shows exactly which input and output the program shall have. Also design enough of the system's start sequence (initiated by the `main` method) to be able to test run the newly designed system operation. When creating the design, **use interaction diagrams**. An interaction diagram shows the flow through the program, how methods call each other. **Do not use a class diagram**, which has no notion of time or execution flow. Whether to use sequence or communication diagrams is a matter of taste.

3. **Strive for high cohesion, low coupling, and a high degree of encapsulation with a small, well-defined public interface.** When adding new functionality, create the required methods in a way that these goals are met to the highest reasonable degree. This is much easier said than done, and often requires much thought and discussion. It helps to remember that an operation shall be placed in a class representing the abstraction to which the operation is associated, and that has the data required for the operation. The domain model helps to find new classes that can be introduced. Also, always be prepared to change previously designed system operations, to improve the overall design.

   Now that the domain model is mentioned, it is appropriate to warn of changing it. Changing the domain model is allowed, but should not be taken lightly. The DM represents an agreement between all stakeholders, on the reality in which the program exists. If the DM is changed, all involved parties must agree on that change.

When designing, *favor objects over primitive data and avoid static members*, since neither primitive data nor static members are object oriented. When using these, the entire *object* concept is completely ignored, and the prime tool (objects) to handle encapsulation, cohesion and coupling is thrown away.

4. **Maintain a class diagram.** When done designing a system operation, summarize the design in the class diagram created in bullet 1, in order to give an overview of the entire program. The diagram shall show packages, classes, methods and associations. When an object in an interaction diagram calls a method in another object, the caller's class will have an association to the callee's class. Also attributes can be added, if any are known. Such a diagram tends to become very big and messy, it is permitted to omit parts to make the diagram clearer. If that is done, it should be clearly specified.

5. **Implement the new design in code.** Design is not an up front activity that can be done once and for all for the entire program. Instead, it shall be done in iterations, as soon as a design is ready it shall be implemented in code, and thus evaluated. **Here, this step is postponed until the next chapter.** The reason is that seminar two would otherwise be far too big. However, when designing, it is important to have an understanding of how the design can be implemented in code.

6. Start over from bullet 2 and design the next system operation.

5.5 Designing the `RentCar` Case Study

As an example, the `RentCar` case study is designed in this section. For convenience, the specification, SSD and domain model are repeated below, in figures 5.21, 5.22 and 5.23. When designing, be sure to have an understanding of how the design may be implemented in code. If that is not clear, now is the time to repeat Java programming, for example by reading chapter 1, especially sections 1.1 and 1.2.

Step 1, Use the Patterns MVC and Layer

Following the method described in section 5.4, the first thing to do is to create a class diagram with the anticipated layers. There is no reason to deviate from the typical architecture of figure 5.20. Therefore, after having introduced the `Controller` class, the design looks as in figure 5.24.

Step 2, Design One System Operation at a Time

The view is not designed here, instead the `view` package contains a single class, `View`, which is a placeholder for a real view, that certainly would consist of more classes. This way, there is no need to bother about view technologies like console IO or HTML. Also, there is nothing conceptually different in designing the view than there is in designing any other layer; exactly the same reasoning as here is followed when the view is designed.

1. The customer arrives and asks to rent a car.
2. The customer describes the desired car.
3. The cashier registers the customer's wishes.
4. The program tells that such a car is available.
5. The cashier describes the car to the customer.
6. The customer agrees to rent the described car.
7. The cashier asks the customer for name and address, and also for the driving license.
8. The cashier registers the customer's name, address and driving license number.
9. The cashier books the car.
10. The program registers that the car is rented by the customer.
11. The customer pays, using cash.
12. The cashier registers the amount payed by the customer.
13. The program prints a receipt and tells how much change the customer shall have.
14. The program updates the balance.
15. The customer receives receipt, change and car keys.
16. The customer leaves.

4a. The program tells that there is no such car available.

1. The cashier tells the customer that there is no matching car.
2. The customer specifies new wishes.
3. Execution continues from bullet three in basic flow.

Figure 5.21 The `RentCar` scenario.

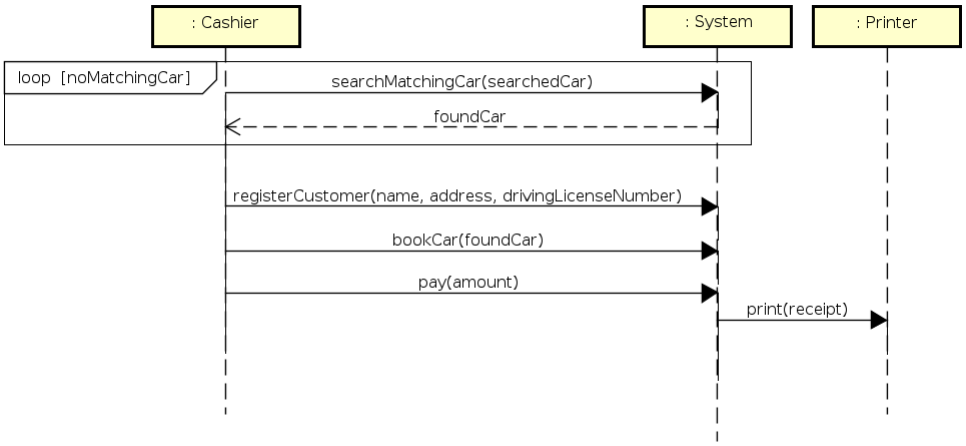


Figure 5.22 The `RentCar` system sequence diagram.

The system operations are designed in the order they are executed according to the SSD, figure 5.22. The first operation in the SSD is `searchMatchingCar`, which takes the parameter `searchedCar` and returns the value `foundCar`. The first step is to create an interaction diagram. Since the MVC pattern says the controller shall contain the system operations, the

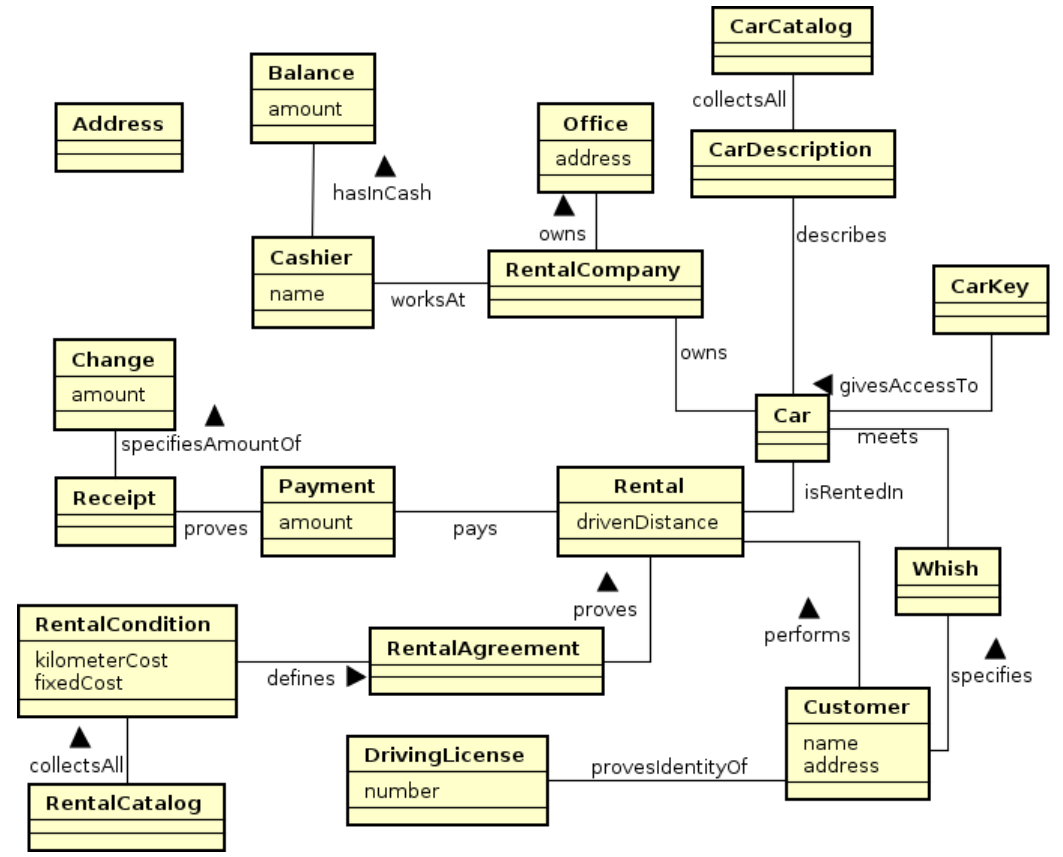


Figure 5.23 The `RentCar` domain model.

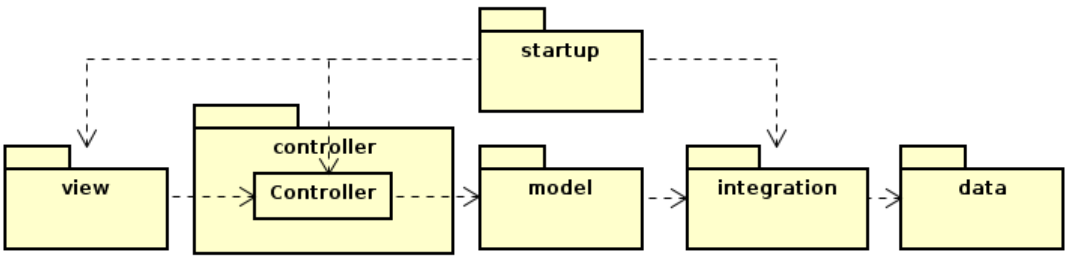


Figure 5.24 The first version of the `RentCar` design class diagram.

method `searchMatchingCar` can be added to the controller right away. Here, however, comes the first design decision. Which is the type of the parameter `searchedCar` and the return value `foundCar`?

Step 3, Strive for encapsulation with a small, well-defined public interface, high cohesion and low coupling

The question, *which is the type of the parameter* `searchedCar` *and the return value* `foundCar`, is the first of a large number of design decisions, let's consider it carefully. The answer shall be guided by the concepts encapsulation, cohesion and coupling. The purpose of `searchedCar` is to represent the customer's requirements on the car to rent, and the purpose of `foundCar` is to describe the available car that best matches those requirements. The design currently contains only two classes, `View` and `Controller`. Obviously, it would be lousy cohesion to let any of those represent the customer's requirements or the matching car. The features of a car is not just one value, but a quite large set. The requirements specification, figure 5.21, does not tell exactly which features the customer can specify. This is definitely something we would have to ask about if the program should be used for real. In this exercise, we have to decide on our own, let's say that the customer can wish for price, size, air condition, four wheel drive and/or color. These values clearly belong together, as they describe the same abstraction, a car. Therefore, they should be fields in the same class, which must be introduced to the design. Can the domain model, figure 5.23, give any inspiration about this new class? Yes, it shows the class `Car`, which seems to be an important abstraction since it has many associations. The `Car` class can also be used for the return value, `foundCar`. This object, however, can represent a specific car, not just any car matching the specified criteria. Therefore, the registration number must also be a field in this class. Now, having decided on the representation of `searchedCar` and `foundCar`, it is possible to create the first version of the `searchMatchingCar` interaction diagram, figure 5.25.

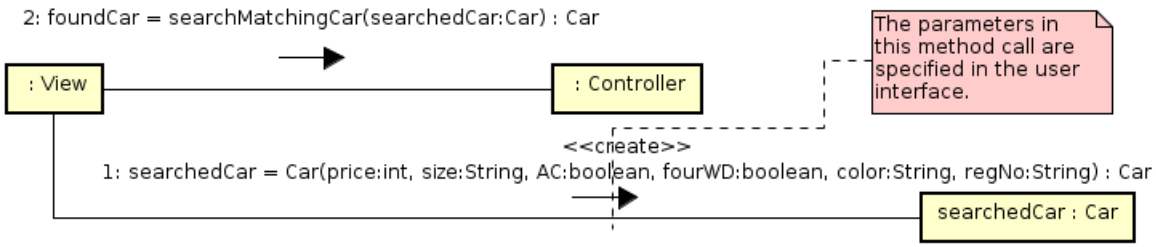
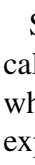


Figure 5.25 The first version of the `searchMatchingCar` design interaction diagram.

There are a few things worth noting in this diagram. First, the name `searchedCar` appears in three different places, the object name, method call one and method call two. The fact that the same name is used implies that it is in fact the very same object in all three places, which is important information for the reader.

!

Second, data can not appear out of nothing, since `searchedCar` is a parameter in method call two, it must be clear from where this object comes. This is illustrated in method call one, where it is created. But what is the origin of the parameters in method call one? That is explained in the note, they are entered by the user in the user interface.



Last, the diagram does not tell in which layer the `Car` class is located. It is quite OK not to show layers in the interaction diagram, but we still must decide the location of `Car`. In fact, all layers are candidates, since it already appears in `view` and `controller`, and we can guess that it will be passed through `model` to `dbhandler`, since a search in the database for a matching car is probably required. A rule of thumb is to place a class in the lowest layer where it is used, in order to avoid dependencies from lower to higher layers. This would indicate that `Car` should be placed in `dbhandler`. However, there are other questions as well, is `Car` a DTO or is it the actual model object, the entity, with the business logic? Also, does the entity (in the model) contain any business logic at all, or has it only got getter methods? In the latter case, is there any need to introduce both a DTO and an entity? Since they would be identical, the `Car` class in the model could be considered to be a DTO instead. These questions can not be answered until more of the system is designed. For now, we just choose the simplest solution, namely to let `Car` be a DTO, place it in `dbhandler`, and not add an entity. This decision might have to be changed later. Note that if we had decided to turn `Car` into an entity object, it could no longer have been created by the view, since model objects shall only be called by the controller. Let's change the name to `CarDTO` to make clear that it is a DTO, this makes the communication diagram look like figure 5.26.

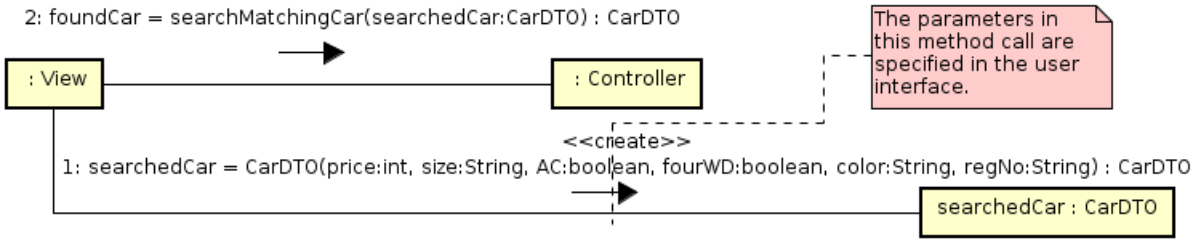


Figure 5.26 `Car` is renamed to `CarDTO`.

Next, it is time to decide which object is called by `Controller`. Shall some object in the model be created or shall the controller just make a search in the database? The answer is that a model object shall be created if it is of any use after this call. For example if it later shall be stored in the database or if it will be used in a future system operation. As far as we know now, none of these cases apply, there is no future use for a model object representing the search. Therefore, `Controller` will just call an object responsible for searching in the database. This object, let's call it `CarRegistry`, will reside in the `dbhandler` layer, since the purpose of that layer is exactly this, to call the database. The database itself, represented by the `data` layer, would normally be another system, called by `CarRegistry`. Here instead, since there is no database, `CarRegistry` will just look in an array of available cars. The final design of `searchMatchingCar` is in figure 5.27. Note that there is no notion of the loop that exists in the system sequence diagram, since the loop condition `noMatchingCar` is not a part

of the program, but is completely decided by the cashier. Exactly the same flow, depicted in figure 5.27, is executed again and again, until the customer is satisfied.

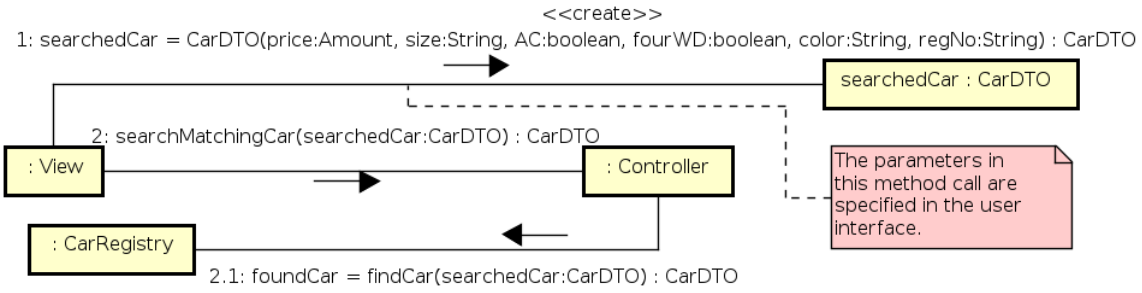


Figure 5.27 Call to `dbhandler` layer is added.

The next task is to design as much of the start sequence, in the `main` method, as is needed to run the newly designed system operation. That start sequence must create all objects that are not created during execution of the system operation itself. There exists currently four objects in total, `searchedCar` and nameless objects of the classes `View`, `Controller` and `CarRegistry`. Of these, only `searchedCar` is created in the design of the system operation, the other three must be created by `main`. But not only must they be created, they must also be given references to each other to be able to call each other. In particular, `View` calls `Controller` and must therefore have a reference to `Controller`. Also, `Controller` calls `CarRegistry` and must thus have a reference to `CarRegistry`. One option is that `main` creates all three objects and passes references as needed. Another option is that main creates fewer objects, for example only `View`, which in turn creates `Controller`, which finally creates `CarRegistry`. Both options have their pros and cons. The latter solution could be problematic if in the future there is the need to create for example a `Controller` without a `CarRegistry`. This might also indicate that the controller gets low cohesion if it creates `CarRegistry`. On the other hand, this solution has less coupling since `main` will only have a reference to `View`, not to `Controller` or `CarRegistry`. It is very hard to tell now which of the solutions that is best for this particular program. Let's not get stuck, but just choose the former alternative, and let `main` create all three objects, see figure 5.28. This decision was taken a bit by chance and gut feeling, which is sometimes needed. It is important to carefully consider alternatives, but it is also important not to get stuck unnecessarily. A parameter to consider is how easy it is to change the decision later on. The easier it is to change, to less time can be spent when taking the decision.

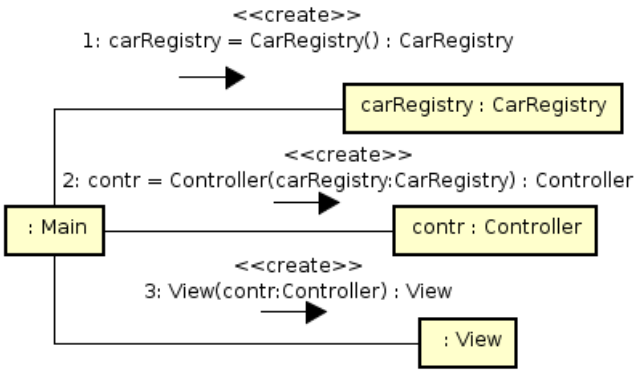


Figure 5.28 The start sequence in the `main` method.

Step 4, Maintain a Class Diagram With All Classes

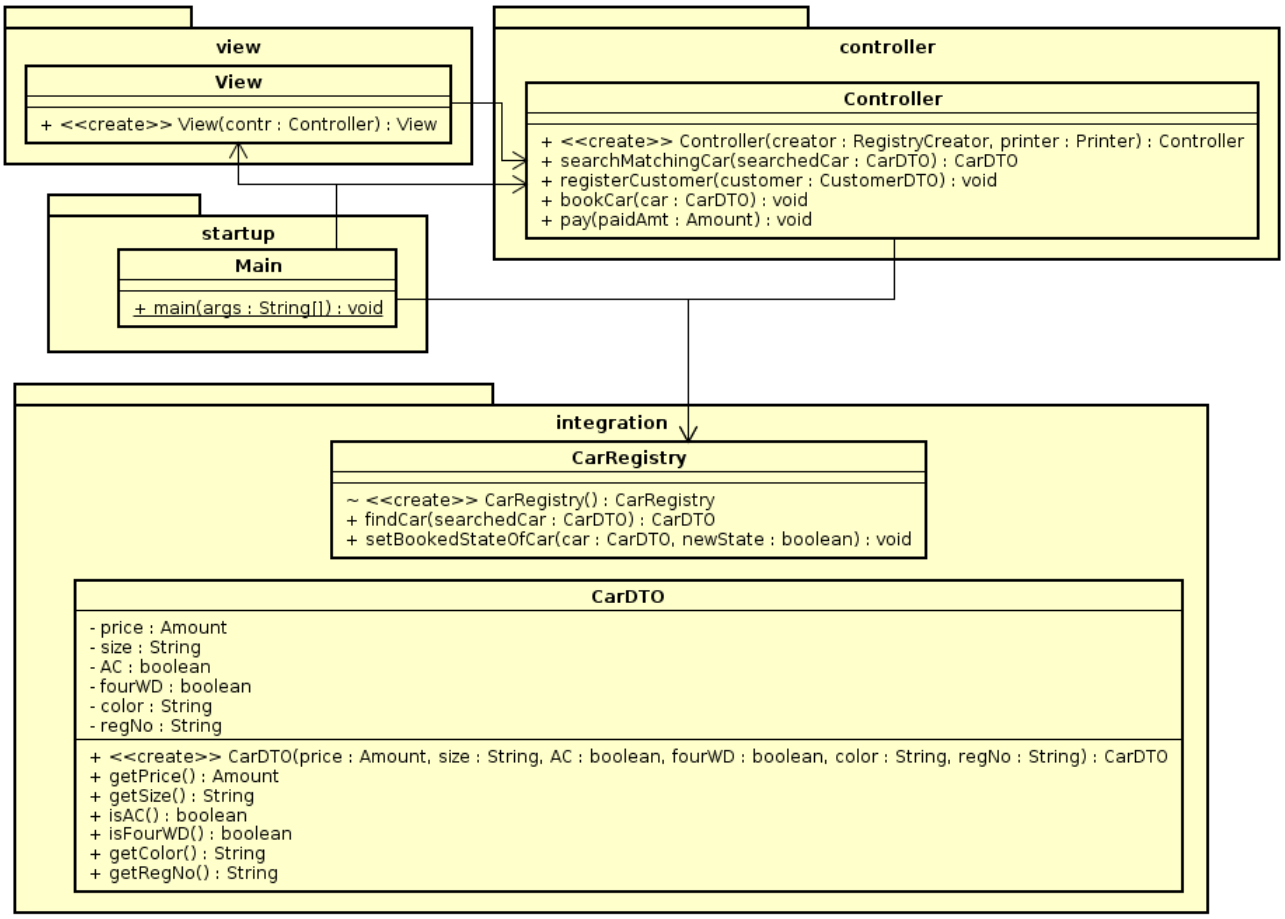


Figure 5.29 `RentCar` design class diagram after designing `searchMatchingCar`.

The last task in the design of `searchMatchingCar` is to summarize what has been done in a class diagram, see figure 5.29. Note that it is not mandatory to include all attributes, methods and references if they do not add any important information, but only obscure the diagram. For example, it is common not to include references to DTOs, since they are used in many different layers and are considered as data types. Not including references to DTOs is similar to not including references to `java.lang.String`, which can also be considered a data type.

Before leaving `searchMatchingCar`, we evaluate it according to the criteria encapsulation, cohesion and coupling. To start with encapsulation, all methods are public. This is not exactly an ideal situation, but often quite unavoidable early in the design. We are creating the different layers and tying them together. In fact, all methods are called across layer borders, and it has to be that way. Otherwise, it would not be possible to communicate between layers, since there are still very few methods. All fields, on the other hand, are private, which is very good. Looking at cohesion, we can safely say that all classes do what they are meant for, nothing more. `Main` starts the program, `View` just calls the controller, `Controller` has

a system operation that calls a search method in the `CarRegistry`. `CarRegistry` has only this search method and `CarDTO`, finally, has no methods at all. Regarding coupling, there is a chain of dependencies from higher to lower layers, that is from `view` to `controller` to `dbhandler`, which is exactly the purpose of the layer pattern. Also, it is perfectly in order to have dependencies from `Main` to the other layers, since the task of `Main` is to start the other layers. It would most likely not be appropriate if `Main` had references to many classes in the same layer, that would probably be too high coupling. For the moment, however, `Main` references only one class in each layer.

That concludes the design of the `searchMatchingCar` system operation. The question naturally arises, is all this designing really necessary just to fetch an element from an array? The answer is *yes*, most definitely yes. A professional programmer should, and normally does, make this kind of considerations all the time. However, having gained more experience by designing more programs, the reasoning made in this section can often be done quite quickly, requiring less diagrams, with less detail.

The `registerCustomer` System Operation

The next system operation is `registerCustomer`. The first thing to do is to introduce the system operation as a method in the controller, since all system operations shall appear as controller methods. What objects shall the `registerCustomer` method call? This is the time to introduce model objects, since the result of customer registration is needed in future system calls, for example when a rented car is associated with the renting customer. It seems quite natural to add a `Customer` class, there is also such a class in the domain model. Again comes the same consideration as for the `Car` class, is this a DTO or an entity? That question can not be properly answered until more is known about the program. For the moment, we choose an easy solution and treat it as we treated the `car` object. Make the class a DTO, place it in the lowest layer where it is used, namely `model`, and create the object in `view`. Treating `car` and `customer` the same way should also make it easier to understand the program. We must, however, be aware that these choices might have to be changed later on, as designing proceeds. Having solved this problem, at least temporarily, another question immediately appears. According to both domain model and SSD, `CustomerDTO` has the attributes `name`, `address` and `drivingLicense`. Shall these be strings or new classes? In order to shorten the discussion, the domain model is followed without further consideration. This means `name` becomes a string, while the other two becomes new classes, `AddressDTO` and `DrivingLicenseDTO`. Now it is possible to draw a UML diagram illustrating the call of the `registerCustomer` method, figure 5.30.

We are not done yet, sending a DTO to the controller does not serve any purpose on its own. Remember that a DTO shall be considered a data type, like `int` or `String`. To complete customer registration, customer data should reasonably be stored somewhere in the model (or database). This is a good time to add the `Rental` class, which is a central class in the domain model. High cohesion is achieved by letting a `rental` object represent the entire rental transaction. This object will know which customer performed the rental, which car was rented, and other facts related to the rental, by having references to appropriate other objects. The final design of the `registerCustomer` system operation look as in figure 5.31.

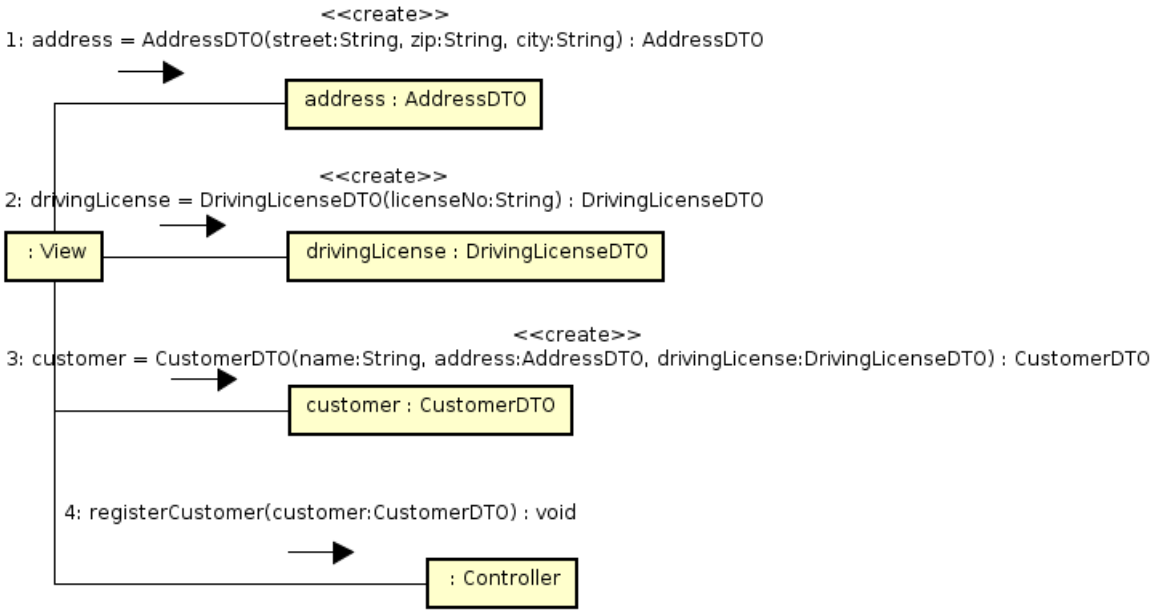


Figure 5.30 The `customer` object and the `registerCustomer` method.

There is no need to add anything to the `main` method, since all new objects that were introduced in this system operation are created in the interaction diagram in figure 5.31. These objects are `address`, `drivingLicense`, `customer` and the unnamed `Rental` object.

A word of caution before proceeding to the next system operation. There are now three different DTOs stored in the model, in the `Rental` object, and there will likely be even more as we proceed. We have not considered how these are handled in the model. Are they simply kept or is all data copied to some other object? This question will be left unanswered until the design is implemented in code. However, it is a problem that the DTO objects are referenced, and therefore potentially updated, by both `view` and `model`. Once `view` has passed a DTO as parameter to `controller`, it should never be updated again by `view`. To make sure this does not happen, all fields, and also the classes themselves, shall be `final`. This makes the DTO *immutable*, which means none of its fields can ever change value. There is no UML symbol for this, it is illustrated with a note in the class diagram, figure 5.32.

The `bookCar` System Operation

The purpose of the `bookCar` system operation is to register which car will be rented. Note bullet ten in the specification, *The program registers that the car is rented by the customer*. This is not illustrated in the system sequence diagram. Correctly, since it is an operation internal in the system, but still it must appear in the design.

The name of the parameter that specifies the car in the SSD is `foundCar`, which is the same name as the variable that was returned from the `searchMatchingCar` system operation. This indicates that these two are the same object. This information will be kept in the design, by using the same name, `foundCar`, for both objects also here.

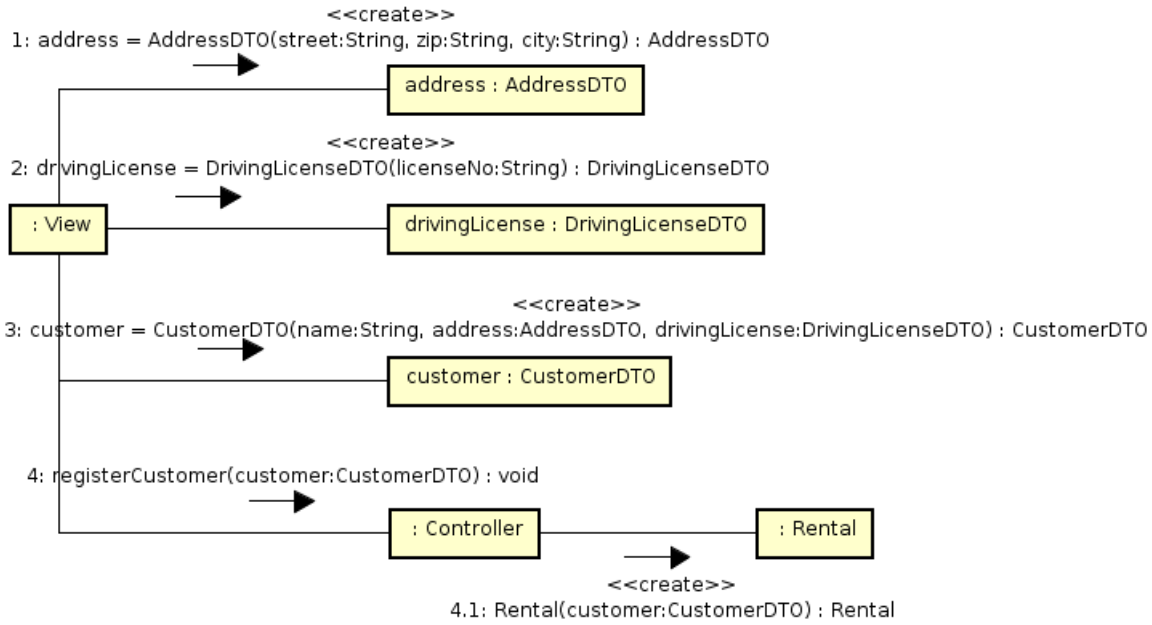


Figure 5.31 The complete design of the `registerCustomer` system operation, the `Rental` class symbolizes the entire rental transaction.

The `foundCar` object must be passed to the `rental` object in the model, to be associated with the current rental and thereby the current customer. That is not enough however, the car database must also be updated to show that the car is now booked, to prevent other customers from renting it. This raises the question, what to store in the database? Just the fact that the car is now booked, or all information in the entire rental object? The latter must be the correct choice, otherwise the information about this rental will be lost when the next rental is initiated. Of course, in a real project, this would be discussed with a domain expert (someone working at the car rental company), but here we have to decide on our own. The decision to store all rental information creates a new problem, shall a rental be stored in a database named `carRegistry`? Isn't that name a bit misleading? Either the name `carRegistry` must be changed, or a new data store must be created. Let's try to get inspiration from the domain model, it shows a `RentalCatalog` and a `CarCatalog`. This indicates that there should be different data stores for cars and rentals. It can also be argued that separating these two classes creates higher cohesion. However, these two are not exactly what we are looking for, in the DM they contain *specifications* of rentals and cars, but in the design we are handling particular *instances* of rentals and cars. Also, comparing the DM and the design diagrams, it becomes clear that the design so far contains no rental or car *specification* stores, is that a problem? This is again something that should be discussed with the domain expert. But let's not deviate to much from the SSD we are implementing now, we will not consider car or rental specifications here, since they are not mentioned in the specification.

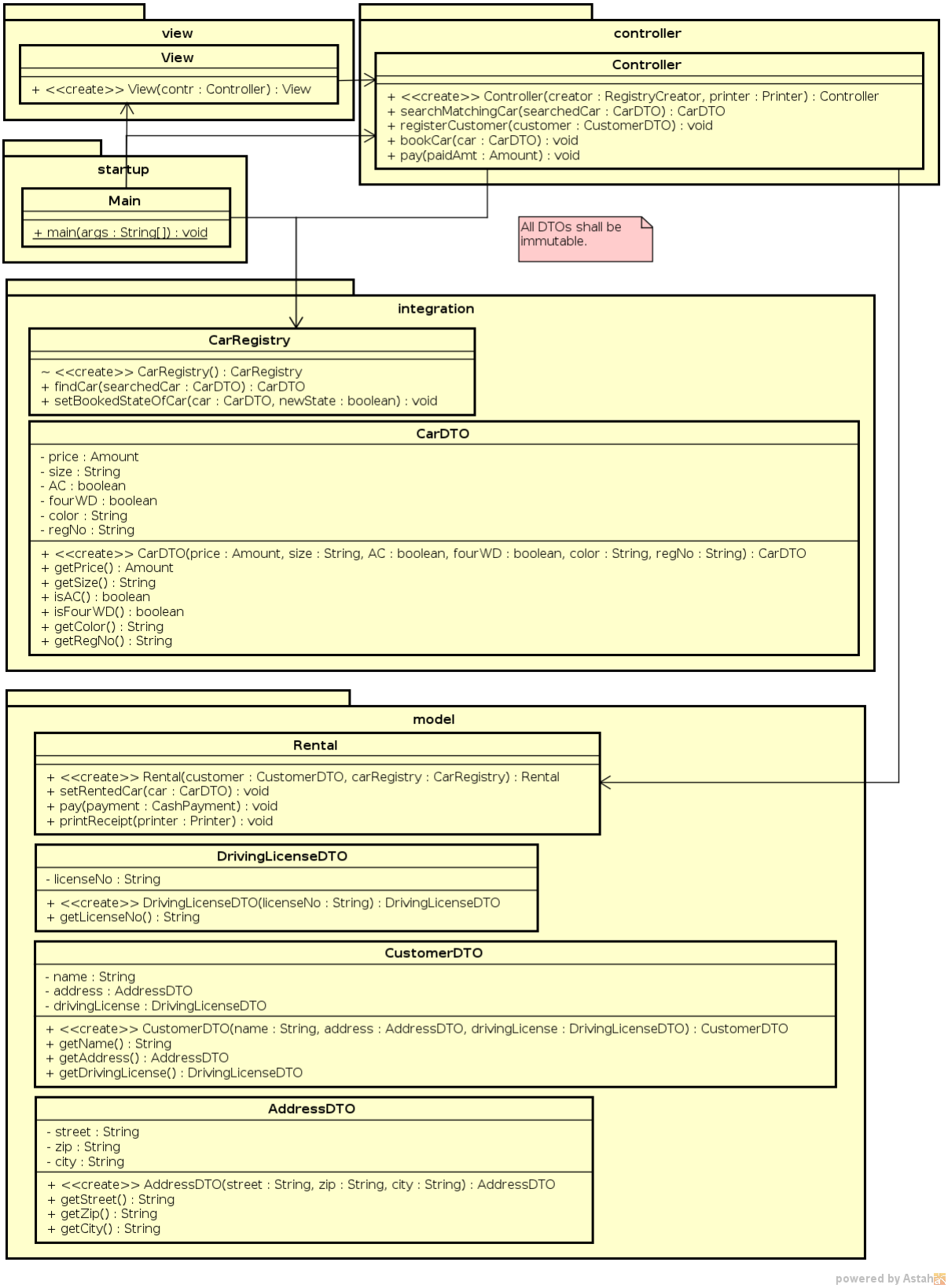


Figure 5.32 `RentCar` design class diagram after designing `registerCustomer`.

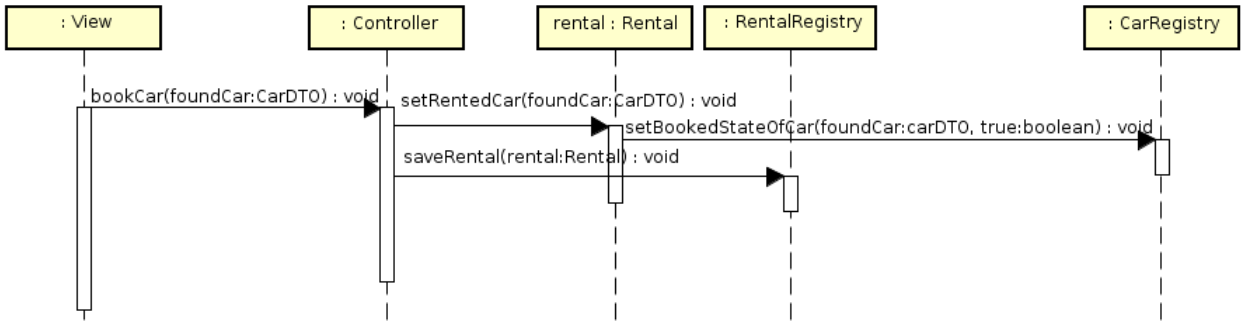


Figure 5.33 The `bookCar` design interaction diagram.

With the cohesion argument, a `RentalRegistry` is added, this gives higher cohesion than storing rentals and cars in the same data store. This results in the design in figure 5.33. Remember that `CarRegistry` and `RentalRegistry` are not the actual databases, but classes calling the database. There is nothing stopping us from letting both those call the same underlying database if that would be appropriate. In this course, we do not implement any database, so we do not have to consider that problem.

Note that the `CarRegistry` method that marks the car as being booked is called `setStateOfBookedCar`, and not `bookCar`. The reason is that booking the car is business logic, which belongs in the model. The purpose of the integration layer is to access the data store, not to perform business logic, like booking a car. Therefore, this method just saves a booked state, without caring about eventual business rules related to performing a booking.

It is a quite interesting decision to let `Rental`, instead of `Controller` call `setStateOfBookedCar`. The motive is that the controller should not have detailed knowledge about all details of all system operations. That would lead towards spider-in-the-web design, with the controller as spider. Now that calls to registers are being made from `Rental`, it might be adequate to move more register calls from `Controller` to `Rental`. Then there would of course be the risk that, in the end, `Controller` does nothing but forward calls to `Rental`, which then becomes the spider class. This reasoning is not an unimportant academic exercise, on the contrary it is quite common design problems both to have a controller doing all work itself, and to have a controller doing nothing but forwarding method calls to another class. To shorten this discussion a bit, the design is kept as in figure 5.33.

The `RentalRegistry` object is not created in any design diagram, it must therefore be created when the system is started. Figure 5.34 shows program startup with instantiation of `RentalRegistry` added. With this modification, `main` creates two objects in the `dbhandler` layer. This is a warning sign that it might be getting unnecessarily high coupling to that layer. Also, the `dbhandler` layer might have a bit bad encapsulation, since it has to reveal the existence of `CarRegistry` and `RentalRegistry` to main. These problems can be solved by changing that startup design to the one in figure 5.35, where the class `RegistryCreator` is responsible for creating the registries, thus hiding their existence to `main`. The design in any of figures 5.34 or 5.35 can be used, since the problem regarding encapsulation in the `dbhandler` layer is not yet very big. But it might grow in the future, if more registries are added.

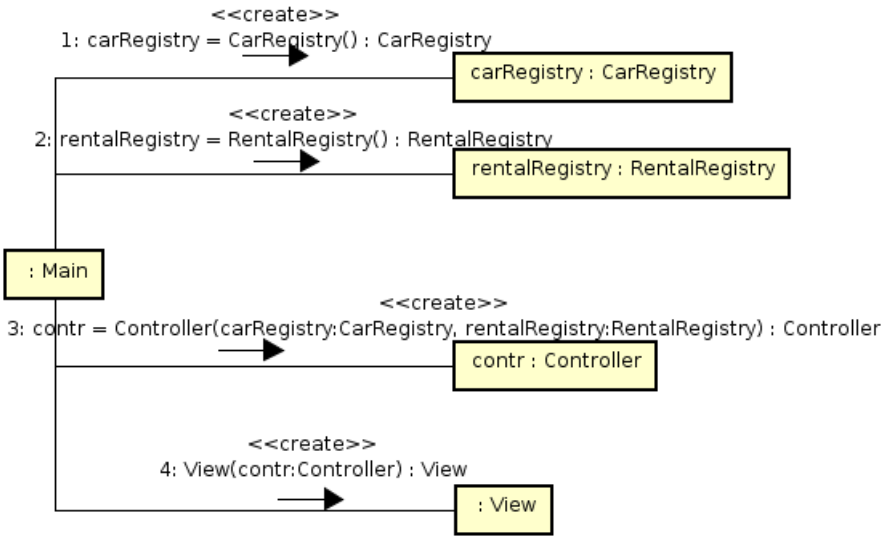


Figure 5.34 The start sequence in the `main` method.

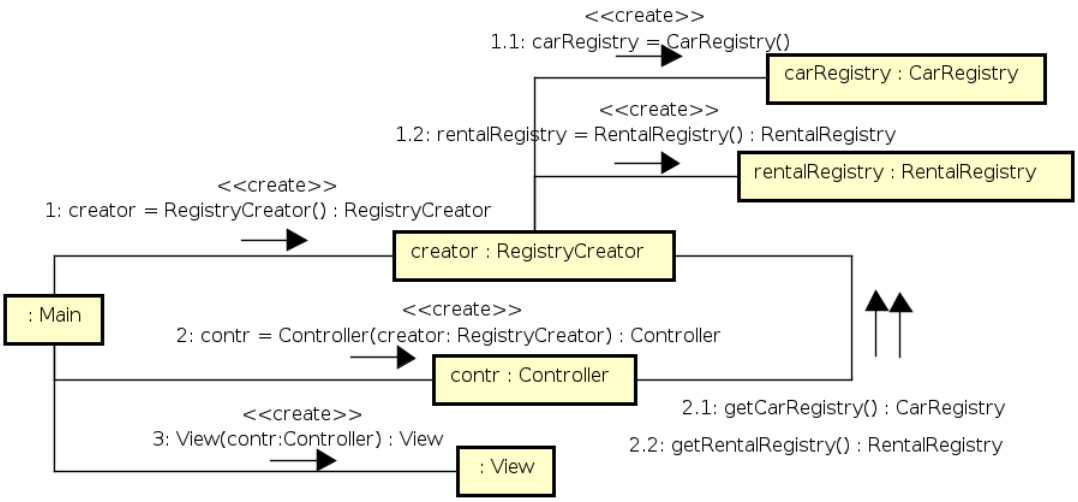


Figure 5.35 The start sequence when `RegistryCreator` is added.

The design class diagram, figure 5.36, is now becoming quite big. In order to reduce it, the DTOs are omitted. Another option would have been to split it into more, smaller diagrams. This class diagram illustrates the start sequence in figure 5.35, not 5.34. Note that the constructors of `CarRegistry` and `RentalRegistry` are package private, since they are called only by `RegistryCreator`, which is located in the same package as those registries.

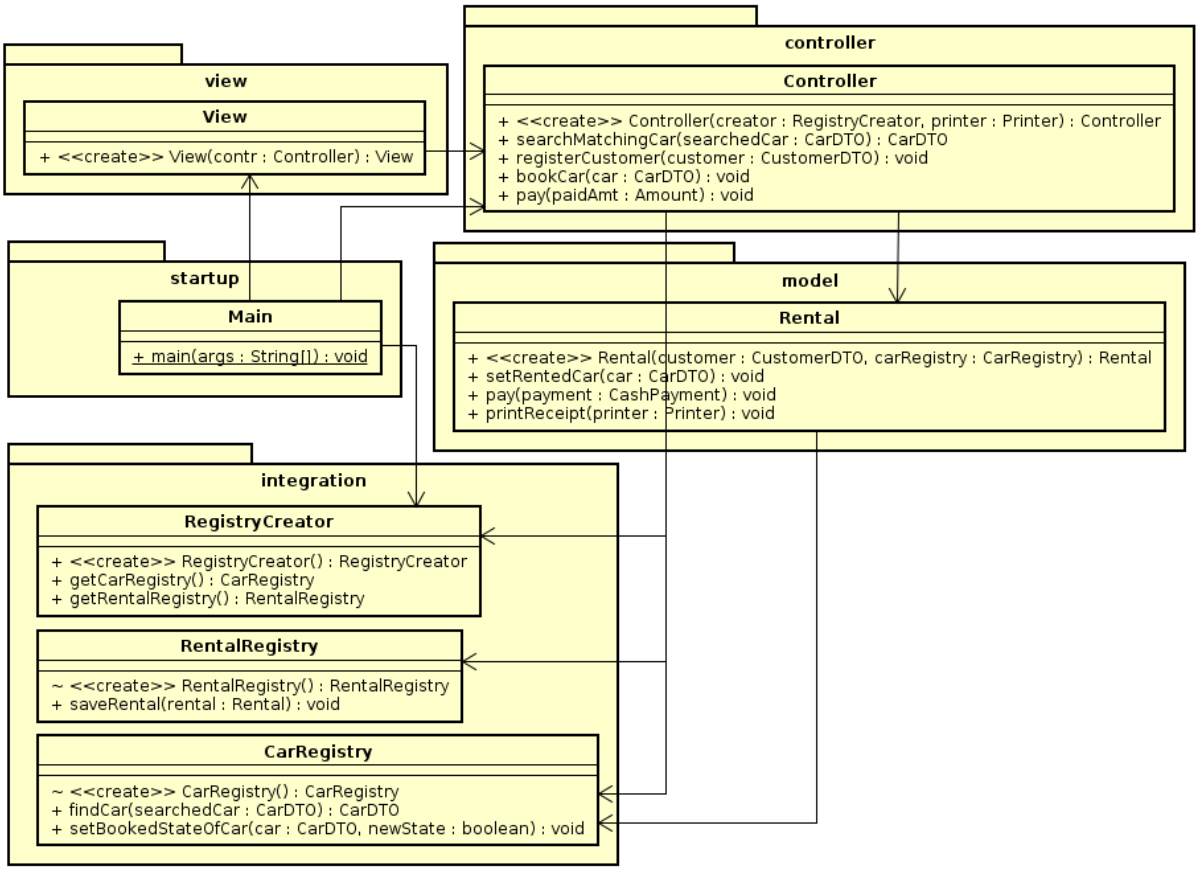


Figure 5.36 `RentCar` design class diagram after designing `bookCar`.

The `pay` System Operation

Only cash payment is implemented in the current iteration. Just as is the case for any system operation, there will be a method in the controller with the same signature as the operation in the SSD, that is `void pay(amount)`. What is the type of the parameter `amount`? So far, `int` has been used to represent amounts, for example the price of a rental. Looking in the domain model, however, there is a class called `Amount` that represents an amount of money. It is most likely a good idea to change the design to use that type instead. Generally, it is a bit dangerous to force an amount to have a specific primitive type. For example it is not clear whether an amount can have decimals or not. By introducing the `Amount` class, the primitive type of the amount is encapsulated inside that class, and can thus easily be changed. It is a great joy to see that the introduction of `Amount` only requires changes from `int` to `Amount` in one single class, `CarDTO`. This is due to the encapsulation of car properties in `CarDTO`. The `pay` interaction diagram now looks as in figure 5.37.

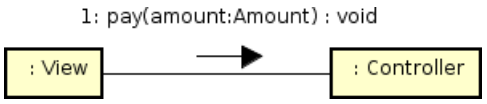


Figure 5.37 The `pay` system operation.

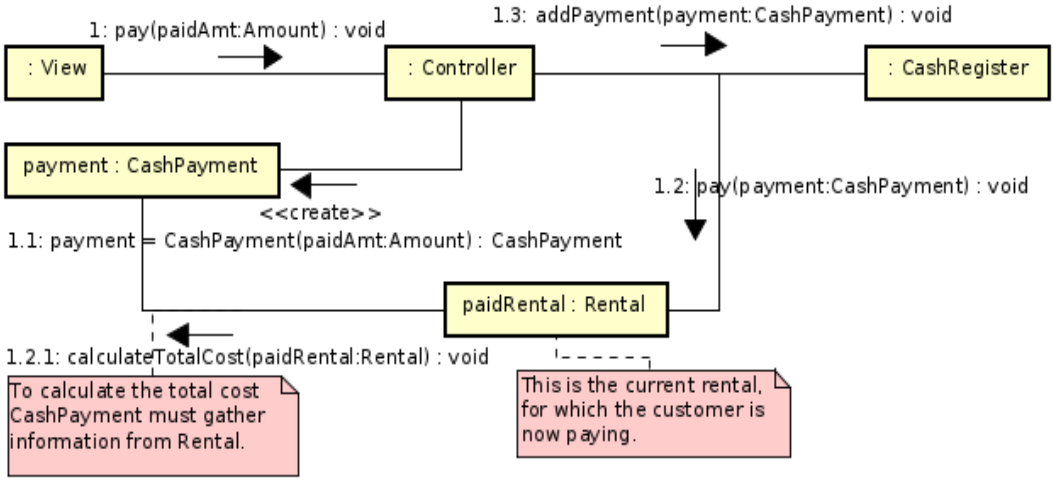


Figure 5.38 `Payment` and `CashRegister` handling the `pay` system operation.

Which object in the model shall handle a payment? `Rental` is the only class that can come under consideration without completely ruining cohesion. Is it reasonable that `Rental` shall prepare the receipt and perform the task listed in bullet 14 in the scenario, namely to *update the balance*? The answer must be no, `Rental` represents a particular rent transaction, it is not responsible for receipt creation or for maintaining the balance of the cashier's cash register. This means a new class must be created. The DM has no really good candidate for this class, which probably means something was missed when it was created. Possible classes in the DM are `Payment` and `Cashier`. The former is associated to `Receipt`, which in turn is associated to `Change`, and seems to be a good candidate for handling one specific payment. One specific payment is, however, not related to the balance in a cash register. Therefore, `Payment` shall not handle the balance. Looking in the DM, `Cashier` is the only possible candidate for handling the balance, but is a balance really an attribute of the cashier that worked at the cash register where the balance was generated? The answer must be *no*, which means none of the classes in the DM can be used. The most reasonable solution seems to be to introduce a new class `CashRegister`, representing the cash register that has the particular balance. The `pay` design, after introducing the `Payment` and `CashRegister`, is depicted in figure 5.38. The payment class is called `CashPayment` instead of `Payment`, because we are anticipating that future handling of credit card payments will be quite different and therefore be placed in a different class. The rental that is being paid is passed to `calculateCost`, call 1.2.1, since `CashPayment` will have to ask the payment about information when calculating the total rental cost.

As a result of `pay`, a receipt shall be printed on the printer, which, according to the SSD, is an external system. Calling an external system is normally handled by a specific class, which represents the external system and handles all communication with it. This class can be named after the external system, here, it will be called `Printer`. This class is *not* the actual printer, but a representation of the printer in the program being developed. In which layer shall this class be placed? Actually, it does not fit in any of the current layers without giving bad cohesion to the layer where it is placed. There are two main options, either to

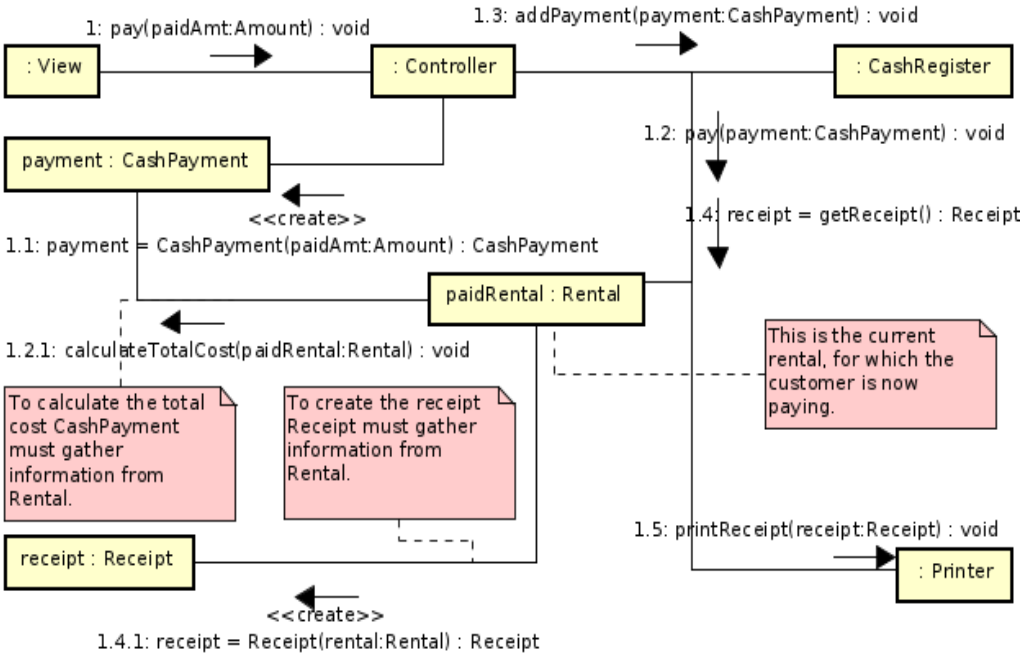


Figure 5.39 Receipt printout is added.

create a new layer or to extend (and rename) `dbhandler` to handle interaction with any other system, so far databases and printers. The former seems like a road to fragmentation of the system into many small layers, to high coupling with many references, and to less possibility for encapsulation, given the many small units. The latter option seems to be a road to low cohesion in `dbhandler`. With the current knowledge about the system, it is quite impossible to tell which option is the best. More or less by chance, the latter option is chosen and the `dbhandler` layer is renamed to `integration`, which is a relatively commonly used name for a layer responsible for interaction with external systems. The resulting `pay` design can be seen in figure 5.39. This design is a bit underspecified, for example it is not clear exactly how `Receipt` will gather the receipt information. Actually, it is not even clear exactly which information the receipt shall contain. However, it is clear that what is designed is sufficient to allow `Receipt` to gather the information from `Rental`. The remaining details will be decided when the design is implemented in code.

Two objects were introduced without being created, namely the `Printer` and `CashRegister` objects, which must therefore be created during startup. Shall `Printer` be created by `Registry Creator` (which must then be renamed), or shall it be created directly by `main`? Let's *not* include it in `RegistryCreator`, since, after all, a printer connection is completely different from a database connection. The `RegistryCreator` will be responsible only for connecting to the database. Perhaps the same connection can be used for both the car and rental registries, but most certainly not for the printer. This decision gives the final startup design of figure 5.40.

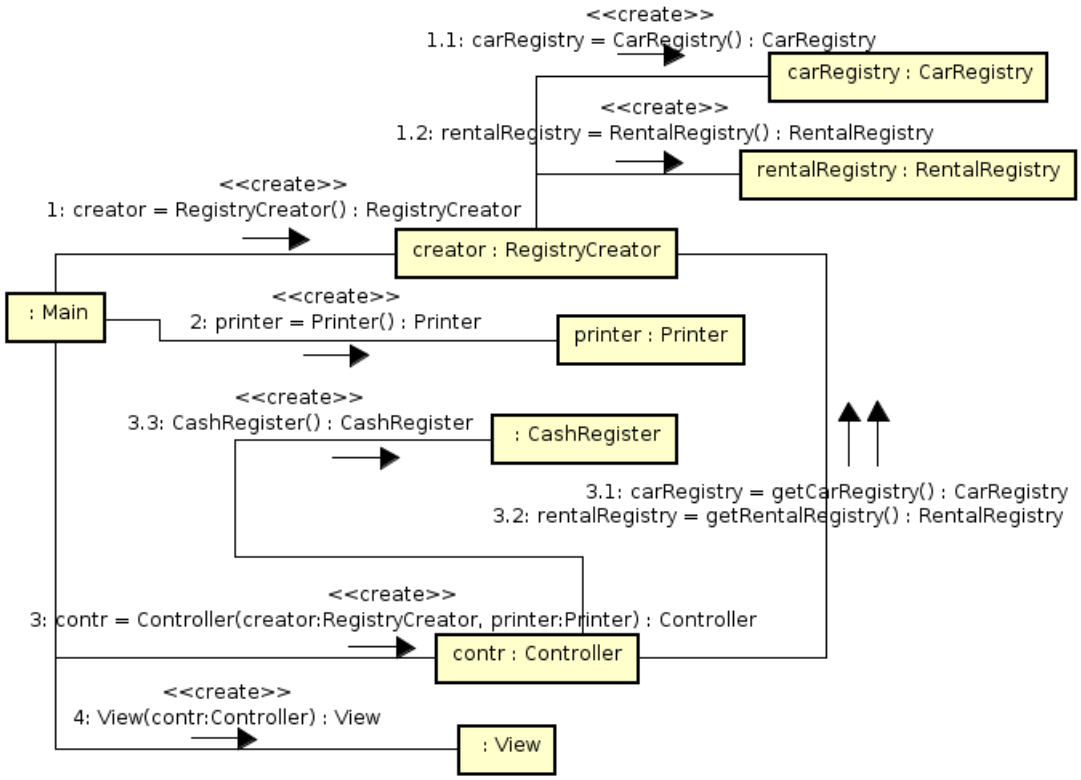


Figure 5.40 The complete startup design.

Why is the `CashRegister` object created by `Controller`, when all other objects are created by `main` and sent to `Controller`? This is a trade-off between two contradicting arguments. On one hand, coupling is lowered if `main` is not associated to all other objects created during startup. On the other hand, cohesion of the `Controller` constructor is increased if it does not create loads of other objects, besides `controller`. The design of figure 5.40 balances these arguments. Since `Controller` is the entry point to `model`, it makes sense that it creates the model objects, like `CashRegister`. The `main` method creates central objects in the `integration`, `controller` and `view` layers, but nothing more.

That's it, all operations in the SSD have now been designed. All that remains is to draw the final design class diagram, which can be seen in figure 5.41. Note that there is no `data` layer, it is not needed since there is no database.

Evaluate the Completed `RentCar` Design

Before leaving the design, it should be evaluated according to the concepts encapsulation, cohesion, and coupling. Starting with encapsulation, is there any method, class, package or layer that has bad encapsulation? Bad encapsulation means that there is a member with too high visibility, public instead of package private or private. With the current design, there is no visibility that can be lowered. Still, there are very few methods that are not public, which

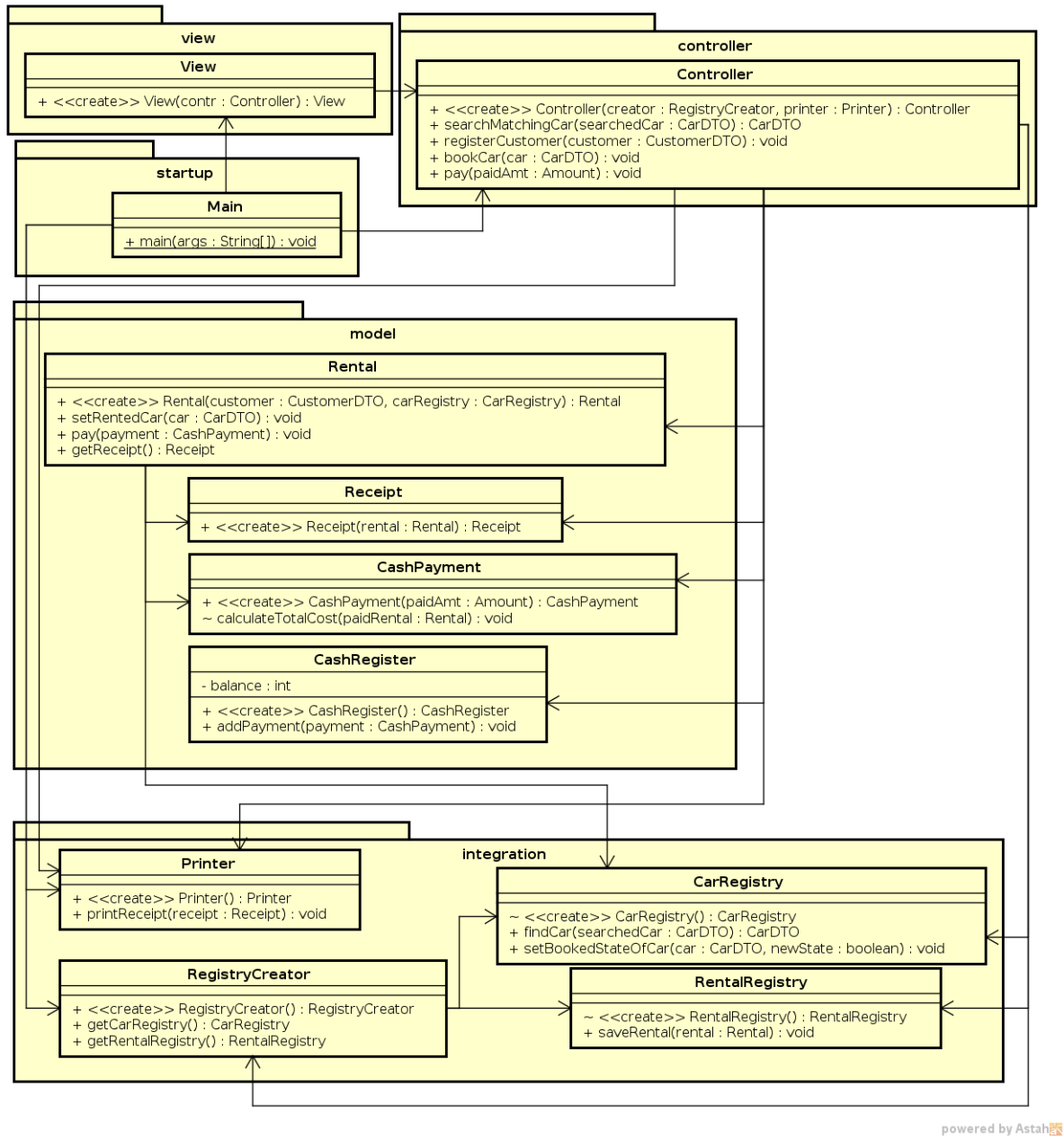


Figure 5.41 The class diagram after all system operations have been designed.

is not the desired result. Also, there is quite high coupling. For example, `Controller` is associated with all classes in `model` and `integration`. This situation would be improved if `model` classes called each other, and also called `integration` classes, instead of passing through `controller`. One thing that can be done is to change the `Rental` method `Receipt getReceipt()` to `void printReceipt(Printer printer)` and move the printer call from `Controller` to `Rental`. This improves the design quite a lot, it removes the association from `Controller` to `Receipt`, removes the association from `Controller` to `Printer`, and

makes the `Receipt` constructor package private. This gives the `pay` design in figure 5.42, reflected in the class diagram in figure 5.43. This is at least a bit better. Further improvements could be considered, for example to somehow let `CashPayment` call `CashRegister` (or vice versa). This would remove one more association from `controller`. However, the current design is quite acceptable, let's leave it like that. It is normal that early in development, a large part of the created members belong to the public interface. The last thing to do is to consider cohesion. There seems to be no issues related to cohesion, all layers, classes and methods do what they shall, nothing more.

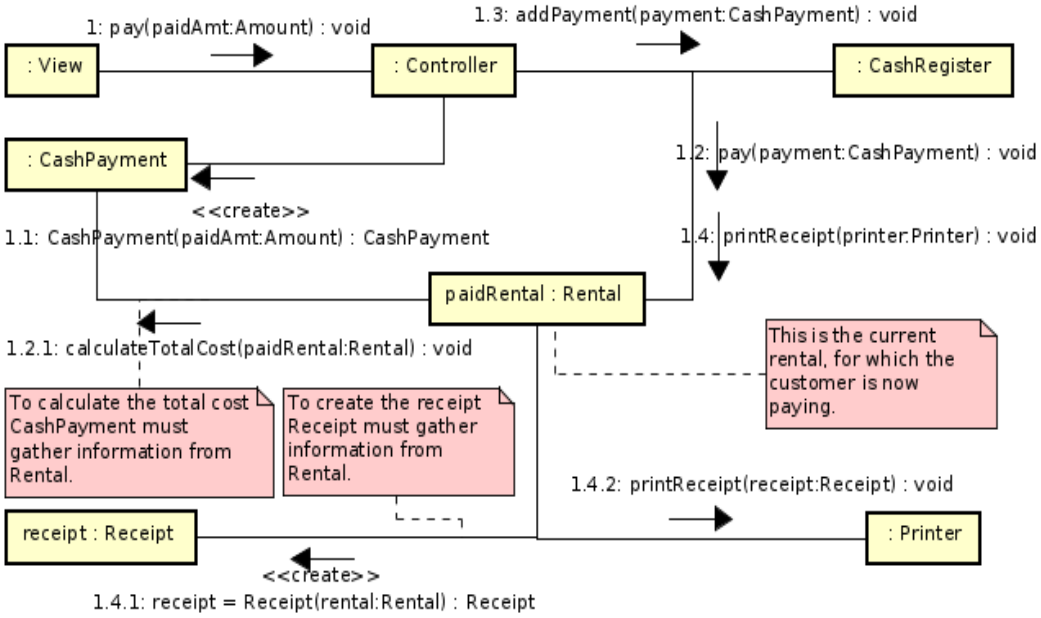


Figure 5.42 Improved `pay` design, with less coupling from `Controller`.

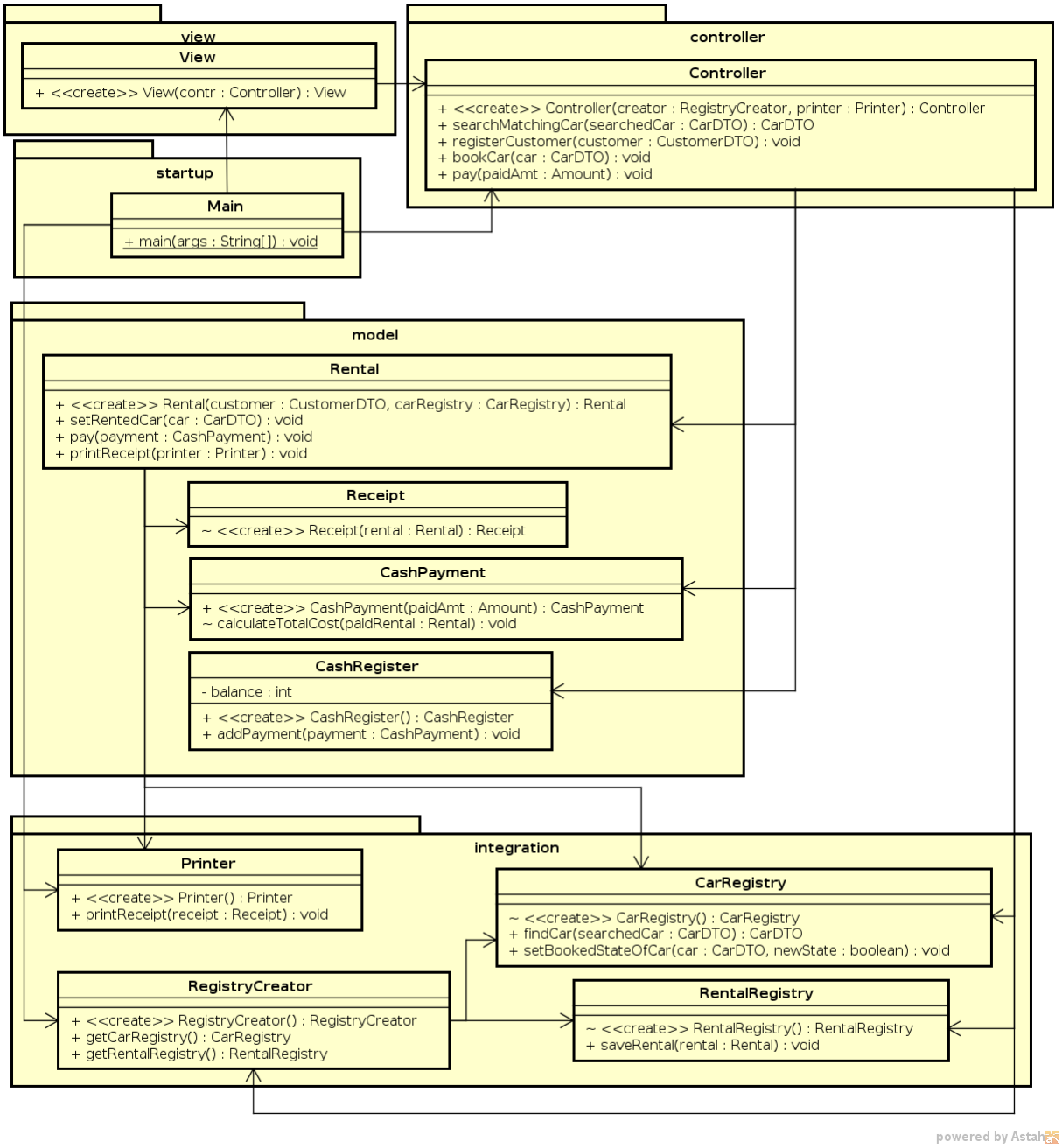


Figure 5.43 The final design class diagram.

5.6 Common Mistakes

Below follows a list of common design mistakes.

Spider-in-the-web class The design has a spider-in-the-web class, which is often the controller. The solution is to remove associations between the spider and some peripheral classes, and instead add associations between peripheral classes. This has been covered in detail previously, when the receipt printout was redesigned.

Too much primitive data Objects are not used sufficiently, instead primitive data is passed in method calls. It is not forbidden to use primitive data, but always consider introducing objects, especially if there are long lists of parameters in a method or of attributes in a class. Not using objects means the prime tool (objects) for handling encapsulation, cohesion and coupling is thrown away.

Unwarranted static methods or fields It is not forbidden to use static members, but there must be a very god reason why there are such. Static members do not belong to any object, and therefore, just as is the case for primitive members, using them means the prime tool (objects) for handling encapsulation, cohesion and coupling is thrown away.

Too few classes It is of course very difficult to tell how many classes there should be in a certain design, but in some cases there are clearly too few. An example is if the model consists of only one class, which performs all business logic. Cohesion is the prime criteria used to decide if there is a sufficient number of classes, too few classes normally means that some existing class(es) has low cohesion.

NO!

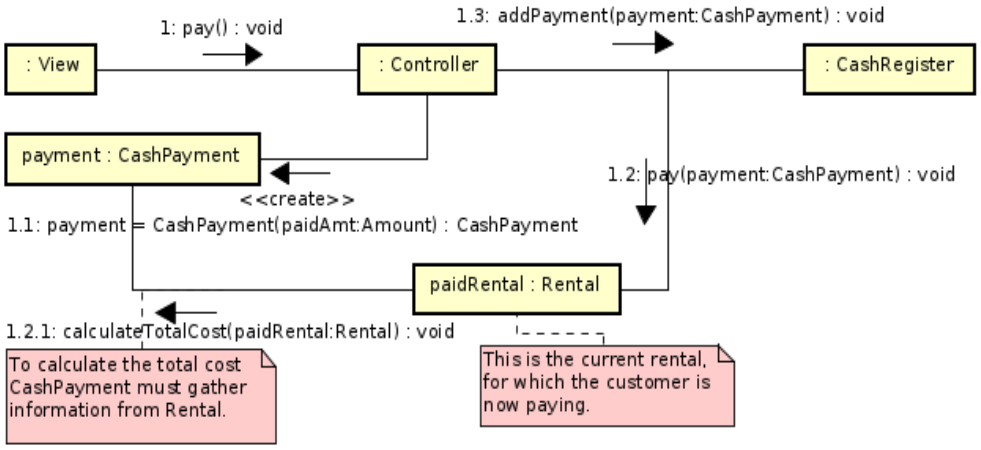
Too few layers This is perhaps a less disastrous mistake than too few classes, especially early in the development, when the program is relatively small. Still, if one of the layers `view`, `controller`, `model`, `startup` or `integration` is missing, there must be a reason why that is the case. If one or more of those layers has another name is probably no problem, what matters is that they exist.

Too few methods This can be discovered by evaluating if existing methods have high cohesion. A method should have one specific task, explained by its name. However, there can be too few methods even if all existing methods do have high cohesion. This is the case if some of the program's tasks is simply not performed at all. If so, the design is not complete and a new method must be introduced, performing the missing task.

MVC and layer patterns used wrong way Under no circumstance must there be any form of input or output outside the view. Also, the model shall not contain any call to a database or any other external system. The integration layer, on the other hand, shall only contain calls to external system, and no business logic. Furthermore, all layers must have high cohesion and there should be calls only from higher (closer to the user) layers to lower layers.

Data appears out of nothing Always consider if it is possible to implement the design in code. That's for example not possible if a certain variable is passed in a method call, but the value of that variable is not known to the calling object. As an example, consider figure 5.44, where the call from `View` to the `pay` method in `Controller`, call 1, has no parameter. This solution won't work, since there's no way the value of `paidAmt` can be known in call 1.1.

NO!

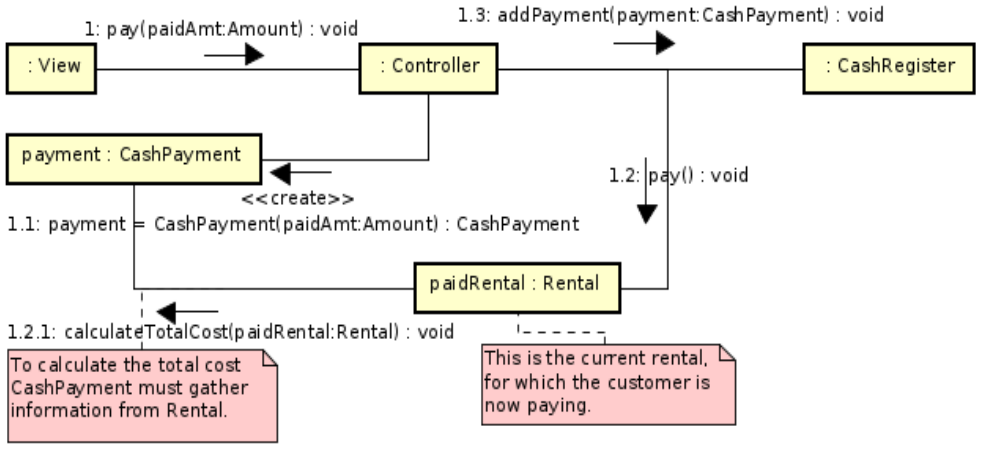


NO!

Figure 5.44 This is the same design as in figure 5.39 above, but the paid amount parameter is missing in call 1.

Calling an object without a reference to that object When a method call is made, the calling object must have a reference to the called object. Also, the design must show how the calling object got that reference. In figure 5.45, the object `paidRental` never gets a reference to the object `payment`, which means call 1.2.1 can't be made.

NO!

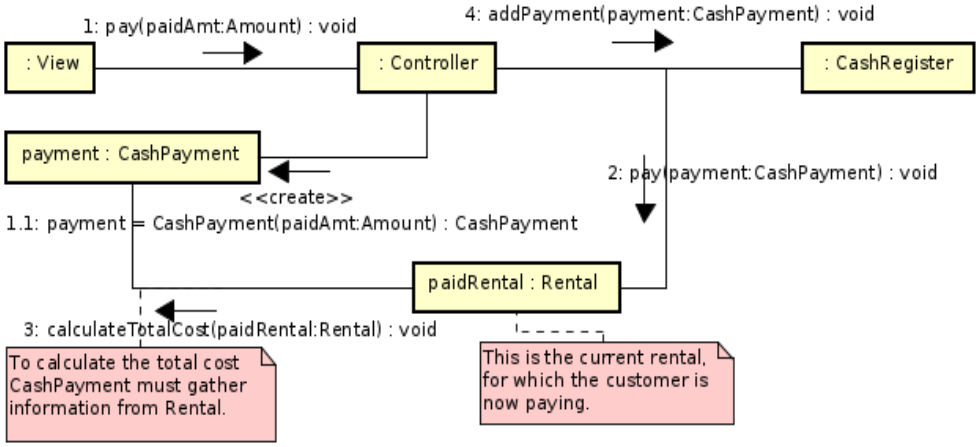


NO!

Figure 5.45 This is the same design as in figure 5.39 above, but call 1.2.1 can't be made since no reference to `payment` is passed to `paidRental` in call 1.2.

Sequence instead of dot notation for message numbers A method call made by the method called in call 1 shall have number 1.1, and a method call made by 1.1 shall have 1.1.1, etc. Using a sequence of numbers (1, 2, 3, ...) instead of this dot notation is wrong. Figure 5.46 is an example of wrong numbering. It shows that the `pay` method, in call 1, ends after call 1.1 is made. Then, at some later point in time, method calls 2, 3 and 4 are made for no apparent reason. The figure doesn't tell what causes calls 2-4.

NO!

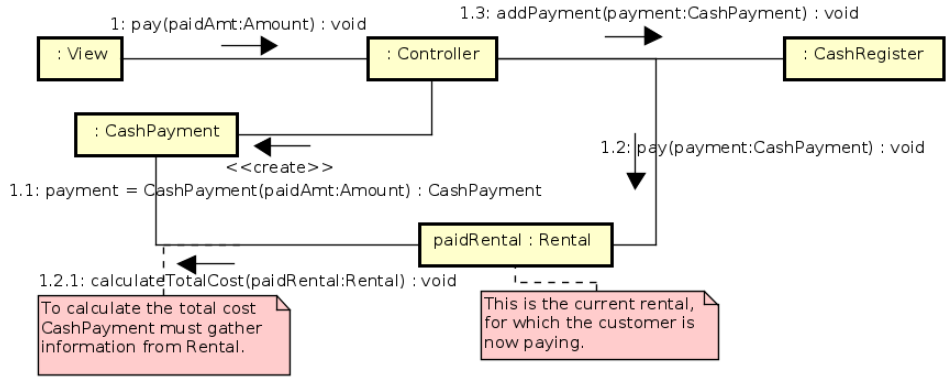


NO!

Figure 5.46 This is the same design as in figure 5.39 above, but method call numbers are wrong. This diagram has no implementation in appendix C, since it's incorrect and can't be implemented.

Objects are not named when required An object shall have a name whenever it appears in more than one place in an interaction diagram. The object can appear for example as an object symbol, as a method parameter or as a return value from a method. This mistake is made in figure 5.47, where the `CashPayment` object created in call 1.1 has no name. This makes it much harder to understand that this object is the same as the parameter `payment` in calls 1.2 and 1.3.

NO!



NO!

Figure 5.47 This is the same design as in figure 5.39 above, but the `CashPayment` object isn't named.

Wrong return type It's for example a quite common mistake to make a method in the controller void, even when it's supposed to return something to the view.

Erroneous constructor A constructor must always have the same name and return type as the class where it's located. It must also have the stereotype «`create`».

Too big class diagram The class diagram might be too big, and therefore unreadable. The diagram might be unreadable because it is messy, showing many details, or because it has been shrunk to fit on a printed page, making the text too small. The solution is to either split it in more, smaller, diagrams, or to remove details that are not needed to understand the design. Examples of things to remove are DTOs, private methods, and private attributes. Remember that the goal of removing details is to make it *easier* to understand the diagram, do not remove things that are required for understanding. After having removed details, it might become difficult to use the class diagram as a template when coding. However, there should still be a complete design of each class in the design tool, which can be used when programming.

NO!

Too big communication diagram Also a communication diagram can be messy and difficult to understand. This typically happens if too many system operations are illustrated in the same diagram. It can also happen that the design of one single system operation, with many alternative flows, can be made clearer by showing different alternative flows in different communication diagrams.

Chapter 6

Programming

This chapter describes how to implement the design in code, which is never just a straightforward translation of the design diagrams. Most likely there are coding details that were not considered during design, and it is also quite likely to discover actual design mistakes. This means there is no sharp line between design and implementation activities, there will be need for decisions regarding program structure, encapsulation, cohesion, coupling and so on also when coding. In addition to this, there will also be questions regarding code quality, that are not really design issues, for example how to name variables and how to write comments in the code.

6.1 Dividing the Code in Packages

A package name consists of components, separated by dots. The first components shall always be the reversed internet domain of the organization, for example `se.kth`. This is to avoid name conflicts with packages created by other organizations. Following that, there are normally components that uniquely identify the product within the organization, e.g., department and/or project names, like `iv1350.carRental`. Finally, there are the components that identify a particular package within the product. This part often starts with layer name, for example `model` for a package in the model. If the layer is large, a single package containing everything in that layer might get low cohesion. If so, the package can be divided according to functionality, e.g., `payment`. When following all these rules and guidelines, a package in the model of the car rental application, handling payment, shall be named `se.kth.iv1350.carRental.model.payment`

Sometimes a class does not clearly belong to a specific layer, but is needed in many different layers. This might be because of a design mistake, but it can also be that the class is a *utility class*. These are normally not application specific, but provide some kind of general service, for example string parsing or file handling. Such utility classes are often placed in a package which does not belong to a specific layer, but is instead called for example `util`. The full name of that package in the car rental application would be `se.kth.iv1350.carRental.util`.

6.2 Code Conventions

It is essential that code is easy to understand, since it will, most likely, be read and changed by many more developers than the original creator. To make the code easy to understand, everyone must agree on a set of rules for formatting, naming, commenting, etc. These rules form a *code convention*. Originally, there was a Java code convention published by Sun Microsystems. It is no longer maintained by Oracle, but is still available at [JCC]. A good summary of Java coding standards, which is close to the original code convention, is available at [JCS]. In addition to these documents, organizations that produce code often have their own code convention. It is essential to agree on which code convention to follow in a particular project.

Below, in table 6.1, follows a brief summary on very frequently used naming conventions for Java. Note, however, that a full code convention is much more extensive than these short rules. It is a good idea to read through one of the documents mentioned above.

Name Type	**Description**	**Example**
package	First letter of each part lowercase. Start with reversed internet domain, continue with product name and end with unique package name	`se.kth.iv1350.rentCar.model` or `se.kth.iv1350.rentcar.model`
class and interface	Full description, first letter of each word uppercase	`CashRegister`
method	Full description, first letter lowercase, first letter of non-initial words uppercase.	`calculateTotalCost`
variable, field, parameter	Full description, first letter lowercase, first letter of non-initial words uppercase.	`paidRental`
constant	This applies to `final static` fields. Only uppercase letters, words separated by underscores.	`MILES_PER_KM`

Table 6.1 A few, very commonly used, Java naming conventions.

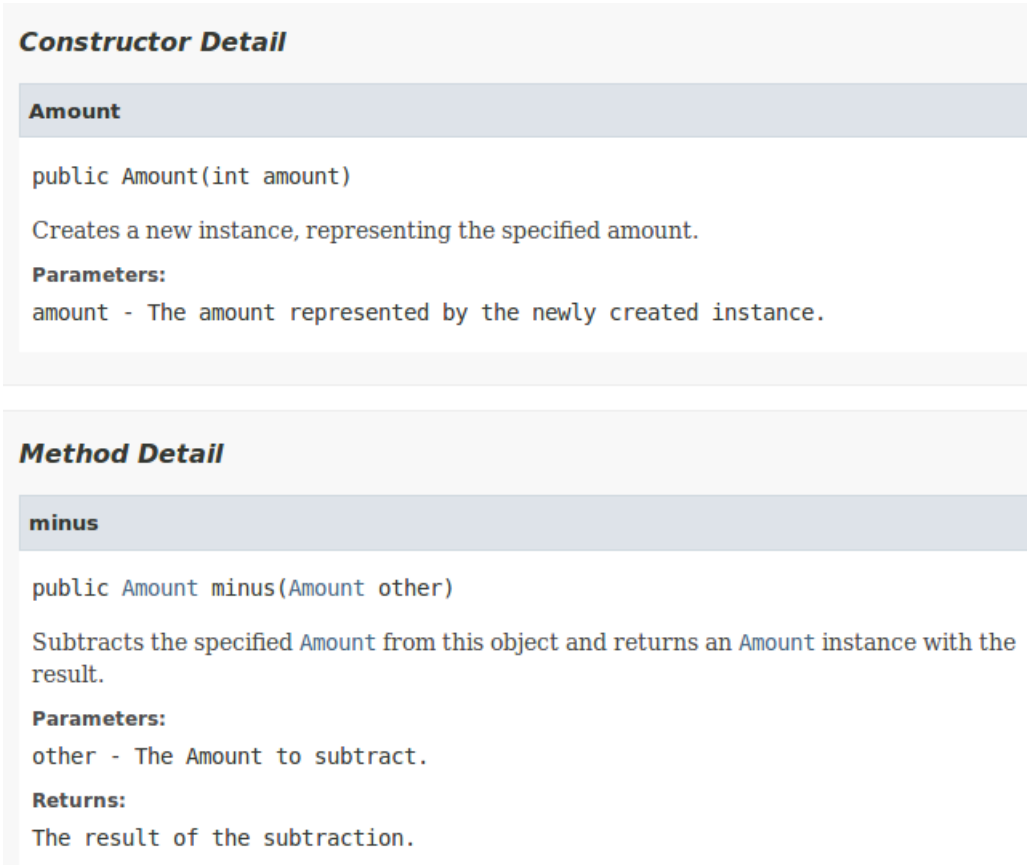
6.3 Comments

There shall be a javadoc comment for each declaration belonging to a public interface. Javadoc comments start with `/**` and end with `*/`. These are used to generate html files with API documentation using the `javadoc` JDK command. Most IDEs (for example NetBeans, IntelliJ and Eclipse) provide a graphical user interface to the `javadoc` command.

A javadoc comment shall describe what the commented unit does, but not how that is done. *How* belongs to the implementation, not to the public interface, and shall therefore not be included in the documentation. Remember that a javadoc comment is part of the contract a declared unit establishes with code using it, and there is no point in promising to perform the job in a particular way, it is sufficient to promise what to do. If the commented unit is a method, the javadoc shall explain not only what the method does, but also its parameters and its return value. These are commented using the `@param` and `@return` javadoc tags. A `<code>` tag shall be used for Java keywords, names and code samples, or, if the code is a declaration in the same program, `@link` can be used instead. The `@link` tag inserts a clickable link to the specified declaration. All tags that have been mentioned here are illustrated in listing 6.1. The `@param` tag is used on lines 11 and 23, the `@return` tag on line 24, the and the `<code>` tag on line 23, and the `@link` tag on lines 19 and 20. The html file generated with the `javadoc` command is, in part, depicted in figure 6.1.

```
1  /**
2   * Represents an amount of money. Instances are immutable.
3   */
4  public final class Amount {
5      private final int amount;
6
7      /**
8       * Creates a new instance, representing the specified
9       * amount.
10      *
11      * @param amount The amount represented by the newly
12      *               created instance.
13      */
14     public Amount(int amount) {
15         this.amount = amount;
16     }
17
18     /**
19      * Subtracts the specified {@link Amount} from
20      * this object and returns an {@link Amount}
21      * instance with the result.
22      *
23      * @param other The <code>Amount</code> to subtract.
24      * @return The result of the subtraction.
25      */
26     public Amount minus(Amount other) {
27         return new Amount(amount - other.amount);
28     }
29 }
```

Listing 6.1 Code with javadoc comments.



Constructor Detail

Amount

```
public Amount(int amount)
```

Creates a new instance, representing the specified amount.

Parameters:

`amount` - The amount represented by the newly created instance.

Method Detail

minus

```
public Amount minus(Amount other)
```

Subtracts the specified `Amount` from this object and returns an `Amount` instance with the result.

Parameters:

`other` - The Amount to subtract.

Returns:

The result of the subtraction.

Figure 6.1 Part of the javadoc generated from the code in listing 6.1.

It is seldom meaningful to add more comments, inside methods. To make sure such comments are up to date is often burdensome extra work, that far too often is simply not done. If comments are not maintained, they will not correctly describe the code, which will result in developers not trusting the comments. Low trust in comments is a very unproductive state of a program. It means both that the commenting work was in vain, and that unnecessary time is spent reading code instead of comments. Therefore, avoid placing comments inside methods. Instead, the need for comments inside a method should be seen as a signal that the method is too long, and ought to be split into shorter methods. More on this below.

6.4 Code Smell and Refactoring

The concept *code smell* describes the state of a particular piece of code. It originates from [FOW1ED], which, despite its age, is still very relevant. There's however a second edition, [FOW], which is a better read today. This book describes certain unwanted states (*smells*) of a code and how to get rid of them. The way to remove a code smell is to *refactor* the code, which means to improve it without changing its functionality. A *refactoring* is a well-defined way to change a specific detail of the code, for example to change a method's name. [FOW]

lists numerous refactorings and tells how to use them to remove different code smells. This section describes a small number of the many code smells and refactorings mentioned in the book. It is of course not necessary to first introduce a smell by making make the corresponding mistake, and then refactor the code. Better is to learn from the particular smell and never make the mistake.

The amount of code smell in a program is a quite sure sign of the programmer's skills. Novice programmers reveal their lack of knowledge by writing code that has several code smells. It is relatively common for employers to test the ability to find such problems in a piece of code, when hiring new programmers.

Duplicated Code

Identical code in more than one place in the program is a really bad smell. It means whenever that piece of code shall be changed, exactly the same editing must be done in all locations where the duplicated code exists. This is of course inefficient since more writing is needed, but far worse is that it is easy to miss one or more code locations, which means the code will not work as expected after the (incomplete) change is made. This will lead to long and boring searches for lines in the program where the duplicated code was not changed as intended.

How sensitive to duplicated code shall one be? The answer is *very sensitive*! The goal must always be that *not a single statement shall be repeated anywhere in the program.* Allowing duplicated code is to enter a road that leads to disaster. Duplicated code is often introduced by copying previously written code, you should hear a loud warning bell ring if you type `ctrl-c ctrl-v` while programming.

As an example, consider the code in listing 6.2, where the printout of the contents of the `names` array is duplicated. In fact, also the javadoc comment to the three methods is duplicated. Duplicated comments introduce exactly the same complications as duplicated code.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * This class has bad smell since it contains duplicated code.
5   * The duplicated  code is the loop printing the contents of
6   * the <code>names</code> array.
7   */
8  public class ClassWithDuplicatedCode {
9      private String[] names;
10
11     /**
12      * To perform its task, this method has to print the
13      * contents of the <code>names</code> array.
14      */
15     public void aMethodThatShowsNames() {
16         //some code.
17         for (String name : names) {
18             System.out.println(name);
```

```
19          }
20          //some code.
21      }
22
23      /**
24       * To perform its task, this method has to print the
25       * contents of the <code>names</code> array.
26       */
27      public void otherMethodThatShowsNames() {
28          //some code.
29          for (String name : names) {
30              System.out.println(name);
31          }
32          //some code.
33      }
34
35      /**
36       * To perform its task, this method has to print the
37       * contents of the <code>names</code> array.
38       */
39      public void thirdMethodThatShowsNames() {
40          //some code.
41          for (String name : names) {
42              System.out.println(name);
43          }
44          //some code.
45      }
46  }
```

Listing 6.2 The loop with the printout of the `names` array is duplicated.

Suppose the printout in listing 6.2 has to be modified, say that lines 18, 30 and 42 shall be changed to `System.out.println("name: " + name);`. This change has to be performed on all three lines. Also, as mentioned above, the fact that there is duplicated code makes it quite difficult to be sure all lines where the code exists where actually changed, especially if the program is large.

This smell is removed by using the refactoring *Extract Method*, which means to move code from an existing method into a new method, which contains this particular code. In the current example, it is the printout loop that shall be placed in the newly created method. This new method is then called on all lines where the printout is required. Listing 6.3 shows the code after applying this refactoring. Here, there is no duplicated code, the desired change is done by editing only line 43.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
```

```
 * This class does not contain duplicated code. The
 * previously duplicated code has been extracted to
 * the method <code>printNames</code>
 */
public class ClassWithoutDuplicatedCode {
    private String[] names;

    /**
     * To perform its task, this method has to print the
     * contents of the <code>names</code> array.
     */
    public void aMethodThatShowsNames() {
        //some code.
        printNames();
        //some code.
    }

    /**
     * To perform its task, this method has to print the
     * contents of the <code>names</code> array.
     */
    public void otherMethodThatShowsNames() {
        //some code.
        printNames();
        //some code.
    }

    /**
     * To perform its task, this method has to print the
     * contents of the <code>names</code> array.
     */
    public void thirdMethodThatShowsNames() {
        //some code.
        printNames();
        //some code.
    }

    private void printNames() {
        for (String name : names) {
            System.out.println("name: " + name);
        }
    }
}
```

Listing 6.3 When the *Extract Method* refactoring has been applied, the loop with the printout of the `names` array is no longer duplicated.

Listing 6.4 shows a more subtle example of duplicated code. The problem here is the code `sequence[1]`, which is used to access the first element in the array `sequence`. This code is wrong, since the first element is located at index zero, not one. To fix this bug, both lines 16 and 23 must be changed. The situation would be even worse in a larger program, where the indexing mistake would occur on numerous lines. Just as in the previous example, the solution is to extract a method containing the duplicated code, see listing 6.5.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * This class has bad smell since it contains duplicated code.
5   * The duplicated code is <code>sequence[1]</code> to access
6   * the first element in the <code>sequence</code> array.
7   */
8  public class ClassWithUnobviousDuplicatedCode {
9      private int[] sequence;
10
11     /**
12      * @return <code>true</code> if the the specified value is
13      * equal to the first element in the sequence.
14      */
15     public boolean startsWith(int value) {
16         return sequence[1] == value;
17     }
18
19     /**
20      * @return The first element in the sequence array.
21      */
22     public int getFirstElement() {
23         return sequence[1];
24     }
25 }
```

Listing 6.4 The duplicated code in this class is the usage of `sequence[1]` to access the first element in the array

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * This class does not contain duplicated code. The previously
5   * duplicated code has been extracted to the method
6   * <code>firstElement</code>
7   */
8  public class ClassWithoutUnobviousDuplicatedCode {
9      private int[] sequence;
10
```

```
11      /**
12       * @return <code>true</code> if the the specified value is
13       * equal to the first element in the sequence.
14       */
15      public boolean startsWith(int value) {
16          return firstElement() == value;
17      }
18
19      /**
20       * @return The first element in the sequence array.
21       */
22      public int getFirstElement() {
23          return firstElement();
24      }
25
26      private int firstElement() {
27          return sequence[0];
28      }
29  }
```

Listing 6.5 The introduction of the method `firstElement` has removed the duplicated code.

All occurrences of the duplicated code were located in the same class in both examples above. This is certainly not always the case, the same code might just as well exist in different classes. Also in this case, the solution is to extract a method with the duplicated code, and replace all occurrences of that code with calls to the newly created method. The specific issue when multiple classes are involved, is where to place the new method. One option is to place it in one of the classes that had the duplicated code, another option is to place it in a new class. In either case, the classes that do *not* contain the new method, must call the new method in the class where it is placed. The best placement must be decided in each specific case, based on how cohesion, coupling and encapsulation are affected by the different alternatives.

Long Method

It is easier to understand the code if all methods have names that clearly explain what the method does. A guideline for deciding if a method is too long is *does the method name tell everything that is needed to fully understand the method body?* If there seems to be need for comments inside a method, that is clearly not the case. In fact, comments or need of comments inside a method is a clear sign that the method is too long. Thus, what matters is not primarily the number of lines in a method, but how easy it is to understand the method body. Consider the method in listing 6.6, which is quite short but still not easy to understand. What is the meaning of the numbers 65 and 90 on line 13? The answer is that the ASCII numbers of upper case letters are between 65 and 90. This becomes clear if a new method, with an explaining name, is introduced, see listing 6.7. To introduce a new method with an explaining name is almost always the best way to shorten and explain a method that is too long.

```
1  /**
2   * Counts the number of upper case letters in the specified
3   * string.
4   *
5   * @param source The string in which uppercase letters are
6   *               counted.
7   * @return The number of uppercase letters in the specified
8   *         string.
9   */
10 public int countUpperCaseLetters(String source) {
11     int noOfUpperCaseLetters = 0;
12     for (char letter : source.toCharArray()) {
13         if (letter >= 65 && letter <= 90) {
14             noOfUpperCaseLetters++;
15         }
16     }
17     return noOfUpperCaseLetters;
18 }
```

Listing 6.6 In spite of the few lines, this method is too long since it is not clear what line 13 does.

```
1  /**
2   * Counts the number of upper case letters in the specified
3   * string.
4   *
5   * @param source The string in which uppercase letters are
6   *               counted.
7   * @return The number of uppercase letters in the specified
8   *         string.
9   */
10 public int countUpperCaseLetters(String source) {
11     int noOfUpperCaseLetters = 0;
12     for (char letter : source.toCharArray()) {
13         if (isUpperCaseLetter(letter)) {
14             noOfUpperCaseLetters++;
15         }
16     }
17     return noOfUpperCaseLetters;
18 }
19
20 private boolean isUpperCaseLetter(char letter) {
21     return letter >= 65 && letter <= 90;
22 }
```

Listing 6.7 Here, each method body is explained by the method's name.

It is sometimes argued that the program becomes slower if there are many method calls. This is simply not true, to perform a method call is not significantly slower than any other statement. Trying to decrease execution time by minimizing the number of method calls is not any smarter than trying to minimize the number of statements in the program.

Large Class

Just as is the case for methods, whether a class is too large is not primarily decided by the number of lines. The main criteria is instead cohesion, a class is too large if it has bad cohesion. Cohesion was covered extensively above, in section 5.2. The class in listing 6.8 shows that cohesion can be improved also by splitting small classes. This listing contains the `Meeting` class, which represents a meeting in a calendar. It has the fields `startTime` and `endTime` that together define the meeting's duration. These two fields are more closely related to each other, than to other fields in the class. The fact that they have a common suffix, *Time*, helps us see this. Cohesion is improved in listing 6.9, by extracting a class with these two fields. This is a quite common way to realize that a new class is appropriate. As programming continues, the new class will probably get more fields and methods.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  import java.time.LocalDateTime;
4
5  /**
6   * This class represents a meeting in a calendar.
7   */
8  public class MeetingLowerCohesion {
9      private LocalDateTime startTime;
10     private LocalDateTime endTime;
11     private String name;
12     private boolean alarmIsSet;
13
14     //More fields and methods.
15 }
```

Listing 6.8 This class has two fields that are more closely related than other fields. This is an indication that cohesion can be improved by extracting a new class, with these fields.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * This class represents a meeting in a calendar.
5   */
6  public class MeetingHigherCohesion {
7      private TimePeriod period;
8      private String name;
```

```
 9     private boolean alarmIsSet;
10
11     //More fields and methods.
12 }
13
14
15 package se.kth.ict.oodbook.prog.smell;
16
17 import java.time.LocalDateTime;
18
19 /**
20  * Represents a period in time, with specific start
21  * and end time.
22  */
23 class TimePeriod {
24     private LocalDateTime startTime;
25     private LocalDateTime endTime;
26
27     //More fields and methods.
28 }
```

Listing 6.9 Here, cohesion is improved by moving the related fields to a new class.

Long Parameter List

Long parameter lists are hard to understand, because it is difficult to remember the meaning of each parameter, especially if there are many parameters of the same type. It is also more likely to have to change a long parameter list than it is to have to change a shorter list, just because the more there is that can change, the more likely it is that something changes. Also, if the method is part of a public interface, changed parameters means changed public interface. When is a parameter list too long? When encapsulation, cohesion or coupling can be improved by shortening it. The criteria is not the number of parameters, but the quality of the code, as is illustrated in the examples below.

The *long parameter list* smell can often be removed with the refactorings *Preserve Whole Object* or *Introduce Parameter Object*, which both replace primitive data with objects. A long parameter list is often long because it consists of primitive data, where objects had been better. Without objects, there is no encapsulation in the list, whenever the need of data changes in the method, it is reflected in its parameter list. *Preserve Whole Object* is illustrated in listing 6.10, that has a long parameter list (line 22), and listing 6.11, where the parameter list is shortened by passing an entire object instead of the fields in that object (line 21). *Introduce Parameter Object* is illustrated in listing 6.12, which has a long parameter list (line 16), and listing 6.13, where the list is shortened by introducing a new object that encapsulates parameters (line 15). It is quite common to discover that this newly created class was needed, and that either existing or new methods belong there.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * This class represents a person. The call to
5   * <code>dbHandler</code> does not preserve the
6   * <code>Person</code> object. The fields are instead passed as
7   * primitive parameters.
8   */
9  public class PersonObjectNotPreserved {
10     private String name;
11     private String address;
12     private String phone;
13
14     /**
15      * Saves this <code>Person</code> to the specified
16      * database.
17      *
18      * @param dbHandler The database handler used to save the
19      *                  <code>Person</code>.
20      */
21     public void savePerson(DBHandler dbHandler) {
22         dbHandler.savePerson(name, address, phone);
23     }
24 }
```

Listing 6.10 The call to `dbHandler` does not preserve the `Person` object. The fields are instead passed as primitive parameters.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * This class represents a person. The call
5   * to <code>dbHandler</code> preserves the
6   * <code>Person</code> object.
7   */
8  public class PersonObjectPreserved {
9      private String name;
10     private String address;
11     private String phone;
12
13     /**
14      * Saves this <code>Person</code> to the specified
15      * database.
16      *
17      * @param dbHandler The database handler used to save the
18      *                  <code>Person</code>.
```

```
19      */
20     public void savePerson(DBHandler dbHandler) {
21         dbHandler.savePerson(this);
22     }
23 }
```

Listing 6.11 The call to `dbHandler` preserves the `Person` object.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * This class represents a bank account. The
5   * <code>deposit</code> method takes primitive parameters
6   * instead of using a parameter object.
7   */
8  public class AccountWithoutParameterObject {
9      /**
10      * Adds the specified amount of the specified currency to
11      * the balance.
12      *
13      * @param currency The currency of the deposited amount.
14      * @param amount   The amount to deposit.
15      */
16     public void deposit(String currency, int amount) {
17     }
18 }
```

Listing 6.12 The `deposit` method takes primitive parameters instead of using a parameter object.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * This class represents a bank account. The parameters of the
5   * <code>deposit</code> method are encapsulated in an object.
6   */
7  public class AccountWithParameterObject {
8      /**
9      * Adds the specified amount of the specified currency to
10     * the balance.
11     *
12     * @param amount   The amount to deposit.
13     */
14    public void deposit(Amount amount) {
15    }
16 }
```

```
17
18 package se.kth.ict.oodbook.prog.smell;
19
20 /**
21  * Represents an amount
22  */
23 class Amount {
24     private String currency;
25     private int amount;
26
27     //Methods
28 }
```

Listing 6.13 The `Amount` class has been created and encapsulates the parameters of the `deposit` method.

In some cases, there are parameters that are simply not needed, because the called method itself can find the data by making a request to an object it already knows. This refactoring is called *Replace Parameter With Method*, and is illustrated in listing 6.14, which has the unnecessary parameter (lines 19 and 39), and listing 6.15, where the called method gets the data instead of using a parameter (lines 18 and 39).

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * The bank application's controller. The call to
5   * <code>withdraw</code> passes the <code>fee</code> parameter
6   * that is not needed.
7   */
8  public class ControllerPassingExtraParameter {
9      private AccountWithExtraParameter account;
10     private AccountCatalog accts;
11
12     /**
13      * Withdraws the specified amount.
14      *
15      * @param amount The amount to withdraw.
16      */
17     public void withdraw(Amount amount) {
18         Amount fee = accts.getWithDrawalFeeOfAccount(account);
19         account.withdraw(amount, fee);
20     }
21 }
22
23 package se.kth.ict.oodbook.prog.smell;
24
25 /**
```

```
26  * Represents a bank account. The method <code>withdraw</code>
27  * takes the <code>fee</code> parameter that is not needed.
28  */
29 public class AccountWithExtraParameter {
30     // Needed for some unknown purpose.
31     private AccountCatalog acctSpecs;
32
33     /**
34      * Withdraws the specified amount.
35      *
36      * @param amount The amount to withdraw.
37      * @param fee    The withdrawal cost.
38      */
39     public void withdraw(Amount amount, Amount fee) {
40     }
41 }
```

Listing 6.14 The call to `withdraw` passes the `fee` parameter that is not needed.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * The bank application's controller. The call to
5   * <code>withdraw</code> does not pass the
6   * <code>fee</code> parameter.
7   */
8  public class ControllerNotPassingExtraParameter {
9      private AccountWithoutExtraParameter account;
10     private AccountCatalog accts;
11
12     /**
13      * Withdraws the specified amount.
14      *
15      * @param amount The amount to withdraw.
16      */
17     public void withdraw(Amount amount) {
18         account.withdraw(amount);
19     }
20 }
21
22 package se.kth.ict.oodbook.prog.smell;
23
24 /**
25  * Represents a bank account. The <code>fee</code> parameter
26  * is not passed to <code>withdraw</code>, since it can be
27  * retrieved in that method itself.
```

```
28  */
29 public class AccountWithoutExtraParameter {
30     // Needed for some unknown purpose, besides getting the
31     // withdrawal fee.
32     private AccountCatalog acctSpecs;
33
34     /**
35      * Withdraws the specified amount.
36      *
37      * @param amount The amount to withdraw.
38      */
39     public void withdraw(Amount amount) {
40         acctSpecs.getWithDrawalFeeOfAccount(this);
41     }
42 }
```

Listing 6.15 The call to `withdraw` does not pass the fee parameter, since it is retrieved by the `withdraw` method.

Excessive Use of Primitive Variables

This code smell is called *Primitive Obsession* in [FOW]. Many advantages of using objects instead of primitive data have already been mentioned. Though primitive data is not forbidden, it should be used with care. Excessive primitive data means everything related to object oriented development is just thrown in the waste bin. There is no encapsulation at all, no cohesion, no possibility to minimize coupling, etc.

A long list of fields in a class is a sign that there are too few classes in the program. The class with the many fields probably has low cohesion, and some of the fields fit better together, in a new class. This refactoring, *Extract Class*, is illustrated in listings 6.16, where there are many fields on lines 8-13, and 6.17, where some of the fields are encapsulated in a new class.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * Represents a person. This class has excessive primitive
5   *  data, since it has fields that fit better as an object.
6   */
7  public class PersonManyFields {
8      private String name;
9      private String street;
10     private int zip;
11     private String city;
12     private String phone;
13     private String email;
14
15     // More code in the class.
```

```
16 }
```

Listing 6.16 This class has excessive primitive data, since it has fields that fit better as an object.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * Represents a person. This class uses objects for the fields.
5   */
6  public class PersonFewerFields {
7      private String name;
8      private Address address;
9      private String phone;
10     private String email;
11
12     // More code in the class.
13 }
14
15 package se.kth.ict.oodbook.prog.smell;
16
17 /**
18  * An address where a person lives.
19  */
20 class Address {
21     private String street;
22     private int zip;
23     private String city;
24
25     // More code in the class.
26 }
```

Listing 6.17 Here, some of the fields have been encapsulated in a class.

It might be that a class has not only fields, but both field(s) and method(s) closer related to each other than to other fields and methods in the class. Also in this case, cohesion can be improved by introducing a new class. The new class shall contain field(s) and method(s) of the original class belonging closely together. The code before the applying the refactoring, listing 6.18, shows the class `Person`, which has the `pnr` field (line 10) holding a person number. The class also has the method `validatePnr` (line 17), that checks if the control digit of the person number is correct. This method really belongs to the `pnr` field, not to the `Person` class. Cohesion is improved in listing 6.19, by introducing the `PersonNumber` class, which will be used to represent person numbers. Note that `validatePnr` (line 30) is called in the constructor of `PersonNumber` (line 26). That way, there can never exist any invalid person numbers in the program, they are immediately revealed when a `PersonNumber` is created.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * Represents a person. This class has low cohesion since the
5   * method <code>validatePnr</code> belongs to the
6   * <code>pnr</code> field, rather than to this class.
7   */
8  public class PersonWithoutPnrClass {
9      private String name;
10     private String pnr;
11
12     public PersonWithoutPnrClass(String name, String pnr) {
13         this.name = name;
14         this.pnr = pnr;
15     }
16
17     private void validatePnr(String pnr) {
18     }
19 }
```

Listing 6.18 This class has low cohesion since the method `validatePnr` belongs to the `pnr` field, rather than to this class.

```
1  package se.kth.ict.oodbook.prog.smell;
2
3  /**
4   * Represents a person. The method <code>validatePnr</code>
5   * has been moved to <code>PersonNumber</code>.
6   */
7  public class PersonWithPnrClass {
8      private String name;
9      private PersonNumber pnr;
10
11     public PersonWithPnrClass(String name, PersonNumber pnr) {
12         this.name = name;
13         this.pnr = pnr;
14     }
15 }
16
17 package se.kth.ict.oodbook.prog.smell;
18
19 /**
20  * Represents a person number.
21  */
22 public class PersonNumber {
23     private String pnr;
```

```
24
25      public PersonNumber(String pnr) {
26          validatePnr(pnr);
27          this.pnr = pnr;
28      }
29
30      private void validatePnr(String pnr) {
31      }
32  }
```

Listing 6.19 The method `validatePnr` has been moved to `PersonNumber`.

A far too common mistake is to use an array of primitive data instead of an object. When arrays are used correctly, all elements in the same array represent the same thing. It is not correct if different elements in an array have different meaning, as is the case with the `stats` array in listing 6.20. In this code, array indexes are used instead of variable names. There is absolutely nothing showing that the first element is the name of a football team, the second the number of wins, the third the number of draws and the fourth the number of losses. This information exists only in the mind of the developer, and it is of course easy to confuse the meaning of the array positions. The code has been improved in listing 6.21, where an object is used instead of the array. The meaning of each value is now clear from the names of the fields in the object.

```
1  String[] stats = new String[3];
2  stats[0] = "Hammarby";
3  stats[1] = "2";
4  stats[2] = "1";
5  stats[3] = "2";
6  printStatistics(stats);
```

Listing 6.20 This code smells, because of the array `stats` where different positions have different meaning.

```
1  Stats stats = new Stats("Hammarby", 2, 1, 2);
2  printStatistics(stats);
3
4  public class Stats {
5      private String name;
6      private int wins;
7      private int draws;
8      private int losses;
9
10     public Stats(String name, int wins, int draws,
11                  int losses) {
12         this.name = name;
13         this.wins = wins;
```

```
14          this.draws = draws;
15          this.losses = losses;
16      }
17  }
```

Listing 6.21 This code encapsulates the values in an object.

Many programming languages, including Java, has enumerations. This enables defining a custom type and the possible values of that type. As an example, consider listing 6.22 that does *not* use an enumerator. Instead, the possible results of the call to `connect` are strings. With such code, nasty bugs can appear because of misspelling the result string. An equally bad alternative is to use integers for the outcomes of `connect`. In this case, bugs might appear because of confusing which number means what. A much better alternative is to introduce a new type for the outcomes, using an enumeration. This is illustrated in listing 6.23. The new type, `ResultCode`, can take the values `SUCCESS`, `PENDING` and `FAILURE`. The meaning of each outcome is obvious from the value and misspelled values will generate a compiler error.

```
1  String result = connect();
2
3  if (result.equals("SUCCESS")) {
4      // Handle established connection.
5  } else if (result.equals("PENDING")) {
6      // Handle pending connection.
7  } else if (result.equals("FAILURE")) {
8      // Handle connection failure.
9  } else {
10     // Something went wrong.
11 }
```

Listing 6.22 Using strings to represent constants.

```
1  Outcome result = connect();
2
3  if (result == Outcome.SUCCESS) {
4      // Handle established connection.
5  } else if (result == Outcome.PENDING) {
6      // Handle pending connection.
7  } else if (result == Outcome.FAILURE) {
8      // Handle connection failure.
9  } else {
10     // Something went wrong.
11 }
12
13 /**
14  * Defines possible outcomes of a connection attempt.
15  */
```

```
16 public enum Outcome {
17     SUCCESS, PENDING, FAILURE
18 }
```

Listing 6.23 Using an enum to represent constants.

Meaningless Names

This is not mentioned as a particular smell in [FOW], but should still be avoided at all costs. Everything that is declared in a program (packages, classes, interfaces, methods, fields, parameters, local variables, etc) must have a meaningful name. This can not be stressed enough. The following list provides a few naming guidelines.

- Do not use one-letter identifiers like `Person p = new Person();`, instead write `Person person = Person();`. There are two exceptions to this guideline. The first is when the full name of the abstraction is just one letter long, it is for example appropriate to use the identifier `x` for an x coordinate. The second exception is that a one letter identifier is accepted for a loop variable, which is normally named `i`. Nested loops are named using following letters in alphabetical order, `j`, `k`, etc.
- Do not name variable `tmp` or `temp` just because it is used temporarily. For example, do not swap two values as in listing 6.24, instead name variables as in listing 6.25.
- Never distinguish similar identifiers by appending numbers. As an example, when transferring money between two bank accounts, they could be named `fromAccount` and `toAccount`, but not `account1` and `account2`.
- Do not be afraid of long names, what matters is that the identifier correctly explain the purpose of what is named. Say for example that some reward is given to the first customer buying a particular item in some campaign in a shop. An adequate name for a variable holding that customer could be `firstCustomerBuyingCampaignItem`.

```
1 int tmp = varsToSwap[0];
2 varsToSwap[0] = varsToSwap[1];
3 varsToSwap[1] = tmp;
```

Listing 6.24 A temporarily used variable is erroneously named `tmp`.

```
1 int valAtIndexZero = varsToSwap[0];
2 varsToSwap[0] = varsToSwap[1];
3 varsToSwap[1] = valAtIndexZero;
```

Listing 6.25 The temporary variable has, correctly, a name describing its purpose.

Anonymous Values

This is not mentioned as a particular smell in [FOW], but shall still be avoided at all costs. Each value in a program shall have an explaining name, since it can be very hard to understand the purpose of a value without any explanation. Never introduce a value in a statement without naming it first, even if that statement is the only place the value is used. Naming the value is a better practice than writing a comment to explain it, since a name is part of the program, the compiler helps to ensure the name is used correctly. A comment, on the other hand, is a kind of duplicated information that exists besides the program. There is always the risk that comments are not maintained when the program changes. Below follows examples of how values can be given explaining names.

- Say that the method `connect(int timeout)` tries to connect for `timeout` number of milliseconds before stopping. Say also that we want to make it try for ten seconds. A straightforward way to write this could be `connect(10000)`, but then the reader gets no information about the purpose of the value 10000. A better way is to write `connect(10 * MILLIS_PER_SECOND)`, and define the constant `private static final int MILLIS_PER_SECOND = 1000`. Still, however, the purpose of the value 10 might not be clear. The best way to clarify it is to also define a constant `private static final int CONNECT_TIMEOUT_SECS = 10`, or, if it is not a constant, the variable `int connectTimeoutSecs = 10`. Now, the code becomes `connect(CONNECT_TIMEOUT_SECS * MILLIS_PER_SECOND)` or `connect(connectTimeoutSecs * MILLIS_PER_SECOND)`.

- The practice of naming values applies (at least) to all primitive types, and also to strings, since a string can be written as a primitive value, without using the keyword `new`. Consider for example opening a file, whose name is in the variable `fileName`, located in the directory whose name is in the variable `dirName`. Assuming that there is a method `openFile`, that opens a file, this might be done with the statement `openFile(dirName + "\" + fileName)`. The meaning of the value `"\"` might seem clear, still, it is even clearer to introduce the constant `private static final String PATH_SEPARATOR = "\"`, and write `openFile(dirName + PATH_SEPARATOR + fileName)`. Using this constant everywhere a path separator is needed also gives the advantage that it is easy to change path separator, if running on a system where the path separator is not a backslash. Using the constant, only one line has to be changed in this situation, namely the declaration of the constant.

- Sometimes it is best to use a method to name a value. This is often the case when naming values that occur in `if` statements. As an example, consider an `if` statement checking for end of line (EOL) in a string. EOL in a unix file is represented by a character with ASCII code 10, therefore, the code in listing 6.26 might be used. This code is unclear, why check for the value 10? A better solution is to introduce the method `isUnixEol`, and use the code in figure 6.27.

```
1  /**
2   * Finds the index of the first Unix EOL in the specified
3   * string.
4   *
5   * @param source The string in which to look for EOL.
6   * @return The index of the first EOL, or -1 if there was
7   *         no EOL in the specified string.
8   */
9  public int findIndexOfFirstEolWorse(String source) {
10     char[] sourceChars = source.toCharArray();
11     for (int i = 0; i < sourceChars.length; i++) {
12         if (sourceChars[i] == 10) {
13             return i;
14         }
15     }
16     return -1;
17 }
```

Listing 6.26 It is quite difficult to understand the meaning of the anonymous value 10 on line 12

```
1  private boolean isUnixEol(char character) {
2      return character == 10;
3  }
4
5  /**
6   * Finds the index of the first Unix EOL in the specified
7   * string.
8   *
9   * @param source The string in which to look for EOL.
10  * @return The index of the first EOL, or -1 if there was
11  *         no EOL in the specified string.
12  */
13 public int findIndexOfFirstEolBetter(String source) {
14     char[] sourceChars = source.toCharArray();
15     for (int i = 0; i < sourceChars.length; i++) {
16         if (isUnixEol(sourceChars[i])) {
17             return i;
18         }
19     }
20     return -1;
21 }
```

Listing 6.27 The purpose of the value 10 on line 2 is explained by the name of the method, `isUnixEol`

Complicated Flow Control

There are many different refactorings related to `if` statements and loops in [FOW], this section is loosely related to some of those. The problem is that both `if` statements and loops can be made very complicated and hard to understand, if written without thought. That happens mainly for two reasons. The first is to nest too many such statements, giving too much indentation, which makes it hard to understand which level of indentation depends on what. The second reason is to introduce unnecessary flags, thereby requiring the reader to keep in mind which flag depends on what. Listing 6.28 shows a small example of both those mistakes. It's a method that checks if a certain character is present in a specified string. Note by the way that it's written just to illustrate this kind om code smell, in reality there's no point in writing such a method, since it already exists in `java.lang.String`.

```
1  /**
2       * Tells if the specified string contains the searched char.
3       *
4       * @param src      The string in which to look for the
5       *                 searched character.
6       * @param searched The searched character.
7       * @return         {@code true} if the specified string
8       *                 contains the searched character,
9       *                 {@code false} if it does not.
10      */
11 public boolean containsChar(String src, char searched) {
12     boolean found = false;
13     if (src != null) {
14         char[] srcAsCharArray = src.toCharArray();
15         for (int i = 0; i < srcAsCharArray.length; i++) {
16             if (srcAsCharArray[i] == searched) {
17                 found = true;
18             }
19         }
20     }
21     return found;
22 }
```

Listing 6.28 An example of unnecessarily complicated code, both the if statement and the loop can be improved

Listing 6.28 has unnecessary indentation because the `if` statement on line 14 wraps the entire method body. This makes the code harder to read since that condition must be remembered throughout the entire method. This problem can be solved by removing checks for valid parameters, and instead check for *invalid* parameters, as is done in listing 6.29, where there's one indentation level less. The trick is to start the method by checking for invalid parameters, and immediately handle those. Then, when the actual work starts, it's sure that all parameters are valid, and the code only has to handle successful work on valid parameters.

```
1  /**
2      * Tells if the specified string contains the searched char.
3      *
4      * @param src      The string in which to look for the
5      *                 searched character.
6      * @param searched The searched character.
7      * @return         {@code true} if the specified string
8      *                 contains the searched character,
9      *                 {@code false} if it does not.
10     */
11 public boolean findChar(String src, char searched) {
12     if (src == null) {
13         return false;
14     }
15     boolean found = false;
16     char[] srcAsCharArray = src.toCharArray();
17     for (int i = 0; i < srcAsCharArray.length; i++) {
18         if (srcAsCharArray[i] == searched) {
19             found = true;
20         }
21     }
22     return found;
23 }
```

Listing 6.29 One level of indentation has been removed compared to figure 6.28, by changing the if statement on line 14

Both listings 6.28 and 6.29 contains the `found` flag that's not needed, and only serves to complicate the code. Besides, they're also ineffective, since they make unnecessary iterations in the loop. The fact is that, on line 20 in listing 6.29, we already know that the string contains the searched character. Thereby the result of the method is found, and no more work is needed. This means the method doesn't have to set the flag and return after the loop, but can instead return immediately, as is done in listing 6.30.

```
1  /**
2      * Tells if the specified string contains the searched char.
3      *
4      * @param src      The string in which to look for the
5      *                 searched character.
6      * @param searched The searched character.
7      * @return         {@code true} if the specified string
8      *                 contains the searched character,
9      *                 {@code false} if it does not.
10     */
11 public boolean containsChar(String src, char searched) {
12     if (src == null) {
```

```
13          return false;
14      }
15      char[] srcAsCharArray = src.toCharArray();
16      for (int i = 0; i < srcAsCharArray.length; i++) {
17          if (srcAsCharArray[i] == searched) {
18              return true;
19          }
20      }
21      return false;
22  }
```

Listing 6.30 The flag has been removed, and the method instead returns immediately when the searched character is found

6.5 Coding Case Study

Now, it is finally time to write the `RentCar` program. This section does not include a complete listing of the entire program. That can be found in the accompanying NetBeans project, which can be downloaded from GitHub [Code]. Here follows a description of the first parts of the code, and of parts where particular afterthought was needed, or where the design made in chapter 5 was not followed.

Even though this is a quite small program, the design is still big enough to make it difficult to decide where to start coding. This is a result of having designed the entire requirements specification in one go, without any intermediate coding. Normally, each system operation would have been coded as soon as the design was finished, since the only way to get a full understanding of a design's strengths and weaknesses is to implement it in code. Also here, however, where all the design has been made up front, it is still best to code one system operation at a time, in the order they are executed. That way it is possible to test run each part of the program as soon as it is written.

The `searchMatchingCar` system operation

The first system operation is `searchMatchingCar`, and the final design is in figure 5.27. The final version of the start sequence, however, is not in figure 5.28. It was changed in figure 5.35, where the class `RegistryCreator` was added. Most of the coding is a very straightforward implementation of those diagrams, listed below, but three things require extra attention. First, nothing has been decided on the implementation of the car registry. If the program should ever be finalized for real use, the registry classes in the integration layer would call real databases, but here, where there is no database, where are the cars stored? The solution is to place a list with some hard coded cars directly in the `CarRegistry` class, see lines 11, 14 and 37-44 in figure 6.34. This is enough for testing purposes. Second, neither requirements specification, nor design, are very specific on the search algorithm to use when looking for a matching car. To which extent must the specified search criteria be met to consider a car to match them? Also, can some of the car's properties be left unspecified? This should of course be discussed

with the customer. The chosen implementation, se lines 48-68 in figure 6.33, requires exact match of all specified parameters, but gives the possibility to leave all parameters except AC and four wheel drive unspecified. Third, and last, what shall happen when the program is executed? Since there is no view, to get some output and be able to verify the functionality, the method `sampleExecution` is added to `View`, as can be seen on lines 28-45 in figure 6.31. This method contains hard coded calls to all system operations and prints the result of those calls. Also, a `toString` method has been added to `CarDTO` to provide an informative printout of objects of that class, see lines 70-80 in figure 6.33.

```
1  package se.kth.ict.rentcar.view;
2
3  import se.kth.ict.rentcar.controller.Controller;
4  import se.kth.ict.rentcar.integration.CarDTO;
5
6  /**
7   * This program has no view, instead, this class is a
8   * placeholder for the entire view.
9   */
10
11 public class View {
12     private Controller contr;
13
14     /**
15      * Creates a new instance.
16      *
17      * @param contr The controller that is used for all
18      *              operations.
19      */
20     public View(Controller contr) {
21         this.contr = contr;
22     }
23
24     /**
25      * Simulates a user input that generates calls to all
26      * system operations.
27      */
28     public void sampleExecution() {
29         CarDTO unavailableCar =
30             new CarDTO(1000, "nonExistingSize", true, true,
31                        "red", null);
32         CarDTO availableCar =
33             new CarDTO(1000, "medium", true, true, "red",
34                        null);
35
36         CarDTO foundCar =
37             contr.searchMatchingCar(unavailableCar);
```

```
38          System.out.println(
39              "Result of searching for unavailable car: " +
40              foundCar);
41          foundCar = contr.searchMatchingCar(availableCar);
42          System.out.println(
43              "Result of searching for available car: " +
44              foundCar);
45      }
46  }
```

Listing 6.31 The class `View` when only the `searchMatchingCar` system operation has been implemented.

```
1  package se.kth.ict.rentcar.controller;
2
3  import se.kth.ict.rentcar.integration.CarRegistry;
4  import se.kth.ict.rentcar.integration.CarDTO;
5  import se.kth.ict.rentcar.integration.RegistryCreator;
6
7  /**
8   * This is the application's only controller class. All
9   * calls to the model pass through here.
10  */
11
12 public class Controller {
13     private CarRegistry carRegistry;
14
15     /**
16      * Creates a new instance.
17      *
18      * @param regCreator Used to get all classes that
19      *                   handle database calls.
20      */
21     public Controller(RegistryCreator regCreator) {
22         this.carRegistry = regCreator.getCarRegistry();
23     }
24
25     /**
26      * Search for a car matching the specified search criteria.
27      *
28      * @param searchedCar This object contains the search
29      *                    criteria. Fields in the object that
30      *                    are set to <code>null</code> or
31      *                    <code>0</code> are ignored.
32      * @return The best match of the search criteria.
33      */
```

```
34      public CarDTO searchMatchingCar(CarDTO searchedCar) {
35          return carRegistry.findCar(searchedCar);
36      }
37  }
```

Listing 6.32 The class `Controller` when only the `searchMatchingCar` system operation has been implemented.

```
1   package se.kth.ict.rentcar.integration;
2   /**
3    * Contains information about one particular car.
4    */
5   public final class CarDTO {
6       private final int price;
7       private final String size;
8       private final boolean AC;
9       private final boolean fourWD;
10      private final String color;
11      private final String regNo;
12
13      /**
14       * Creates a new instance representing a particular car.
15       *
16       * @param price  The price paid to rent the car.
17       * @param size   The size of the car, e.g.,
18       *               <code>medium</code>.
19       * @param AC     <code>true</code> if the car has air
20       *               condition.
21       * @param fourWD <code>true</code> if the car has four
22       *               wheel drive.
23       * @param color  The color of the car.
24       * @param regNo  The car's registration number.
25       */
26      public CarDTO(int price, String size, boolean AC,
27                    boolean fourWD, String color, String regNo) {
28          this.price = price;
29          this.size = size;
30          this.AC = AC;
31          this.fourWD = fourWD;
32          this.color = color;
33          this.regNo = regNo;
34      }
35
36      /**
37       * Checks if the specified car has the same features as
38       * this car. Fields that are set to <code>null</code> or
```

```
39          * <code>0</code> are ignored. Note that the check is
40          * for matching features, not for identity. Therefore,
41          * registration number is ignored.
42          *
43          * @param searched Contains search criteria.
44          * @return <code>true</code> if this object has the same
45          *         features as <code>searched</code>,
46          *         <code>false</code> if it has not.
47          */
48         boolean matches(CarDTO searched) {
49             if (searched.getPrice() != 0 &&
50                 searched.getPrice() != price) {
51                 return false;
52             }
53             if (searched.getSize() != null &&
54                 !searched.getSize().equals(size)) {
55                 return false;
56             }
57             if (searched.getColor() != null &&
58                 !searched.getColor().equals(color)) {
59                 return false;
60             }
61             if (searched.isAC() != AC) {
62                 return false;
63             }
64             if (searched.isFourWD() != fourWD) {
65                 return false;
66             }
67             return true;
68         }
69
70         @Override
71         public String toString() {
72             StringBuilder builder = new StringBuilder();
73             builder.append("regNo: " + regNo + ", ");
74             builder.append("size: " + size + ", ");
75             builder.append("price: " + price + ", ");
76             builder.append("AC: " + AC + ", ");
77             builder.append("4WD: " + fourWD + ", ");
78             builder.append("color: " + color);
79             return builder.toString();
80         }
81
82         // Getters are not listed.
83
84 }
```

Listing 6.33 The class `CarDTO` when only the `searchMatchingCar` system operation has been implemented.

```
1  package se.kth.ict.rentcar.integration;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  /**
7   * Contains all calls to the data store with cars that may be
8   * rented.
9   */
10 public class CarRegistry {
11     private List<CarDTO> cars = new ArrayList<>();
12     CarRegistry() {
13         addCars();
14     }
15
16     /**
17      * Search for a car matching the specified search criteria.
18      *
19      * @param searchedCar This object contains the search
20      *                    criteria. Fields in the object that
21                           are set to <code>null</code> or
22      *                    <code>0</code> are ignored.
23      * @return <code>true</code> if a car with the same
24      *         features as <code>searchedCar</code> was found,
25      *         <code>false</code> if no such car was found.
26      */
27     public CarDTO findCar(CarDTO searchedCar) {
28         for (CarDTO car : cars) {
29             if (car.matches(searchedCar)) {
30                 return car;
31             }
32         }
33         return null;
34     }
35
36     private void addCars() {
37         cars.add(new CarDTO(1000, "medium", true, true, "red",
38                             "abc123"));
39         cars.add(new CarDTO(2000, "large", false, true, "blue",
40                             "abc124"));
41         cars.add(new CarDTO(500, "medium", false, false, "red",
42                             "abc125"));
```

```
43      }
44  }
```

Listing 6.34 The class `CarRegistry` when only the `searchMatchingCar` system operation has been implemented.

```
1  package se.kth.ict.rentcar.integration;
2  /**
3   * This class is responsible for instantiating all registries.
4   */
5  public class RegistryCreator {
6      private CarRegistry carRegistry = new CarRegistry();
7
8      /**
9       * Get the value of carRegistry
10      *
11      * @return the value of carRegistry
12      */
13     public CarRegistry getCarRegistry() {
14         return carRegistry;
15     }
16 }
```

Listing 6.35 The class `RegistryCreator` when only the `searchMatchingCar` system operation has been implemented.

```
1  package se.kth.ict.rentcar.startup;
2
3  import se.kth.ict.rentcar.controller.Controller;
4  import se.kth.ict.rentcar.integration.RegistryCreator;
5  import se.kth.ict.rentcar.view.View;
6
7  /**
8   * Contains the <code>main</code> method. Performs all startup
9   * of the application.
10  */
11 public class Main {
12     /**
13      * Starts the application.
14      *
15      * @param args The application does not take any command
16      *             line parameters.
17      */
18     public static void main(String[] args) {
19         RegistryCreator creator = new RegistryCreator();
20         Controller contr = new Controller(creator);
21         new View(contr).sampleExecution();
```

```
22      }
23  }
```

Listing 6.36 The class `Main` when only the `searchMatchingCar` system operation has been implemented.

The `registerCustomer` system operation

The code for the `registerCustomer` system operation is a very straightforward implementation of figure 5.31. There is only one thing that requires a bit of consideration, namely what to do with the `CustomerDTO` passed to the `Rental` constructor. For now, there is no reason to do anything at all, except to save it in a field in the constructed `Rental` object. That is safe to do since the DTO is immutable, and there is therefore no risk that any other object changes its content. Remember that being *immutable* means an object can not change state, for example because the class itself and all its fields are final. If `CustomerDTO` had not been final, it would have been suicide to just keep it in `Rental`. In that case, the object that sent it to `Rental` could have kept a reference to the same DTO object, and later updated it.

There is also another issue with keeping the DTO in `Rental`. Is it really sure it is just a DTO, having no logic at all? If, for example, there is the need to validate the driving license number, or to calculate a person's age based on the driving license number, the methods performing this would, with the argument of cohesion, be placed in `DrivingLicenseDTO` or `CustomerDTO`. This would turn that object into an entity object, with business logic, instead of a DTO. This change is not just a matter of renaming, e.g., from `CustomerDTO` to `Customer`, but also concerns how the object is handled. A DTO may be used in the view, but an entity object may not. To conclude this discussion, it is clear that the objects named DTO are, at least for the moment, DTOs, but we must be aware that this might change in the future. `Rental` is listed in listing 6.37, to illustrate the reasoning above. The rest of the `registerCustomer` implementation can be found in the accompanying NetBeans project [Code].

```
1  package se.kth.ict.rentcar.model;
2
3  /**
4   * Represents one particular rental transaction, where one
5   * particular car is rented by one particular customer.
6   */
7  public class Rental {
8      private CustomerDTO customer;
9
10     /**
11      * Creates a new instance, representing a rental made by
12      * the specified customer.
13      *
14      * @param customer The renting customer.
15      */
16     public Rental(CustomerDTO customer) {
```

```
17          this.customer = customer;
18      }
19  }
```

Listing 6.37 The class `Rental`, after implementing the `registerCustomer` system operation.

The `bookCar` system operation

The `bookCar` design can be found in figures 5.33 and 5.35. Implementing this system operation reveals one serious problem, the implementation of the `setBookedStateOfCar` method in `CarRegistry`. The cars in the registry are currently stored in a list of `CarDTO`s. How to mark that one of them is booked, and not available for rental? This problem reveals a flaw in the implementation of `CarRegistry`. That class is supposed to call a database or some other system that stores car data persistently. Such a datastore does not hold a list of immutable DTOs, but instead raw, mutable data. This data shall not be object-oriented, since it mimics a store with primitive data. Instead of having methods, objects shall have only primitive variables. Therefore, the list in `CarRegistry` is changed, to hold objects of a class `CarData`, which has just primitive fields, no methods at all. This class shall not be used anywhere outside `CarRegistry`, since it mimics the contents of the `CarRegistry` datastore. To ensure it is not used anywhere else, it is a private inner class, see lines 97-117 in listing 6.38.

```
1  package se.kth.ict.rentcar.integration;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  /**
7   * Contains all calls to the data store with cars that may be
8   * rented.
9   */
10 public class CarRegistry {
11     private List<CarData> cars = new ArrayList<>();
12
13     CarRegistry() {
14         addCars();
15     }
16
17     /**
18      * Search for a car matching the specified search criteria.
19      *
20      * @param searchedCar This object contains the search
21      *                    criteria. Fields in the object
22      *                    that are set to <code>null</code>
23      *                    or <code>0</code> are ignored.
```

```
24          * @return <code>true</code> if a car with the same
25          *         features as <code>searchedCar</code> was found,
26          *         <code>false</code> if no such car was found.
27          */
28         public CarDTO findAvailableCar(CarDTO searchedCar) {
29             for (CarData car : cars) {
30                 if (matches(car, searchedCar) && !car.booked) {
31                     return new CarDTO(car.regNo, car.price,
32                                       car.size, car.AC,
33                                       car.fourWD, car.color);
34                 }
35             }
36             return null;
37         }
38
39         /**
40          * If there is an existing car with the registration
41          * number of the specified car, set its booked
42          * property to the specified value. Nothing is changed
43          * if the car's booked property already had the specified
44          * value.
45          *
46          * @param car         The car that shall be marked as
47          *                    booked.
48          * @param bookedState The new value of the booked property.
49          */
50         public void setBookedStateOfCar(CarDTO car,
51                                         boolean bookedState) {
52             CarData carToBook = findCarByRegNo(car);
53             carToBook.booked = bookedState;
54         }
55
56         private void addCars() {
57             cars.add(new CarData("abc123", 1000, "medium", true,
58                                  true, "red"));
59             cars.add(new CarData("abc124", 2000, "large", false,
60                                  true, "blue"));
61             cars.add(new CarData("abc125", 500, "medium", false,
62                                  false, "red"));
63         }
64
65         private boolean matches(CarData found, CarDTO searched) {
66             if (searched.getPrice() != 0 &&
67                 searched.getPrice() != found.price) {
68                 return false;
69             }
```

```
70          if (searched.getSize() != null &&
71              !searched.getSize().equals(found.size)) {
72              return false;
73          }
74          if (searched.getColor() != null &&
75              !searched.getColor().equals(
76                  found.color)) {
77              return false;
78          }
79          if (searched.isAC() != found.AC) {
80              return false;
81          }
82          if (searched.isFourWD() != found.fourWD) {
83              return false;
84          }
85          return true;
86      }
87
88      private CarData findCarByRegNo(CarDTO searchedCar) {
89          for (CarData car : cars) {
90              if (car.regNo.equals(searchedCar.getRegNo())) {
91                  return car;
92              }
93          }
94          return null;
95      }
96
97      private static class CarData {
98          private String regNo;
99          private int price;
100         private String size;
101         private boolean AC;
102         private boolean fourWD;
103         private String color;
104         private boolean booked;
105
106         public CarData(String regNo, int price, String size,
107                        boolean AC, boolean fourWD,
108                        String color) {
109             this.regNo = regNo;
110             this.price = price;
111             this.size = size;
112             this.AC = AC;
113             this.fourWD = fourWD;
114             this.color = color;
115             this.booked = false;
```

```
116             }
117         }
118 }
```

Listing 6.38 The class `CarRegistry`, after implementing the `bookCar` system operation.

With the above change to `CarRegistry`, it becomes necessary to change the `CarDTO` method `matches`, which compares the features the customer wishes with the features of an available car. It must now compare fields in a `CarDTO` with fields in a `CarData`, and the latter must not be used outside `CarRegistry`. This is solved by removing `matches` from `CarDTO` and instead making it a private method in `CarRegistry`, see lines 65-86 in listing 6.38. This is in fact a better location for `matches`. First, the total public interface decreases since a public method becomes private. Second, it was never a very good idea to have such a method in a DTO. A DTO shall not have any business logic, and `matches` can be regarded as business logic, it performs a matching algorithm and is not just a simple data comparison.

Now that it is possible to mark a car as booked, it also becomes necessary to prevent a booked car from being rented by other customers. Therefore, the `findCar` method must not return a booked car, even if it matches the specified search criteria. This results in the `if`-statement on line 30 in listing 6.38. To clarify this new behaviour, the method name is changed from `findCar` to `findAvailableCar`.

Another refactoring was to change the order of the parameters in the `CarDTO` constructor, `regNo` is now the first parameter. This was done because every time that constructor was called, the first thought was to place the registration number first. This is a clear sign that having `regNo` first is a more logical ordering of the parameters. Also the `Rental` constructor had to be changed, according to figure 5.36, to include a reference to `CarRegistry`. This is needed since `Rental` will call the method `setBookedState OfCar` in `CarRegistry`. That concludes the implementation of the `bookCar` system operation, the rest of the code can be downloaded in the accompanying NetBeans project [Code]. Note, however, that there is one unhandled issue left, what happens if the car that shall be booked is already booked? This situation is not handled yet, but should be addressed before the program is completed.

The `pay` system operation

The final system operation, `pay`, is designed in figures 5.40 and 5.42. Here, there are two questions that were left unanswered during design. The first is how the receipt text is created, the design only shows that a `Receipt` object is created and passed to the printer. The chosen solution is to add a method `createReceiptString` to `Receipt`, see lines 26-47 in listing 6.39. The printer will call this method to get a a formatted string with the entire receipt text. This string is then printed to `System.out`, since there is no real printer in this program. To create this receipt string, `Receipt` must gather information from `Rental`, which leads to the question what information `Rental` shall reveal. It has three objects with data, a customer object, a car object and a payment object. Either we create methods that hand out these objects, like `getRentedCar`, or we create methods that hand out the data in the objects, like `getRegNoOfRentedCar`. The former alternative, to hand out whole objects, is normally to

prefer. That way objects are kept intact, instead of breaking them up and passing primitive data. By handing out the whole object, the receiver can call any method in the received object, not just use its data. This alternative is chosen here, which means, for example, that `Receipt` calls `rental.getPayment().getTotalCost()` to get the cost of the rental.

The other unresolved issue left from design is the method `calculateTotalCost` in `CashPayment`. Actually, we have no idea how a total cost is calculated. A very simple solution is chosen, but would certainly have to be improved if development were to continue to a complete car rental system. This simple solution is to just use the price in `CarDTO` as total cost, which means mileage and insurance costs are ignored, and also that there can be no discounts or campaigns. To implement this solution, `CashPayment` uses `paidRental.getRentedCar().getPrice()` as total cost, line 29 in listing 6.40. This might seem a strange solution. Why shall `Rental` pass itself to `CashPayment`, which then only calls back to `Rental` to get the cost, lines 28-29 in listing 6.40. Why not just let `Rental` pass the cost, instead of itself, to `CashPayment`? The reason is it is assumed that, as the program grows, `CashPayment` will have to gather more information, like driven distance and possible discounts. It is `CashPayment` that shall know what data is needed to calculate the cost, from where to get that data, and how to use it. That is why cost calculation is handed over from `Rental` to `CashPayment`.

```
1  package se.leiflindback.oodbook.rentcar.model;
2
3  import java.time.LocalDateTime;
4
5  /**
6   * The receipt of a rental
7   */
8  public class Receipt {
9      private final Rental rental;
10
11     /**
12      * Creates a new instance.
13      *
14      * @param rental The rental proved by this receipt.
15      */
16     Receipt(Rental rental) {
17         this.rental = rental;
18     }
19
20     /**
21      * Creates a well-formatted string with the entire content
22      * of the receipt.
23      *
24      * @return The well-formatted receipt string.
25      */
26     public String createReceiptString() {
27         StringBuilder builder = new StringBuilder();
```

```
28          appendLine(builder, "Car Rental");
29          endSection(builder);
30
31          LocalDateTime rentalTime = LocalDateTime.now();
32          builder.append("Rental time: ");
33          appendLine(builder, rentalTime.toString());
34          endSection(builder);
35
36          builder.append("Rented car: ");
37          appendLine(builder, rental.getRentedCar().getRegNo());
38          builder.append("Cost: ");
39          appendLine(builder, rental.getPayment().
40                                  getTotalCost().toString());
41          builder.append("Change: ");
42          appendLine(builder, rental.getPayment().
43                                  getChange().toString());
44          endSection(builder);
45
46          return builder.toString();
47      }
48
49      private void appendLine(StringBuilder builder,
50                              String line) {
51          builder.append(line);
52          builder.append("\n");
53      }
54
55      private void endSection(StringBuilder builder) {
56          builder.append("\n");
57      }
58  }
```

Listing 6.39 The class `Receipt`, after implementing the `pay` system operation.

```
1  package se.kth.ict.oodbook.rentcar.model;
2
3  /**
4   * Represents one specific payment for one specific rental. The
5   * rental is payed with cash.
6   */
7  public class CashPayment {
8      private Amount paidAmt;
9      private Amount totalCost;
10
11     /**
12      * Creates a new instance. The customer handed over the
```

```
13          * specified amount.
14          *
15          * @param paidAmt The amount of cash that was handed over
16          *                by the customer.
17          */
18         public CashPayment(Amount paidAmt) {
19             this.paidAmt = paidAmt;
20         }
21
22         /**
23          * Calculates the total cost of the specified rental.
24          *
25          * @param paidRental The rental for which the customer is
26          *                   paying.
27          */
28         void calculateTotalCost(Rental paidRental) {
29             totalCost = paidRental.getRentedCar().getPrice();
30         }
31
32         /**
33          * @return The total cost of the rental that was paid.
34          */
35         Amount getTotalCost() {
36             return totalCost;
37         }
38
39         /**
40          * @return The amount of change the customer shall have.
41          */
42         Amount getChange() {
43             return paidAmt.minus(totalCost);
44         }
45 }
```

Listing 6.40 The class `CashPayment`, after implementing the `pay` system operation.

6.6 Common Mistakes

Below follows a list of common coding mistakes.

- **Incomplete comments** Each public declaration (class, method, etc) shall have a javadoc comment. Method comments shall cover parameters and return values, using the javadoc tags `@param` and `@return`. It is often argued that it is unnecessary to comment getter and setter methods. That might very well be the case, but how long does it take to add a one line comment to a getter or setter? It might even be that the IDE can generate the comment. If *every* public declaration has a comment, there is no risk of missing to comment something by mistake, or by pure laziness.
- **Excessive comments** There should be no comments besides the above mentioned javadoc comments. If there is a need for more comments to explain the code, it probably means the code is too complex, and has low cohesion.
- **Comments written too late** Write the comments together with the code that is commented, maybe even before. That way, writing the comment makes it necessary to clarify what the code shall do, before (or immediately after) it is written. Also, if comments are written together with the code, they will be of use in future development of the program. If comments are written last, when the program is already working, commenting is just a burden, and probably quite a heavy burden.

NO!

- Many of the common design mistakes can be introduced when programming, even if they were avoided during design. For example, there is a big risk to use **primitive data instead of objects**, to use **static declarations** when they are not appropriate or to place **input or output outside the view**. See the text on common design mistakes in section 5.6 for more details on this.
- Section 6.4, on code smell and refactorings, covers many things that shall be avoided when coding. Maybe the most common of those possible mistakes are **meaningless names** and **unnamed values**.

Chapter 7

Testing

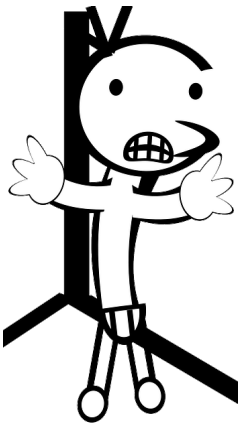


Figure 7.1 Lack of tests will bring fear, uncertainty and doubt, since programmers can not trust the program. Image by unknown creator [Public domain], via https://pixabay.com

How is it possible to know if a program works? The answer to that question makes a very big difference. If it is complicated to verify that the program works as intended, developers will be extremely reluctant to make changes. They will neither be willing to apply refactorings to improve the design of existing code, nor to change existing functionality. Instead they will argue against customer's requirement changes, and solve all problems by adding new code. This is a disastrous state of development, characterized by fear, uncertainty and doubt. Because of the reluctance to work with existing code, developers will have little knowledge about the code and the code will be in bad state. This will make them even more reluctant to make changes, which will lead to even less knowledge and worsen code state even more. This is exactly the opposite of the flexibility we want to achieve. The code will constantly become *less* flexible.

If, on the other hand, it is very easy to verify that the code works as intended, developers will be happy to change it. They will constantly improve its design with refactorings. They will also be glad to improve customer satisfaction by adjusting to changing requirements. This is the flexibility of well-designed software! Code quality constantly improves, developers get better knowledge about the code, and thereby becomes even more willing to change it.

The difference between the two scenarios above is *automated tests*. There should be a test program which gives input to the program under test, and also evaluates the output. If a test passes, the test program does not do anything. If a test

Figure 7.2 Complete tests will bring confidence, since programmers can trust the program. Image by unknown creator [Public domain], via https://pixabay.com

fails, it prints an informative message about the failure. With extensive tests that cover all, or most, possible execution paths through the program with all, or most, possible variable values, it is guaranteed that the program works if all tests pass. This is a *very* good situation, one command starts the test, which tells if the program under test works, or, if not, exactly which problems there are.

7.1 Unit Tests and The JUnit Framework

A *unit test* is a test of the smallest possible piece of code that makes sense on its own, typically a method. Unit tests constitute, by far, the most common way for developers to verify that their code works as intended. Listings 7.1 and 7.2 show a first example of a unit test. The first of those listings contains the *system under test, SUT*. It is a method `equals`, in a class `Amount`. The method shall compare two `Amount` instances and return `true` if they represent the same amount, or `false` if they represent different amounts. Listing 7.2 contain a unit test for that method. It creates two `Amount` instances representing the same amount (lines four and five), calls the `equals` method (line 7) and verifies that the result is as expected (lines eight to ten). This test is written using the JUnit 5 framework. It will be covered in more detail below, when JUnit has been introduced.

```
1 public boolean equals(Object other) {
2     if (!(other instanceof Amount)) {
3         return false;
4     }
5     Amount otherAmount = (Amount) other;
6     return amount == otherAmount.amount;
7 }
```

Listing 7.1 The SUT (System Under Test) is the `equals` method in the class `Amount`

```
1  @Test
2  public void testEqual() {
3      int amount = 3;
4      Amount instance = new Amount(amount);
5      Amount other = new Amount(amount);
6      boolean expResult = true;
7      boolean result = instance.equals(other);
8      assertEquals(expResult, result,
9                   "Amounts with same state are not equal.");
10 }
```

Listing 7.2 A unit test for the code in listing 7.1

Frameworks

There are many frameworks that facilitate unit testing, since it is an extremely common testing approach. JUnit was one of the first, and is also very frequently used. But why use a framework at all? And exactly what is a framework? A framework provides some functionality that is not specific for a particular application, but is needed in different applications. Think of the Java APIs from Oracle, they provide functionality for common tasks, and can be used in many different applications. In contrast to an API, a framework not only provides code, but also flow control. This means the main method is in the framework, not in application code written by application developers. Thus, the framework is responsible for calling application code at the right time, not the opposite. This fact, that the application is *not* responsible for flow control, when to call which method, is very important. Consider for example a framework providing some security control. It would be very hard for application developers to always remember, and never forget, to call the security controlling methods in the framework in all necessary places. Just one miss would introduce a security hole. If, instead, the framework itself has the main method and is responsible for when to handle security, application code will be completely relieved of everything related to security control. This is illustrated in figure 7.3, where the blue piece, representing the application, is placed inside the framework, represented by all the white pieces. The colored lines are different executions through the program. Execution may start in a main method inside the framework, as is the case for the black line. Execution may also enter the framework from the outside, for example via a network call, as is the case for the red and green lines, but execution never starts in the application. When the colored lines enter the application piece, it typically means the framework has called a method in the application. When the lines exit the application piece, that method has returned.

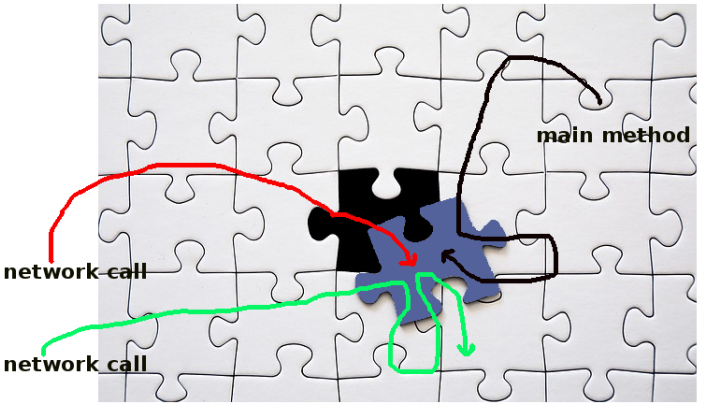


Figure 7.3 The application fits in the framework like a piece in a puzzle. Execution, the colored lines, enter the application via method calls from the framework.

There are many good reasons to use a framework whenever one can be found. First, a framework is thoroughly tested and proven to work well. If it did not work well, it would not be used. Second, if there are many developers using the same framework, there will be lots of documentation, and it will be easy to get help. Third, the fact that the framework is responsible for flow control makes sure all code is executed in correct order. Last, not using a framework means writing new code, which means introducing new bugs.

JUnit

JUnit[JU] is one of the most popular unit testing frameworks for Java. It is based on *annotations*. An annotation is a part of a Java program that is not executed, but instead provides information about the program for the compiler, or for the JVM, or, as is the case here, for a framework (JUnit). An annotation is usually used for properties unrelated to the functionality of the source code, for example to configure security, networking, multithreading or testing. It starts with the at sign, @, for example `@SomeAnnotation`. It may take parameters, for example `@SomeAnnotation(someString = "abc", someBoolean = true)`. When writing tests with JUnit, annotations are used to describe the contents of methods in the test code, for example that a certain method contains a test. Some of the most common JUnit annotations are explained in table 7.1

Annotation Example	Explanation
`@Test` `public void aTest()`	**`aTest`** contains tests and will be executed when tests are run.

`@Disabled("Not implemented")`
`@Test`
`public void aTest()`
	`aTest` will not be executed.

`@BeforeEach` `public void prepareTest()`	**`prepareTest`** is executed before each test method.
`@AfterEach` `public void cleanup()`	**`cleanup`** is executed after each test method.
`@BeforeAll` `public void prepareTests()`	**`prepareTests`** is executed once before the first test in this class.
`@AfterAll` `public void cleanup()`	**`cleanup`** is executed once after the last test in this class.

Table 7.1 Some of the most common JUnit annotations

A fully automated test must not only call the SUT, but also evaluate if the result of the call is the expected, that is, if the test passed or failed. This evaluation is done with assert methods in JUnit. An assert method verifies that its parameters meet some constraint, for example that they are equal. If the constraint is met, the test passes and nothing is printed to the console. If the parameters do not meet the constraint, the test fails and the specified explaining message is printed. Some of the most common assert methods are explained in table 7.2.

Assertion Example	Explanation
`fail("explanation")`	Always fails. Can be placed at a code line that should never be reached.
`assertTrue(condition, "explanation")`	Passes if **`condition`** is true.
`assertFalse(condition, "explanation")`	Passes if **`condition`** is false.
`assertEquals(expected, actual, "explanation")`	Passes if **`expected`** and **`actual`** are equal. **`expected`** and **`actual`** can be of any Java type.
`assertNull(object, "explanation")`	Passes if **`object`** is **`null`**.
`assertNotNull(object, "explanation")`	Passes if **`object`** is not **`null`**.

Table 7.2 Some of the most common JUnit assert methods

With this knowledge about frameworks and JUnit, we can understand the first example in listing 7.2 in more detail. The complete test for the `equals` method in the `Amount` class can be found in listing 7.3.

```
1  package se.leiflindback.oodbook.tests.firstexample;
2
3  import org.junit.jupiter.api.AfterEach;
4  import org.junit.jupiter.api.BeforeEach;
5  import org.junit.jupiter.api.Test;
6  import static org.junit.jupiter.api.Assertions.*;
7
8  public class AmountTest {
9      private Amount amtNoArgConstr;
10     private Amount amtWithAmtThree;
11
12     @BeforeEach
13     public void setUp() {
14         amtNoArgConstr = new Amount();
15         amtWithAmtThree = new Amount(3);
16     }
17
18     @AfterEach
19     public void tearDown() {
20         amtNoArgConstr = null;
21         amtWithAmtThree = null;
22     }
```

```
23
24      @Test
25      public void testEqualsNull() {
26          Object other = null;
27          boolean expResult = false;
28          boolean result = amtNoArgConstr.equals(other);
29          assertEquals(expResult, result,
30                       "Amount instance equal to null.");
31      }
32
33      @Test
34      public void testEqualsJavaLangObject() {
35          Object other = new Object();
36          boolean expResult = false;
37          boolean result = amtNoArgConstr.equals(other);
38          assertEquals(expResult, result,
39                   "Amount instance equal to " +
40                   "java.lang.Object instance.");
41      }
42
43      @Test
44      public void testNotEqualNoArgConstr() {
45          int amountOfOther = 2;
46          Amount other = new Amount(amountOfOther);
47          boolean expResult = false;
48          boolean result = amtNoArgConstr.equals(other);
49          assertEquals(expResult, result,
50                       "Amount instances with different " +
51                       " states are equal.");
52      }
53
54      @Test
55      public void testNotEqual() {
56          int amountOfOther = 2;
57          Amount other = new Amount(amountOfOther);
58          boolean expResult = false;
59          boolean result = amtWithAmtThree.equals(other);
60          assertEquals(expResult, result,
61                       "Amount instances with different " +
62                       " states are equal.");
63      }
64
65      @Test
66      public void testEqual() {
67          int amountOfOther = 3;
68          Amount other = new Amount(amountOfOther);
```

```
69          boolean expResult = true;
70          boolean result = amtWithAmtThree.equals(other);
71          assertEquals(expResult, result,
72                       "Amount instances with same states " +
73                       "are not equal.");
74      }
75  }
```

Listing 7.3 Complete unit test for the `equals` method in listing 7.1

On line 12 in listing 7.3, the `setUp` method is annotated `@BeforeEach`. This means it is executed before each test method. That way, each test is performed on the two fresh `Amount` objects, that were just created in `setUp`. In a similar way, the `tearDown` method, which is annotated `@AfterEach` on line 18, is executed after each test method. That way, the `Amount` instances on which the test was performed, are dropped, and will not be used for any more test. Each method containing a test is annotated `@Test`, see lines 24, 33, 43, 54, and 65. Each of these methods will be called by JUnit when the tests are executed. All test methods follow the same pattern. First, they set up the test creating required objects. Second, they define the expected result of the call to the SUT. Third, the SUT is called and the actual result is saved. Finally, the expected and actual results are evaluated to check if the test passed. This is a very typical layout of a test method, but there are other alternatives, as we will see below.

7.2 Unit Testing Best Practices

Much can be said about best practices for unit tests, but absolutely most important is to get started writing them. Any test, no matter what shortcomings it has, is better than no test at all. Therefore, do not spend such an amount of time planning tests that the burden of writing them becomes too big, and in the end they are not written at all, or maybe cover only a small part of the code. Better to start writing and improve them later, when need is discovered. Below follows a collection of best practices for unit testing.

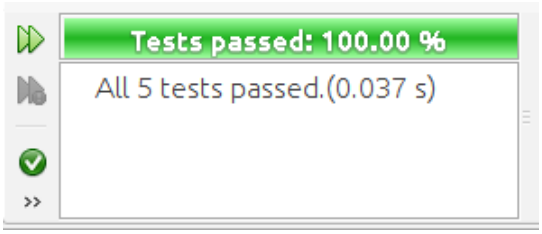


Figure 7.4 Most important is that tests are written. Nothing beats the feeling of seeing all tests pass and knowing that the program works.

Write tests! Preferably a lot of them. It is essential to have a large and increasing number of unit tests. To reach this goal, make it a habit never to test anything manually. Whenever program functionality shall be verified, write a test. Never give input and evaluate output manually without having first written a unit test. Also never remove a test. If a certain test seems unnecessary, add an `@Disabled` annotation instead of deleting it. A test that at some point seems meaningless might very well later turn out to be useful.

Tests for every known bug When a bug is found, immediately add a test that fails because of the bug. Only when that test is in place may bug fixing start. That way, there will always be a test for every known condition that might make the program fail.

Do not over-design There is no need to design or document test code as thoroughly as the product that is tested. Allow a certain amount of hacking when writing tests. In test code, we can play around a bit and write some of those funny and interesting hacks that never really seem to fit in production code.

Organization Place a test class in the same package as the class it tests. This enables testing of package private methods. However, do not mix test classes with the SUT. It is better to maintain two different directory structures, one for the program itself and one for the tests, figure 7.5. That way it is easy to see which code contains tests and which is the actual SUT. It is also easy to deliver only the product itself, without tests. These two parallel directory structures are maintained by all common IDEs, it is not required to arrange them manually.

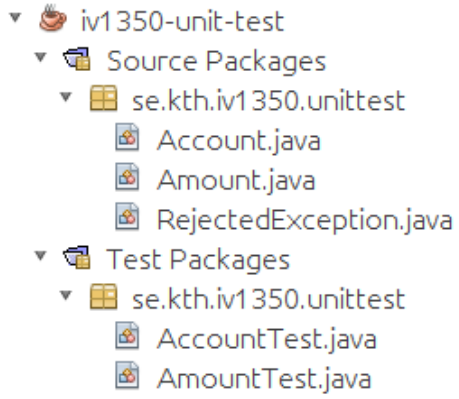


Figure 7.5 A test class is placed in the same package as the class it tests, but in a different directory.

Normally, one test class is written for each tested class, and given the same name as the tested class, with the word `Test` appended to the name. For example, the tests for a class called `Person` are in a class called `PersonTest`. This makes it easy to find the tests for a certain class. Names of test methods normally start with `test`, followed by a description of the test. For example, the method `testNotEqual` on line 55 in listing 7.3 tests that the `equals` method returns `false` for two objects that are not equal. It is possible to write any number of assertions in the same test method, but execution will stop after first failed assertion. Also, test results are listed per method, individual assertions are not shown. Therefore, it is best to write few assertions in each method, and instead write more methods.

Testing takes time, but it is worth that time A rough estimate, which is true remarkably often, is that test code has about the same length as the tested code, and takes about the same time to write. This means it is quite time consuming to write tests. However, it is also true remarkably often that once the tests are in place, they give immediate return on the time invested in writing them. This return comes as reduced time to update the code, since it will be easy to verify that it is still working after it has been changed. This will give us confidence that the code is really working, and make us more willing to improve and extend it.

Independent and self-evaluating To get highest possible value from the tests, they shall be quick and easy to execute. This means they shall start with one command (or one click in an IDE), no complex manual setup shall be required. Also, they must be self-evaluating. Either a test passes, and prints nothing, or it fails, and gives a short informative message about the failure. It must not be required to manually evaluate return values from calls to the SUT. Finally, all tests must be independent. Do not rely on them being executed in a specific order, or on previous tests having passed. All test executions must give the same result.

What to test? Test public, protected and package private code, but not private. A private method can not be tested, since it can not be called from a test class. There is also no need to test a private method, because if methods with all other accessibilities work, also private methods work. Other categories of methods that do not require testing are setters or constructors that only store a value, and getters that only return a value. Unless bugs appear, we can take for granted that such methods work as intended. Apart from these cases, tests shall cover as much as possible of the code in the SUT. Try to cover all branches of `if` statements. Also, try to test boundary conditions and extreme parameter values, like `null`, zero, negative values, objects of wrong type etc. It is also important to test that a method fails the correct way if illegal parameter values are given, or if some other precondition is not met.

7.3 When Testing is Difficult

Some methods are very complicated to test, but it is almost always possible! This section covers three different situations when testing might be difficult. One, it is hard to give input to the SUT. Two, it is hard to read the test result. Three, the SUT has complex dependencies on other objects and is therefore hard to start.

Figure 7.6 Testing can be very complicated and frustrating. However, with hard work and a lot of fantasy, it is almost always possible to write tests without worsening SUT design. Image by unknown creator [Public domain], via https://pixabay.com

Hard to give input The SUT might not get input from method parameters, but from a file, a database, a complex set of objects or another source. In this case, it is not always obvious how the test

shall provide the input, but it is virtually always possible to make it create everything the SUT requires. If the SUT reads from file, the test can write a file with appropriate content. If the SUT reads from a database, the test can create a database or insert data into an existing database. If the SUT gets data from other objects, the test can create all objects needed and somehow make them available to the SUT. It is quite common to write a large amount of test code in order to create files, databases, etc required for testing. Just remember that tests must be completely independent and repeatable, a test must leave no traces of its execution. Whatever test structure is created, must be deleted after the test is executed.

Hard to read output It might be that no usable result is returned by the tested method, nor is there any getter that can be used to read the result. In this situation, do never write a getter to facilitate testing, since it breaks encapsulation of the SUT. This is a slightly controversial statement, it is often suggested to break encapsulation by adding a get method to enable retrieving the state of an object. The only purpose of such a method would be to verify that the state is correct after a method in the object has been called from a test. This is, however, practically never necessary. Using hard work, a lot of fantasy, and by pragmatically testing more than one method together, tests can almost always be written without worsening SUT design. A method must have some effect somewhere, otherwise it would be useless. To test it, we just have to find a way to dig out that effect. As an example, consider a class that can create, read, update and delete values in some storage that can not be accessed by test code. These methods can be tested together, for example create a value, read it, and check that the read value equals the created value. Then create a value, delete it and verify that it can not be read. Then create, update, read, etc.

Complex dependencies Classes in higher layers depend on classes in lower layers. The controller in figure 7.7, to the right, depends on the model, the database integration layer and the database itself. If a test for the controller fails, we do not know which of these layers has the bug. A simple solution to this problem is to write unit tests as usual for all classes, and let all tests execute code all the way down to the database. The lowest class with a failed test is the class with the bug. This is not a pure unit test, since a call to the controller will execute code also in the model and integration layers. However, this does not matter very much since all code is tested and it is possible to locate bugs. What is more important, is that this strategy leaves the SUT completely unchanged!

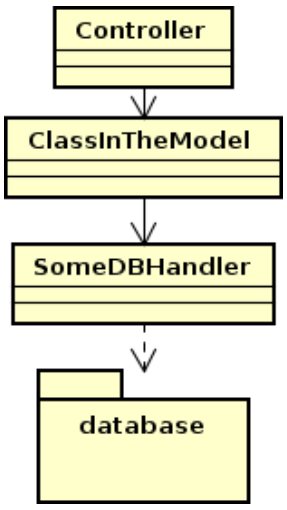


Figure 7.7 The SUT might be hard to test since it depends on many other objects.

Finally, it can not be stated often enough, *whatever the problem is, do not worsen SUT design just to enable testing*. There must be a way to test without changing the SUT.

7.4 Unit Testing Case Study

This section does not include a complete listing of all unit tests. That can be found in the accompanying NetBeans project, which can be downloaded from GitHub [Code]. Here follows a description of the first tests that were written, and of tests where particular afterthought was needed.

NetBeans Support for Unit Testing

NetBeans [NB] is the IDE used when developing this unit test case study, and this section illustrates how it facilitates creating the tests. Such functionality is not unique for NetBeans, similar functionality is available also in all other major IDEs.

To generate a new test class in NetBeans, right-click the project and choose `New` → `Test for Existing Class...`. This will display the `New Test For Existing Class` dialog, which is depicted in figure 7.8. Click the `Browse...` button, marked with a red circle, to choose for which class tests shall be generated. In this example, the chosen class is `CarRegistry`, from the rent car case study. When the SUT has been chosen, click `Finish`, and NetBeans will generate test code similar to listing 7.4.

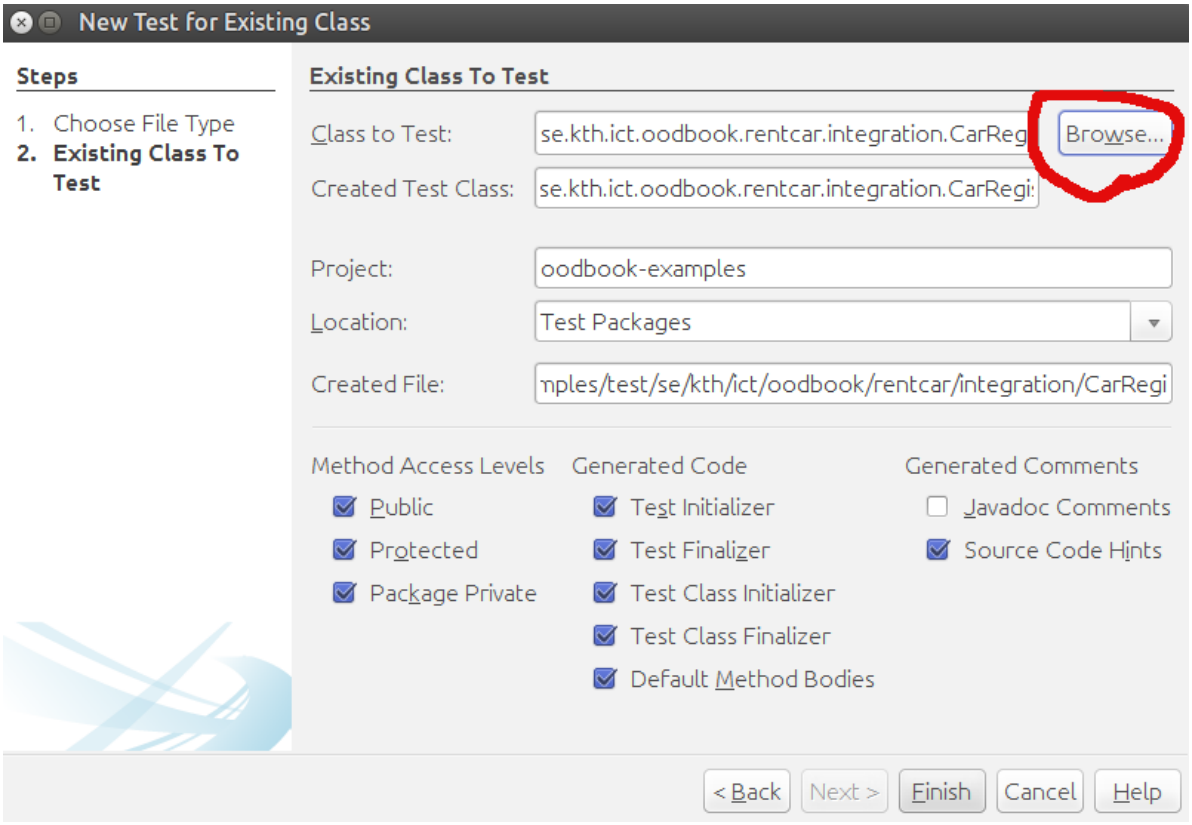


Figure 7.8 NetBeans' `New Test For Existing Class` dialog. The `Browse...` button, marked with a red circle, is used to decide for which class tests shall be generated.

```
1  package se.kth.ict.oodbook.rentcar.integration;
2
3  import org.junit.jupiter.api.AfterEach;
4  import org.junit.jupiter.api.AfterAll;
5  import org.junit.jupiter.api.BeforeEach;
6  import org.junit.jupiter.api.BeforeAll;
7  import org.junit.jupiter.api.Test;
8  import static org.junit.jupiter.api.Assertions.*;
9
10 public class CarRegistryTest {
11     @BeforeAll
12     public static void setUpClass() {
13     }
14
15     @AfterAll
16     public static void tearDownClass() {
17     }
18
19     @BeforeEach
20     public void setUp() {
21     }
22
23     @AfterEach
24     public void tearDown() {
25     }
26
27     @Test
28     public void testFindAvailableCar() {
29         System.out.println("findAvailableCar");
30         CarDTO searchedCar = null;
31         CarRegistry instance = new CarRegistry();
32         CarDTO expResult = null;
33         CarDTO result = instance.findAvailableCar(searchedCar);
34         assertEquals(expResult, result);
35         fail("The test case is a prototype.");
36     }
37 }
```

Listing 7.4 Part of the skeleton code for a test class that was generated by NetBeans.

The class has the same name as the tested class, but with `Test` appended to the class name (line 10). All four *before* and *after* methods are generated (lines 11-25), but they are empty. If some code is needed to prepare a test or to clean up after a test, it shall be added here. Methods that remain empty can be removed. One test method is generated for each public, protected or package private method in the SUT, but only one such method is included in the listing (lines 27-36). These methods contain a printout (line 29), which should be removed since tests are not supposed to produce any output if they pass. After this, the test methods create an instance of the SUT (line 31) and of other objects that are necessary to perform the test (line 30). There is no guarantee these objects are created correctly, always check if changes are required. Next, the tested method is called and the result is saved in a variable (line 33). Then an assertion is called to evaluate the test result (line 34). Again, there is no guarantee that the assertion is correct. Finally, there is a call to `fail` (line 35), which should be removed when the test is completed.

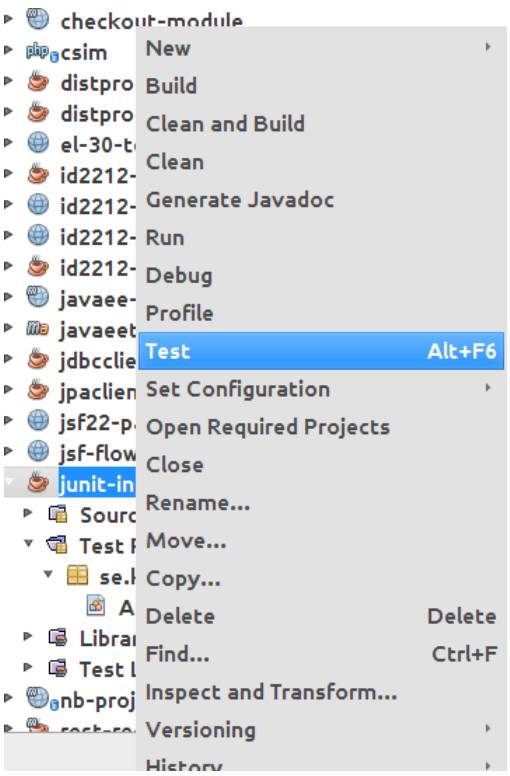


Figure 7.9 To run the tests, right-click the NetBeans project and chose `Test`.

To execute the tests, right-click the NetBeans project and chose `Test`, as depicted in figure 7.9. The test result will be displayed in a window similar to figure 7.10. Currently, none of the two auto-generated tests, `testFindAvailableCar` and `testSetBookedStateOfCar`, are implemented. As a result, they both fail.

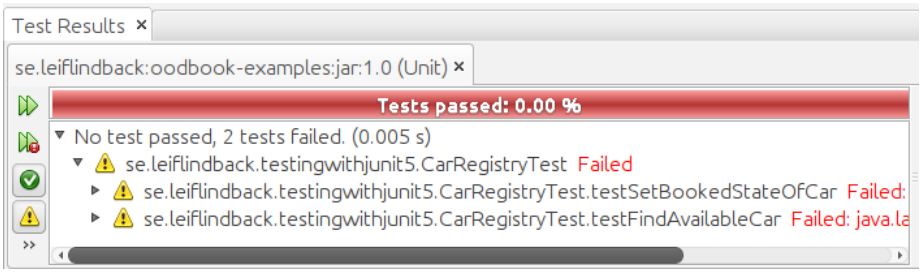


Figure 7.10 NetBeans' test result window. The auto-generated code always makes a test fail.

IntelliJ Support for Unit Testing

This section shows how to create unit tests if IntelliJ [IJ] is used instead of NetBeans. As mentioned above, similar functionality is available in all major IDEs. To generate a test class in IntelliJ, place the cursor in the source code on the name of the class for which a test shall be generated, click `alt-enter` and choose `Create Test`, as illustrated in figure 7.11. This will display the `Create Test` dialog, which is depicted in figure 7.12. Choose Junit 5 testing library, marked with red in the figure. If there is the message *JUnit 5 library not found in the module*, marked with blue in the figure, click the `Fix` button, which is marked with green. Note that the error message might not disappear, but it will still be possible to create the test. Finally, check the methods that shall be created, click `OK`, and IntelliJ will generate an empty test skeleton code similar to listing 7.5. To run the tests, right-click the project and choose `Run 'All Tests'`.

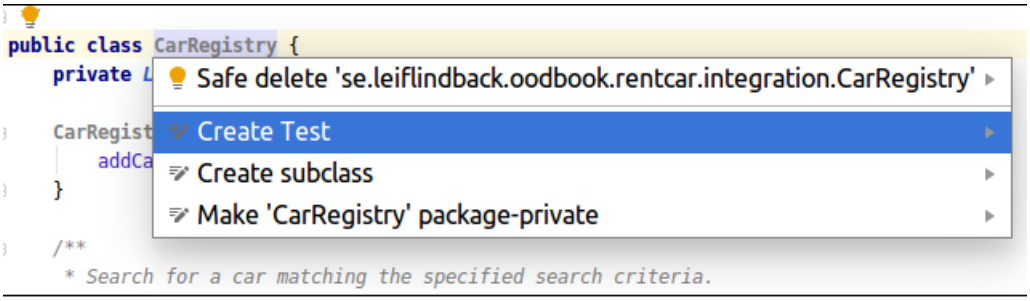


Figure 7.11 Right-click the class name in the source code and choose `Create Test` to create tests for a class.

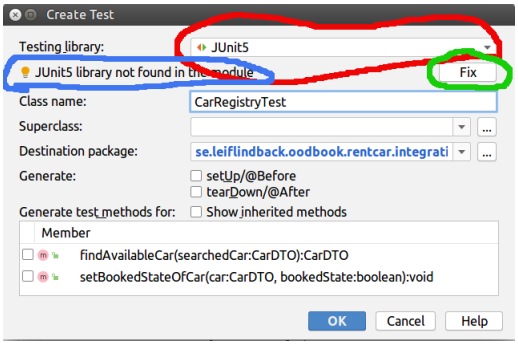


Figure 7.12 IntelliJ's `Create Test` dialog. The drop-down menu marked with a red circle is used to decide which test library to use. The message marked with a blue circle tells that the JUnit library is not installed, click the `Fix` button, marked with green, to install it.

```
1  package se.leiflindback.oodbook.rentcar.integration;
2
3  import org.junit.jupiter.api.AfterEach;
4  import org.junit.jupiter.api.BeforeEach;
5  import org.junit.jupiter.api.Test;
6
7  import static org.junit.jupiter.api.Assertions.*;
8
9  class CarRegistryTest {
10
11     @BeforeEach
12     void setUp() {
13     }
14
15     @AfterEach
16     void tearDown() {
17     }
18
19     @Test
20     void findAvailableCar() {
21     }
22
23     @Test
24     void setBookedStateOfCar() {
25     }
26 }
```

Listing 7.5 Skeleton code for a test class, generated by IntelliJ.

Writing the Tests

Tests will be written in bottom-up order, first for classes with no dependencies on other classes, then for classes with dependencies. This means we will only write tests for classes without dependencies, or for classes that are already tested. That way, it becomes possible to run a test as soon as it is written, and immediately know if the tested class works as intended.

It must be emphasized that the workflow followed here is very unnatural. The entire program was written before starting to write tests, while normally a test is written either before or immediately after the method it tests. The only reason for this workflow is to mix theory and practice, by using programming best practices as soon as they were introduced, instead of of first covering both programming and unit testing best practices, and then write code.

This odd workflow means the first task is to identify a class with no dependency on any other class. A natural place to start looking for such a class is in the lowest layer, `integration`. That layer contains `CarDTO`, which depends on `Amount`; `CarRegistry`, which depends on `CarDTO`; `RegistryCreator`, which depends on `CarRegistry` and `RentalRegistry`;

`RentalRegistry`, which depends on `Rental`; and `Printer`, which depends on `Receipt`. No suitable class was found in the `integration` layer, next candidates will be classes on which the classes in the `integration` layer depends. The first that was mentioned was `Amount`, which in fact has no dependency. It will be the first class to test.

The First Tested Class, `Amount`

All public, protected and package private methods shall be tested, but not private methods. The `Amount` class has only public methods and constructors, which thus shall all be tested. The constructors, however, contain no logic, they only set a value, and will hence not be tested.

```
1  /**
2   * Two <code>Amount</code>s are equal if they represent the
3   * same amount.
4   *
5   * @param other The <code>Amount</code> to compare with this
6   *              amount.
7   * @return <code>true</code> if the specified amount is equal
8   *         to this amount, <code>false</code> if it is not.
9   */
10 @Override
11 public boolean equals(Object other) {
12     if (other == null || !(other instanceof Amount)) {
13         return false;
14     }
15     Amount otherAmount = (Amount) other;
16     return amount == otherAmount.amount;
17 }
```

Listing 7.6 The `equals` method of the `Amount` class

The first method that is tested is `equals`, since other tests will use it, as will soon be clear. The `equals` method is listed in listing 7.6. To take the branch `other == null`, on line 12, the method must be called with a `null` parameter. To take the `!(other instanceof Amount)` branch, also on line 12, the parameter must be an object that is not an instance of `Amount`. An easy choice is to use a `java.lang.Object` instance. Finally, there are two different executions of line 16, one where `amount == otherAmount.amount` and one where `amount != otherAmount.amount`. These tests can be found in listing 7.7.

```
1  package se.leiflindback.oodbook.rentcar.model;
2
3  import org.junit.jupiter.api.AfterEach;
4  import org.junit.jupiter.api.BeforeEach;
5  import org.junit.jupiter.api.Test;
6  import static org.junit.jupiter.api.Assertions.*;
7
```

```
 8 public class AmountTest {
 9     private Amount amtNoArgConstr;
10     private Amount amtWithAmtThree;
11
12     @BeforeEach
13     public void setUp() {
14         amtNoArgConstr = new Amount();
15         amtWithAmtThree = new Amount(3);
16     }
17
18     @AfterEach
19     public void tearDown() {
20         amtNoArgConstr = null;
21         amtWithAmtThree = null;
22     }
23
24     @Test
25     public void testNotEqualsNull() {
26         Object other = null;
27         boolean expResult = false;
28         boolean result = amtNoArgConstr.equals(other);
29         assertEquals(expResult, result,
30                     "Amount instance equal to null.");
31     }
32
33     @Test
34     public void testNotEqualsJavaLangObject() {
35         Object other = new Object();
36         boolean expResult = false;
37         boolean result = amtNoArgConstr.equals(other);
38         assertEquals(expResult, result,
39                     "Amount instance equal to " +
40                     "java.lang.Object instance.");
41     }
42
43     @Test
44     public void testNotEqualUsingNoArgConstr() {
45         boolean expResult = false;
46         boolean result = amtNoArgConstr.equals(amtWithAmtThree);
47         assertEquals(expResult, result,
48                     "Amount instances with different " +
49                     "states are equal.");
50     }
51
52     @Test
53     public void testNotEqual() {
```

```
54          int amountOfOther = 2;
55          Amount other = new Amount(amountOfOther);
56          boolean expResult = false;
57          boolean result = amtWithAmtThree.equals(other);
58          assertEquals(expResult, result,
59                       "Amount instances with different " +
60                       "states are equal.");
61      }
62
63      @Test
64      public void testEqual() {
65          int amountOfOther = 3;
66          Amount other = new Amount(amountOfOther);
67          boolean expResult = true;
68          boolean result = amtWithAmtThree.equals(other);
69          assertEquals(expResult, result,
70                       "Amount instances with same states are " +
71                       "not equal.");
72      }
73  }
```

Listing 7.7 Tests for all possible paths through the `equals` method of the `Amount` class

Next thing to look for is extreme values of the parameters. All obvious extreme values, like `null`, are already covered. However, since this is our first test, we might be extra careful and write a test also for a successful comparison with `Amount` objects representing zero. Theoretically, we could test with `Amount`s representing positive, negative, `Integer.MAX_VALUE` and `Integer.MIN_VALUE` amounts, but there is really no reason to suspect that the `equals` method would behave differently for such values. Listing 7.8 shows the test for an `Amount` representing the value zero.

```
1  @Test
2  public void testEqualNoArgConstr() {
3      int amountOfOther = 0;
4      Amount other = new Amount(amountOfOther);
5      boolean expResult = true;
6      boolean result = amtNoArgConstr.equals(other);
7      assertEquals(expResult, result,
8                   "Amount instances with same states " +
9                   "are not equal.");
10 }
```

Listing 7.8 Test for the `equals` method of an `Amount` representing the amount zero.

Finally, is there any way the parameter can have an illegal value, or is there some precondition that must be met for method to work properly? The answer is no, the method should

function the same way for all possible parameter values. That means we are done testing it. Remember to run the tests and check that they all pass, figure 7.13. Our first green bar!!

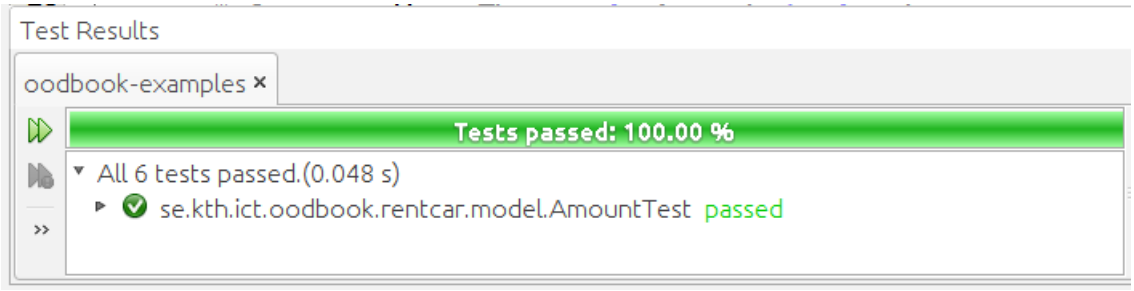


Figure 7.13 All tests for the `equals` method pass.

The other `Amount` methods, namely `minus`, `plus` and `toString`, are independent, neither any of the methods, nor its test, will use any of the other methods. Therefore, they can be written in any order. Let's start with `minus`, which is listed in listing 7.9.

```
1  /**
2   * Subtracts the specified <code>Amount</code> from this
3   * object and returns an <code>Amount</code> instance with
4   * the result.
5   *
6   * @param other The <code>Amount</code> to subtract.
7   * @return The result of the subtraction.
8   */
9  public Amount minus(Amount other) {
10     return new Amount(amount - other.amount);
11 }
```

Listing 7.9 The `minus` method of the `Amount` class

This method has only one execution path, since there are no flow control statements. There are no illegal parameter values, but the subtraction may overflow. If, for example, -1 is subtracted from `Integer.Min_VALUE`, the result is a negative integer with a magnitude too big to fit in an `int`. In fact, we have discovered a flaw in the design. The method ought to check if an overflow occurred, and, if so, throw an exception. However, since exception handling is covered in a later chapter, this check is not introduced here. Instead, an explaining text is added to the javadoc comment, saying that `The operation will overflow if the result is smaller than <code>Integer.MIN_VALUE</code>`. What can then be tested regarding overflow? Nothing in fact, the method might fail, but the failure is not handled in any way. The conclusion is that one test would probably be enough, just perform a subtraction and check that the result is correct. However, since we have just started, let's be a bit overambitious and test positive, negative and zero results, see listing 7.10. Once the first test for `minus` is written, it takes about thirty seconds to add the other two, and the more tests

that pass, the greater the pleasure to see them pass. Note that `assertEquals`, which is called on lines 10, 23 and 35, will use the `equals` method in `Amount` to verify that the two specified `Amount` instances are equal. This is why it was important to know that `equals` worked when `minus` was tested. It is now clear that if a test for `minus` fails, it is because of a bug in `minus`, not in `equals`.

```
1  @Test
2  public void testMinus() {
3      int amountOfOperand1 = 10;
4      int amountOfOperand2 = 3;
5      Amount operand1 = new Amount(amountOfOperand1);
6      Amount operand2 = new Amount(amountOfOperand2);
7      Amount expResult = new Amount(amountOfOperand1 -
8                                    amountOfOperand2);
9      Amount result = operand1.minus(operand2);
10     assertEquals(expResult, result,
11                  "Wrong subtraction result");
12 }
13
14 @Test
15 public void testMinusNegResult() {
16     int amountOfOperand1 = 3;
17     int amountOfOperand2 = 10;
18     Amount operand1 = new Amount(amountOfOperand1);
19     Amount operand2 = new Amount(amountOfOperand2);
20     Amount expResult = new Amount(amountOfOperand1 -
21                                   amountOfOperand2);
22     Amount result = operand1.minus(operand2);
23     assertEquals(expResult, result,
24                  "Wrong subtraction result");
25 }
26
27 @Test
28 public void testMinusZeroResultNegOperand() {
29     int amountOfOperand1 = -3;
30     int amountOfOperand2 = -3;
31     Amount operand1 = new Amount(amountOfOperand1);
32     Amount operand2 = new Amount(amountOfOperand2);
33     Amount expResult = new Amount(amountOfOperand1 -
34                                   amountOfOperand2);
35     Amount result = operand1.minus(operand2);
36     assertEquals(expResult, result,
37                  "Wrong subtraction result");
38 }
```

Listing 7.10 The tests for the `minus` method of the `Amount` class

The tests for `plus` are created exactly the same way as the tests for `minus`, and are therefore not covered here. Finally, there is the `toString` method, which returns a string representation of the amount, listing 7.11. Also this method is tested with positive, negative, and zero amounts, see listing 7.12. That concludes testing `Amount`. There are 15 tests in total, and all pass, brilliant!

```
1 @Override
2 public String toString() {
3     return Integer.toString(amount);
4 }
```

Listing 7.11 The `toString` method of the `Amount` class

```
1  @Test
2  public void toStringPosAmt() {
3      int representedAmt = 10;
4      Amount amount = new Amount(representedAmt);
5      String expResult = Integer.toString(representedAmt);
6      String result = amount.toString();
7      assertEquals(expResult, result,
8                   "Wrong string returned by toString");
9  }
10
11 @Test
12 public void toStringNegAmt() {
13     int representedAmt = -10;
14     Amount amount = new Amount(representedAmt);
15     String expResult = Integer.toString(representedAmt);
16     String result = amount.toString();
17     assertEquals(expResult, result,
18                  "Wrong string returned by toString");
19 }
20
21 @Test
22 public void toStringZeroAmt() {
23     int representedAmt = 0;
24     Amount amount = new Amount(representedAmt);
25     String expResult = Integer.toString(representedAmt);
26     String result = amount.toString();
27     assertEquals(expResult, result,
28                  "Wrong string returned by toString");
29 }
```

Listing 7.12 The tests for the `toString` method of the `Amount` class

The First Problematic Test, a `void` Method

Following the same reasoning as when testing `Amount`, tests are written also for `CarDTO`. This is quite straightforward, just remember that object parameters must be tested with `null`, and string parameters also with an empty string. The code for these tests can be found in the accompanying NetBeans project [Code]. The next class to test is `CarRegistry`, which is a bit more challenging since it has a `void` method, namely `setBookedStateOfCar`. Remember the strategy, *the method must have some effect somewhere, just locate that effect*. Here the effect is that a booked car will not be returned by `findAvailableCar`, even if the description matches. Therefore, `setBookedStateOfCar` can be tested together with `findAvailableCar`, as illustrated in listing 7.14. Listing 7.13 shows `setBookedStateOfCar` and `findAvailableCar`. There is an interesting problem here, the calls to `assertEquals`, for example on line nine in listing 7.14, will use the `equals` method in `CarDTO` to evaluate if two instances are equal. But there is no such method, which means the default instance of `equals`, in `java.lang.Object`, will be used! That method considers two objects to be equal only if they are exactly the same object, residing in the same memory location. Since this is not appropriate here, an `equals` method is added to `CarDTO`, and of course it is also tested. It could be argued that the SUT is now changed, only to facilitate testing. That might be the case, but what really matters is that the design of the SUT is not worsened. Furthermore, an `equals` method might very well turn out to be appropriate for the SUT itself.

```
1  /**
2   * Search for a car matching the specified search criteria.
3   *
4   * @param searchedCar This object contains the search criteria.
5   *                    Fields in the object that are set to
6   *                    <code>null</code> or <code>0</code> are
7   *                    ignored.
8   * @return <code>true</code> if a car with the same features
9   *         as <code>searchedCar</code> was found,
10  *         <code>false</code> if no such car was found.
11  */
12 public CarDTO findAvailableCar(CarDTO searchedCar) {
13     for (CarData car : cars) {
14         if (matches(car, searchedCar) && !car.booked) {
15             return new CarDTO(car.regNo, new Amount(car.price),
16                               car.size, car.AC, car.fourWD,
17                               car.color);
18         }
19     }
20     return null;
21 }
22
23 /**
24  * If there is an existing car with the registration number of
```

```
25   * the specified car, set its booked property to the specified
26   * value. Nothing is changed if the car's booked property
27   * already had the specified value.
28   *
29   * @param car         The car that shall be marked as booked.
30   * @param bookedState The new value of the booked property.
31   */
32  public void setBookedStateOfCar(CarDTO car,
33                                   boolean bookedState) {
34      CarData carToBook = findCarByRegNo(car);
35      carToBook.booked = bookedState;
36  }
```

Listing 7.13 The `setBookedStateOfCar` and `findAvailableCar` methods of the `CarRegistry` class

```
1   @Test
2   public void testSetBookedStateOfCar() {
3       CarDTO bookedCar = new CarDTO("abc123", new Amount(1000),
4                                     "medium", true, true, "red");
5       CarRegistry instance = new CarRegistry();
6       instance.setBookedStateOfCar(bookedCar, true);
7       CarDTO expResult = null;
8       CarDTO result = instance.findAvailableCar(bookedCar);
9       assertEquals(expResult, result, "Booked car was found");
10  }
```

Listing 7.14 The test for the `setBookedStateOfCar` method of the `CarRegistry` class

More Difficult Tests

No tests are needed for `CustomerDTO`, `AddressDTO` or `DrivingLicenseDTO`, since they contain only setters, getters and constructors that do nothing but save or return values. `CashRegister` and `RentalRegistry` can, in fact, not be properly tested. They contain one `void` method each, that only updates a field in the same object. This is not a proof that there are untestable methods, instead it shows that the program is not complete. As the development continues, it will certainly become possible to read the balance of the cash register and to see which rentals have been made. Thereby, it will also possible to test those classes. For now, however, all that can be done is to call those methods in their tests, there is no way to evaluate their outcome. The next class that is tested is `RegistryCreator`, tests are available in the accompanying NetBeans project [Code].

After that it is not possible to continue in pure bottom-up order any more, since there are no remaining classes without dependencies or depending only on tested classes. The only thing to do is to write tests for all remaining classes in `model` and `integration`, and then run all

of them. Writing tests for these classes clearly shows, as was already known, that there is no error handling in this program. For example, there are many methods which should check that they are not called with `null` parameters, but they do not. This lack of error handling makes it meaningless to test those erroneous conditions. That must be postponed until later, when error handling is added.

The test environment for some methods, for example `createReceiptString` in `Receipt`, listing 7.15, require quite a lot of work to set up. This is quite normal, and should be expected to happen. Listing 7.16 shows `testCreateReceiptString`, which illustrates that with some fantasy and quite a lot of code, it is possible to test also methods depending on many other objects or methods. Note that the string that makes up the expected result (line 19-24) does not, in any way, depend on `createReceiptString` or any other method in `Receipt`. This eliminates the risk that the test has the same bug as the SUT.

```
1  /**
2   * Creates a well-formatted string with the entire content of
3   * the receipt.
4   *
5   * @return The well-formatted receipt string.
6   */
7  public String createReceiptString() {
8      StringBuilder builder = new StringBuilder();
9      appendLine(builder, "Car Rental");
10     endSection(builder);
11
12     LocalDateTime rentalTime = LocalDateTime.now();
13     builder.append("Rental time: ");
14     appendLine(builder, rentalTime.toString());
15     endSection(builder);
16
17     builder.append("Rented car: ");
18     appendLine(builder, rental.getRentedCar().getRegNo());
19     builder.append("Cost: ");
20     appendLine(builder, rental.getPayment().getTotalCost().
21                         toString());
22     builder.append("Change: ");
23     appendLine(builder, rental.getPayment().getChange().
24                         toString());
25     endSection(builder);
26
27     return builder.toString();
28 }
29
30 private void appendLine(StringBuilder builder, String line) {
31     builder.append(line);
32     builder.append("\n");
33 }
```

```
34
35 private void endSection(StringBuilder builder) {
36     builder.append("\n");
37 }
```

Listing 7.15 The `createReceiptString` method of the `Receipt` class, and its private helper methods.

```
1  @Test
2  public void testCreateReceiptString() {
3      Amount price = new Amount(100);
4      String regNo = "abc123";
5      String size = "medium";
6      boolean AC = true;
7      boolean fourWD = true;
8      String color = "red";
9      CarDTO rentedCar = new CarDTO(regNo, price, size, AC,
10                                   fourWD, color);
11     Amount paidAmt = new Amount(500);
12     CashPayment payment = new CashPayment(paidAmt);
13     Rental paidRental = new Rental(null, new RegistryCreator().
14                                    getCarRegistry());
15     paidRental.setRentedCar(rentedCar);
16     paidRental.pay(payment);
17     Receipt instance = new Receipt(paidRental);
18     LocalDateTime rentalTime = LocalDateTime.now();
19     String expResult = "\n\nRented car: " + regNo +
20                        "\nCost: " + price +
21                        "\nChange: " + paidAmt.minus(price) +
22                        "\n\n";
23     String result = instance.createReceiptString();
24     assertTrue(result.contains(expResult), "Wrong printout.");
25     assertTrue(result.contains(Integer.toString(rentalTime.
26                                                 getYear())),
27                "Wrong rental year.");
28     assertTrue(result.contains(Integer.toString(
29                                rentalTime.getMonthValue())),
30                "Wrong rental month.");
31     assertTrue(result.contains(Integer.toString(
32                                rentalTime.getDayOfMonth())),
33                "Wrong rental day.");
34     assertTrue(result.contains(Integer.toString(
35                                rentalTime.getHour())),
36                "Wrong rental hour.");
37     assertTrue(result.contains(Integer.toString(
38                                rentalTime.getMinute())),
```

```
39                 "Wrong rental minute.");
40     assertTrue(result.contains(expResult),
41                 "Wrong receipt content.");
42 }
```

Listing 7.16 The test for the `createReceiptString` method of the `Receipt` class

Testing User Interface

`printReceipt` in `Printer` (listing 7.17) is an interesting method. It is `void`, but calling it has an effect, though only on the screen. It produces output to `System.out`. Luckily, it is easy to test such output, since it is possible to replace the stream `System.out` with another stream, that prints to a buffer in memory instead of the screen. This is done on lines 8-9 in listing 7.18. The content of this in-memory buffer becomes the outcome of the SUT call, which is compared with the expected result on line 43.

This is the only user interface test that is written, since development of user interfaces is not included in the course. However, testing `System.in` should be done exactly the same way as testing `System.out`, by reassigning the stream and let it read from an in-memory buffer instead of the keyboard. This strategy only works for command line user interfaces, not for graphical or web-based user interface. Still, it is very much possible to test also such user interfaces, since there are many frameworks which makes it possible to give input to, and read output from, different kinds of UIs.

```
1 /**
2  * Prints the specified receipt. This dummy implementation
3  * prints to <code>System.out</code> instead of a printer.
4  *
5  * @param receipt
6  */
7 public void printReceipt(Receipt receipt) {
8     System.out.println(receipt.createReceiptString());
9 }
```

Listing 7.17 The outcome of the `printReceipt` method appears only in `System.out`.

```
1 public class PrinterTest {
2     private ByteArrayOutputStream outContent;
3     private PrintStream originalSysOut;
4
5     @BeforeEach
6     public void setUpStreams() {
7         originalSysOut = System.out;
8         outContent = new ByteArrayOutputStream();
9         System.setOut(new PrintStream(outContent));
```

```
10      }
11
12      @AfterEach
13      public void cleanUpStreams() {
14          outContent = null;
15          System.setOut(originalSysOut);
16      }
17
18      @Test
19      public void testCreateReceiptString() {
20          Amount price = new Amount(1000);
21          String regNo = "abc123";
22          String size = "medium";
23          boolean AC = true;
24          boolean fourWD = true;
25          String color = "red";
26          CarDTO rentedCar = new CarDTO(regNo, price, size, AC,
27                                        fourWD, color);
28          Amount paidAmt = new Amount(5000);
29          CashPayment payment = new CashPayment(paidAmt);
30          Rental paidRental = new Rental(null,
31                                         new RegistryCreator().
32                                         getCarRegistry());
33          paidRental.setRentedCar(rentedCar);
34          paidRental.pay(payment);
35          Receipt receipt = new Receipt(paidRental);
36          Printer instance = new Printer();
37          instance.printReceipt(receipt);
38          LocalDateTime rentalTime = LocalDateTime.now();
39          String expResult = "\n\nRented car: " + regNo +
40                             "\nCost: " + price +
41                             "\nChange: " +
42                             paidAmt.minus(price) + "\n\n\n";
43          String result = outContent.toString();
44          assertTrue(result.contains(expResult),
45                     "Wrong printout.");
46          assertTrue(result.contains(Integer.toString(
47                                       rentalTime.getYear())),
48                     "Wrong rental year.");
49          assertTrue(result.contains(Integer.toString(
50                                     rentalTime.getMonthValue())),
51                     "Wrong rental month.");
52          assertTrue(result.contains(Integer.toString(
53                                     rentalTime.getDayOfMonth())),
54                     "Wrong rental day.");
55          assertTrue(result.contains(Integer.toString(
```

```
56                                      rentalTime.getHour())),
57                      "Wrong rental hour.");
58          assertTrue(result.contains(Integer.toString(
59                                      rentalTime.getMinute())),
60                      "Wrong rental minute.");
61      }
62  }
```

Listing 7.18 The test for the `createReceiptString` method of the `Receipt` class

The last classes, `Controller` and `Main`

As was mentioned above, user interface testing is not included in the course. Thus, the only remaining classes are `Controller` and `Main`. The tests for `Controller` once again reveals the lack of a possibility to read from the rental registry, it is impossible to check any property of the rental that is created by the `Controller` methods. In fact, a "read-only registry" is a very strange thing, why store anything in the registry if it can not later be read? It is impossible to claim that the design is worsened by a read method in `RentalRegistry`. Rather, it is a bug that there is no such method. According to this reasoning, the method `findRentalByCustomerName` is added to the `RentalRegistry`. It returns all rentals made by a customer with the specified name. Having added this method, the test of `saveRental` in `RentalRegistry` can be extended to verify that the `Rental` is actually saved. According to the same reasoning, the method `getRentingCustomer` is added to `Rental`. What point is there to store information about the renting customer, if that information can not be read?

Testing the controller is a bit tricky. Since it is high up in the layer stack, both setting up test environment, giving input, and reading output, involves many other objects. As an example, consider the method `testRentalWithBookedCarIsStored` on line 26 in listing 7.20, which tests that the method `bookCar` (listing 7.19) correctly stores the current rental to the rental registry. First, lines 27-36 creates objects that are required input to the SUT. Next, line 37 prepares the SUT. Only after this call can the tested method, `bookCar` be called. Lines 39-41 and 47 extracts the result, namely the stored `Rental` object. The `assertEqual` call on lines 44-46 assures that the correct number of rentals (one) is stored. If this is not the case, there is no point in continuing the test, it has already failed. Lines 48-54 checks that the correct rental was stored in the registry. The only way to do this is to print the receipt and check that the rented car is specified there. A more straightforward way would have been to get the rented car by calling `getRentedCar` in `Rental`, but that method can not be reached by the test since it is package private. We do not want to worsen the design by making it public, and thus part of the public interface.

```
1  /**
2   * Books the specified car. After calling this method, the car
3   * can not be booked by any other customer. This method also
4   * permanently saves information about the current rental.
5   *
```

```
 * @param car The car that will be booked.
 */
public void bookCar(CarDTO car) {
    rental.setRentedCar(car);
    rentalRegistry.saveRental(rental);
}
```

Listing 7.19 The `bookcar` method of the `Controller` class.

```
public class ControllerTest {
    private Controller instance;
    private RegistryCreator regCreator;
    ByteArrayOutputStream outContent;
    PrintStream originalSysOut;

    @BeforeEach
    public void setUp() {
        originalSysOut = System.out;
        outContent = new ByteArrayOutputStream();
        System.setOut(new PrintStream(outContent));
        Printer printer = new Printer();
        regCreator = new RegistryCreator();
        instance = new Controller(regCreator, printer);
    }

    @AfterEach
    public void tearDown() {
        outContent = null;
        System.setOut(originalSysOut);
        instance = null;
        regCreator = null;
    }

    @Test
    public void testRentalWithBookedCarIsStored() {
        String customerName = "custName";
        CustomerDTO rentingCustomer =
            new CustomerDTO(customerName,
                            new AddressDTO("street", "zip",
                                           "city"),
                            new DrivingLicenseDTO("1234567"));
        String regNo = "abc123";
        CarDTO rentedCar = new CarDTO(regNo, new Amount(1000),
                                      "medium", true,
                                      true, "red");
        instance.registerCustomer(rentingCustomer);
```

```
38          instance.bookCar(rentedCar);
39          List<Rental> savedRentals =
40              regCreator.getRentalRegistry().
41              findRentalByCustomerName(customerName);
42          int expectedNoOfStoredRentals = 1;
43          int noOfStoredRentals = savedRentals.size();
44          assertEquals(expectedNoOfStoredRentals,
45                       noOfStoredRentals,
46                       "Wrong number of stored rentals.");
47          Rental savedRental = savedRentals.get(0);
48          Amount paidAmt = new Amount(5000);
49          CashPayment payment = new CashPayment(paidAmt);
50          savedRental.pay(payment);
51          savedRental.printReceipt(new Printer());
52          String result = outContent.toString();
53          assertTrue(result.contains(regNo),
54                     "Saved rental does not contain rented car");
55      }
56  }
```

Listing 7.20 The test for the `bookcar` method of the `Controller` class

The last class is `Main`, which has only the method `main`. This is very hard to test, since it does nothing but create some objects. When the program is ready, it will most likely start a user interface, then it will be possible to verify that something happens on the screen. While this can not be done now, since no user interface is created, it is still possible to verify that some chosen part of the output from the test run in `View.sampleExecution` appears on the screen. It is also possible to inspect the JVM to see that the expected objects are created, but this involves starting a debugger in another JVM, and attaching it to the inspected JVM, which is too complicated for this course. Maybe even too complicated to be meaningful at all, if the only purpose is to see that some `new` statements behave as expected.

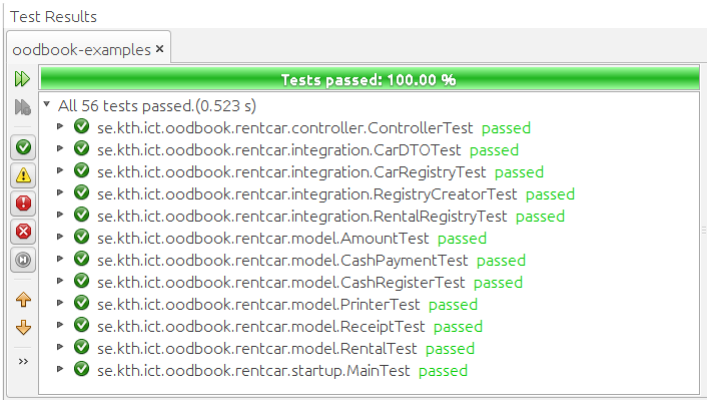


Figure 7.14 All 56 tests of the case study pass.

That concludes the rent car testing case study. A total of 56 test methods were created, which is acceptable for such a small program. All 56 tests pass, which gives the joyful sight presented in figure 7.14. Quite amazingly, the SUT consists of 1353 lines of code in total, and the tests of 1322 (no cheating). A difference of only two percent!

7.5 Common Mistakes

Below follows a list of common mistakes made when writing unit tests.

Too few tests Both the most common and most severe mistake is probably not to write enough tests. Try to cover all possible branches of `if` statements and loops. Also try to write tests for extreme and illegal parameter values.

Too many assertions in the same test method Place as few assertions as possible in each test method. It is clearer what happens and easier to give an explaining name to a test method if it has few assertions. Also, remember that a test method stops executing after the first failed assertion. Therefore, fewer assertions per method might make more assertions execute. Ideally, there should only be one assertion per test method, but it is not always possible to evaluate the outcome of a call to the SUT in one single assertion. Sometimes more than one is actually required.

NO!

Not self-evaluating Test result should be evaluated using assertions, not with `if` statements in the test methods, nor by forcing the tester to read output.

Producing output A test shall not write to `System.out`. The more tests there are, the more confusing it becomes if they print some kind of status messages.

Worsen SUT design The design of the SUT shall not be worsened just to facilitate testing. It is practically always possible to test without changing the SUT, even though it often requires extra work.

7.6 Exercises

Below are exercises that will help getting started writing unit tests. There are solutions in appendix D

Exercise 1

We are going to write unit tests for three classes, `Account`, which represents a bank account; `Holder`, which represents bank customers owning accounts; and `IllegalBankTransactionException`, which is thrown when an operation would violate the bank's business rules. To start with, we just consider the `Holder` class. It's listed below, and can also be found in the package `se.leiflindback.oodbook.tests.exercises.acct` in the accompanying git repository [Code].

The first exercise is to write a unit test for the method `getName` on lines 45-47. Normally, we wouldn't write a unit test for a getter that just returns a value, but now let's do that because it's the easiest way to get started. Write a complete test class containing just one test. It shall create an instance of the `Holder` class and verify that the holder name passed to the constructor is also returned by the method `getName`.

```
1  public class Holder {
2      private SecureRandom randomNoGenerator =
3                           new SecureRandom();
4      private Set<Account> accounts = new HashSet<>();
5      private String name;
6      private long holderNo;
7
8      /**
9       * Creates a new instance with the specified name. A
10      * unique holder number will be set on the newly
11      * created instance.
12      *
13      * @param name The holder's name.
14      */
15     public Holder(String name) {
16         this.name = name;
17         holderNo = generateHolderNo();
18     }
19
20     /**
21      * @return A set containing all accounts owned by this
22      *         account holder.
23      */
24     public Set<Account> getAccounts() {
25         Set<Account> copyOfAccts = new HashSet<>();
26         copyOfAccts.addAll(accounts);
27         return copyOfAccts;
28     }
29
30     /**
31      * Adds the specified account to the set of accounts owned
32      * by this holder. There is no limit on the number of
33      * accounts that can be owned by the same holder.
34      *
35      * @param acct The account to add to this holder's
36      *             accounts.
37      */
38     public void addAccount(Account acct) {
39         accounts.add(acct);
40     }
41
42     /**
43      * @return This holder's name.
44      */
45     public String getName() {
46         return name;
```

```
47      }
48
49      /**
50       * Changes this holder's name.
51       *
52       * @param hldName The new name of this holder.
53       */
54      public void setName(String hldName) {
55          this.name = hldName;
56      }
57
58      /**
59       * @return This holder's holder number.
60       */
61      public long getHolderNo() {
62          return holderNo;
63      }
64
65      @Override
66      public int hashCode() {
67          return Long.valueOf(holderNo).hashCode();
68      }
69
70      @Override
71      public boolean equals(Object object) {
72          if (!(object instanceof Holder)) {
73              return false;
74          }
75          Holder other = (Holder) object;
76          return this.holderNo == other.holderNo;
77      }
78
79      @Override
80      public String toString() {
81          StringBuilder builder = new StringBuilder("Holder");
82          builder.append("[");
83          addFieldToStringRep(builder, "name", name);
84          addFieldSeparatorToStringRep(builder);
85          addFieldToStringRep(builder, "holderNo",
86                              Long.toString(holderNo));
87          addFieldSeparatorToStringRep(builder);
88          builder.append("accounts: [");
89          for (Account acct : accounts) {
90              builder.append(acct);
91              addFieldSeparatorToStringRep(builder);
92          }
```

```
93          builder.append("]]");
94          return builder.toString();
95      }
96
97      private long generateHolderNo() {
98          return randomNoGenerator.nextLong();
99      }
100
101     private void addFieldToStringRep(StringBuilder builder,
102                                      String name,
103                                      String value) {
104         builder.append(name);
105         builder.append(":");
106         builder.append(value);
107     }
108
109     private void addFieldSeparatorToStringRep(
110                  StringBuilder builder) {
111         builder.append(", ");
112     }
113 }
```

Listing 7.21 The `Holder` class, which represents a holder of bank accounts.

Exercise 2

Before writing complete tests for `Holder`, let's first decide what to test. Without writing any code, decide exactly what you want to test to verify that all methods in `Holder` work as intended. Which input will you give to each test and what is the expected answer?

Exercise 3

This exercise is to write the tests for `Holder`. Extend the test class you wrote in exercise one with complete tests for all methods in `Holder`.

Exercise 4

Now consider the `Account` class in the same package as `Holder`, and listed below in listing 7.22. First decide exactly what you want to test to verify that all methods work as intended, just as was done for `Holder` in exercise two. Which input will you give to each test and what is the expected answer? Some methods in `Account` throws exceptions, but ignore all tests that would involve testing when and how an exception is thrown. How to test exceptions will be covered in chapter 8.

Exercise 5

Write a test class for `Account`, containing all tests you planned in exercise four.

```
1  package se.leiflindback.oodbook.tests.exercises.acct;
2
3  import java.security.SecureRandom;
4
5  public class Account {
6      private SecureRandom acctNoGenerator = new SecureRandom();
7      private int balance;
8      private long acctNo;
9      private Holder holder;
10
11     /**
12      * Behaves like {@link #Account(Holder, int)}, except that
13      * the balance is set to zero.
14      *
15      * @param holder The account holder.
16      */
17     public Account(Holder holder) {
18         this(holder, 0);
19     }
20
21     /**
22      * <p>
23      * Creates a new instance with the specified holder and
24      * balance. Note that an account always has exactly one
25      * holder. The newly created account will <em>not</em> be
26      * passed to the specified holder, that must be done after
27      * the constructor returns and the new instance is
28      * completely created.
29      * </p>
30      *
31      * <p>
32      * A unique account number will be set on the newly
33      * created instance.
34      * </p>
35      *
36      * @param holder The account holder.
37      * @param balance The initial balance.
38      */
39     public Account(Holder holder, int balance) {
40         this.holder = holder;
41         this.balance = balance;
42         acctNo = generateAcctNo();
43     }
```

```
44
45      /**
46       * @return This account's balance.
47       */
48      public int getBalance() {
49          return balance;
50      }
51
52      /**
53       * @return This account's account number.
54       */
55      public long getAcctNo() {
56          return acctNo;
57      }
58
59      /**
60       * @return This account's holder.
61       */
62      public Holder getHolder() {
63          return holder;
64      }
65
66      /**
67       * Withdraws the specified amount.
68       *
69       * @param amount The amount to withdraw.
70       * @throws IllegalBankTransactionException When attempting
71       *         to withdraw a negative or zero amount, or if
72       *         withdrawal would result in a negative balance.
73       */
74      public void withdraw(int amount) throws
75                                  IllegalBankTransactionException {
76          if (amount <= 0) {
77              throw new IllegalBankTransactionException(
78                    "Attempt to withdraw non-positive amount: "
79                    + amount);
80          }
81          if (amount > balance) {
82              throw new IllegalBankTransactionException(
83                      "Attempt to withdraw amount greater than"
84                      + " balance, balance: " + balance
85                      + ", amount: " + amount);
86          }
87          balance = balance - amount;
88      }
89
```

```
90      /**
91       * Deposits the specified amount.
92       *
93       * @param amount The amount to deposit.
94       * @throws IllegalBankTransactionException When attempting
95       *         to deposit a negative or zero amount.
96       */
97      public void deposit(int amount) throws
98                              IllegalBankTransactionException {
99          if (amount <= 0) {
100             throw new IllegalBankTransactionException(
101             "Attempt to deposit non-positive amount: "
102             + amount);
103         }
104         balance = balance + amount;
105     }
106
107     @Override
108     public int hashCode() {
109         return Long.valueOf(acctNo).hashCode();
110     }
111
112     @Override
113     public boolean equals(Object object) {
114         if (!(object instanceof Account)) {
115             return false;
116         }
117         Account other = (Account) object;
118         return this.acctNo == other.acctNo;
119     }
120
121     @Override
122     public String toString() {
123         StringBuilder builder = new StringBuilder("Holder");
124         builder.append("[");
125         addFieldToStringRep(builder, "acctNo",
126                             Long.toString(acctNo));
127         addFieldSeparatorToStringRep(builder);
128         addFieldToStringRep(builder, "balance",
129                             Integer.toString(balance));
130         addFieldSeparatorToStringRep(builder);
131         addFieldToStringRep(builder, "holder",
132                             Long.toString(
133                                  holder.getHolderNo()));
134         builder.append("]");
135         return builder.toString();
```

```
136     }
137
138     private long generateAcctNo() {
139         return acctNoGenerator.nextLong();
140     }
141
142     private void addFieldToStringRep(StringBuilder builder,
143                                      String name,
144                                      String value) {
145         builder.append(name);
146         builder.append(":");
147         builder.append(value);
148     }
149
150     private void addFieldSeparatorToStringRep(
151                  StringBuilder builder) {
152         builder.append(", ");
153     }
154 }
```

Listing 7.22 The `Account` class, which represents a bank account.

Exercise 6

The last class in the system under test is `IllegalBankTransactionException`, see listing 7.23. Exceptions are not tested in this chapter, but one test still is possible without caring about the fact that this class represents an exception. Write a test which verifies that the message passed to the constructor is returned by the method `getMessage`. This might seem like an unnecessary test of a getter, but it's not. The motivation behind writing this test is that the message is handled by the superclass, `java.lang.Exception`. The passing of the string between subclass and superclass is enough logic to motivate a test.

```
1  package se.leiflindback.oodbook.tests.exercises.acct;
2
3  /**
4   * Thrown whenever an attempt is made to perform a transaction
5   * that is not allowed by the bank's business rules.
6   */
7  public class IllegalBankTransactionException extends
8                                                 Exception {
9      /**
10      * Creates a new instance with the specified message.
11      *
12      * @param msg A message explaining why the exception
13      *            is thrown.
```

```
14       */
15      public IllegalBankTransactionException(String msg) {
16          super(msg);
17      }
18  }
```

Listing 7.23 The `IllegalBankTransactionException` class, which is thrown when the bank's business rules are violated.

Exercise 7

This exercise covers testing of a slightly more complex system, which has three different layers, `controller`, `model` and `integration`. The SUT consists of four classes, which are listed below and also found in the accompanying Git repository, [Code], in the packages whose names begin with `se.leiflindback.oodbook.tests.exercises.contr`. The two classes in the `integration` layer, `ClassInInteg` in listing 7.24 and `FileHandler` in listing 7.25, are there just to make the controller a bit more complex. They don't contain any code that can be tested. The class in the model, `ClassInModel`, see listing 7.26, is very simple. It has only one method, which is there to fake some business logic. It only adds two to the parameter and returns the result. The real challenge is to test the controller in listing 7.27. To instantiate the controller you have to first instantiate the two classes in the integration layer, since the controller's constructor expects objects of those classes. Furthermore, the method `createObjInModel` is void and the method `addTwo` can not be executed without first calling `createObjInModel`. Now write tests for `ClassInModel` and `Controller`. Begin with `ClassInModel`, it's normally easier to write tests for lower layers since they have fewer dependencies.

```
1  package
2      se.leiflindback.oodbook.tests.exercises.contr.integration;
3
4  /**
5   * A placeholder for some class in the integration layer.
6   */
7  public class ClassInInteg {
8
9  }
```

Listing 7.24 The class `ClassInInteg`.

```
1  package
2      se.leiflindback.oodbook.tests.exercises.contr.integration;
3
4  /**
5   * A placeholder for some class whose task is to read and
```

```
 * write to a file.
 */
public class FileHandler {
    private String fileName;

    /**
     * Creates anew instance that uses the specified file.
     *
     * @param fileName The file from which to read and write.
     */
    public FileHandler(String fileName) {
        this.fileName = fileName;
    }

}
```

Listing 7.25 The class `FileHandler`.

```
package se.leiflindback.oodbook.tests.exercises.contr.model;

/**
 * A placeholder fo some class in the model.
 */
public class ClassInModel {
    /**
     * Simulates some method that produces a result.
     *
     * @param operand This number is increased by two.
     * @return        <code>operand<code> + 2
     */
    public int addTwo(int operand) {
        return operand + 2;
    }

}
```

Listing 7.26 The class `ClassInModel`.

```
package
   se.leiflindback.oodbook.tests.exercises.contr.controller;

import se.leiflindback.oodbook.tests.exercises.contr.
       integration.ClassInInteg;
import se.leiflindback.oodbook.tests.exercises.contr.
       integration.FileHandler;
```

```
 8 import se.leiflindback.oodbook.tests.exercises.contr.
 9        model.ClassInModel;
10
11 /**
12  * Simulates a controller with two system operations.
13  */
14 public class Controller {
15     private ClassInInteg integObj;
16     private FileHandler fileHandler;
17     private ClassInModel classInModel;
18
19     /**
20      * Creates a new instance using the specified objects.
21      *
22      * @param integObj    Some required object in the
23      *                    integration layer.
24      * @param fileHandler Used for all file access.
25      */
26     public Controller(ClassInInteg integObj,
27                       FileHandler fileHandler) {
28         this.integObj = integObj;
29         this.fileHandler = fileHandler;
30     }
31
32     /**
33      * Initializes the model.
34      */
35     public void createObjInModel() {
36         classInModel = new ClassInModel();
37     }
38
39     /**
40      * Adds two to the specified operand.
41      *
42      * @param operand The number two which two will be added.
43      * @return        <code>operand<code> + 2
44      */
45     public int addTwo(int operand) {
46         return classInModel.addTwo(operand);
47     }
48 }
```

Listing 7.27 The class `Controller`.

Chapter 8

Handling Failure

A program is not complete if it does not handle all possible failures. Some failures can be solved, in order to make the program work despite the exceptional condition causing the failure. In other cases, all that can be done is to report that the operation failed. In any case, the outcome of a system operation shall never be undefined, no matter what happens.

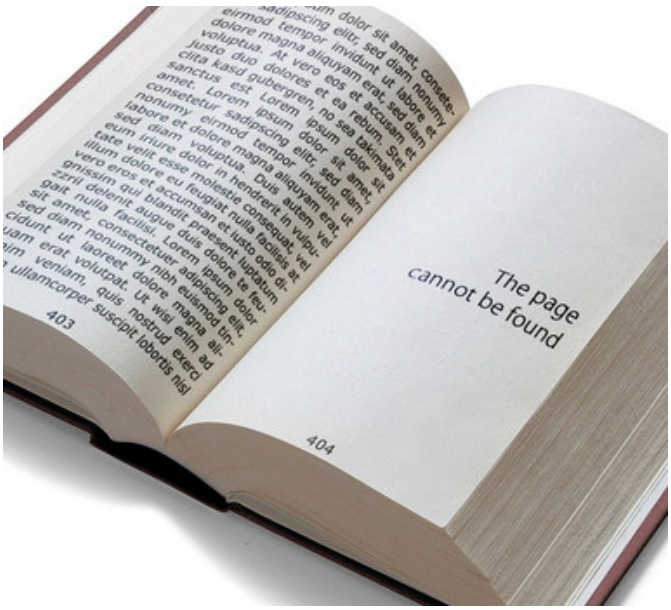


Figure 8.1 A program must handle abnormal situations, giving the user appropriate feedback. Image by FreeImages.com/Alexandre Galant

Many programming languages enables using exceptions for error handling, which is an important mechanism to make the code more flexible and easier to understand. An exception represents an abnormal situation, which disrupts the normal execution of the program. Say, for example, that a method requires a connection to a server to fulfill its task, and that this connection can not be established. This means the method can not do what it is supposed to, and has to return immediately, informing the caller that the abnormal situation *could not connect to server* occurred. The caller then has to switch to error handling, since it did not get any result from the called method. This scenario is quite easily implemented using exceptions. Without exceptions, on the other hand, the failure would have to be reported via return values, which would require messy `if` statements in order to check for possible error codes.

8.1 UML

It is remarkably unclear and difficult to illustrate exception handling in UML. Figure 8.2 shows two possible ways to draw exception handling in a class diagram. The curly brackets in figure 8.2a define a constraint, which means some condition or restriction related to the element where it is placed. The content of a constraint is free text, anything can be written there. This

particular constraint specifies exceptions the method may throw. Figure 8.2b, shows another way to illustrate exceptions in a class diagram, using a reference to the exception class. It has the disadvantage of not showing which method throws the exception.

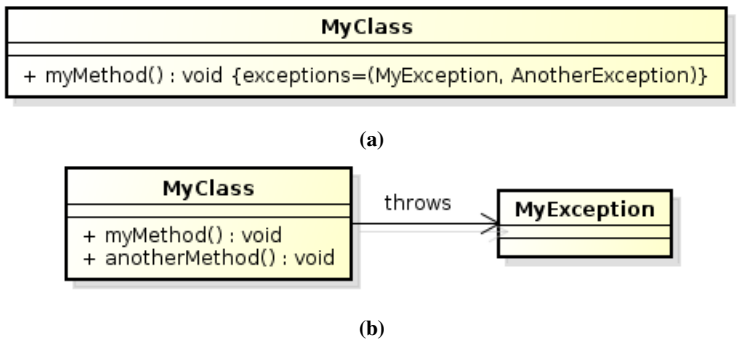


Figure 8.2 Exception handling in class diagram.
(a) using constraint
(b) using reference to exception class

Figure 8.3 illustrates how a sequence and a communication diagram can indicate that an exception is thrown. An open arrow is used, which means the message is asynchronous, and does not follow the normal sequential flow. The stereotype «`exception`» is used to further clarify that the asynchronous message is an exception. A stereotype says that an element belongs to a certain category of such elements. Here, it says that the message belongs to the category *thrown exception*. Notice that, at the blue arrow in the sequence diagram, there is an extra horizontal line in the activation bar. This is because there is a new block starting here, to handle the message symbolizing the exception. Here, this block is not shifted to the right as usual, since the message is asynchronous. Whether it is shifted or not is differs between UML editors, and even between different versions of astah. This block, shifted or not, can be considered to represent the `catch` block. Finally, note the unfortunate fact that there is a parenthesis, `()`, after the exception name in both diagrams. There is no way to omit the parenthesis in astah.

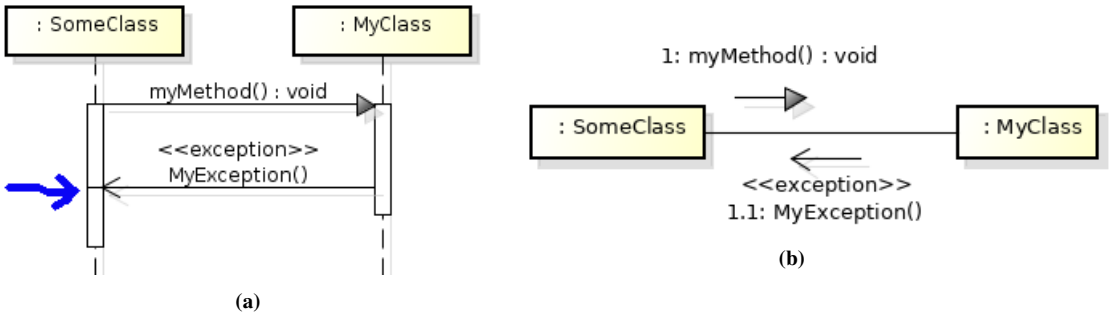


Figure 8.3 exception handling
(a) in sequence diagram, (b) in communication diagram

8.2 Exception Handling Best Practices

This section introduces best practices for exception handling. They should all be considered, and if some of them are not followed, it should be clear why that is the case. The best practices are introduced by implementing failure handling for a particular condition. This condition is when, in the rent car case study, the user tries to book a car, and the booking procedure fails. That can happen either because the car was already booked, or because of a failure in the call to the underlying database.

The `bookCar` system operation is handled by the `bookCar` method in `Controller`, see listing 8.1. To book the car, this method calls `setRentedCar` in `Rental`, listing 8.2, which, in turn, calls `setBookedStateOfCar` in `CarRegistry`, listing 8.3. This last method changes the car's *booked* state. As can be seen in these listings, a car is booked without checking whether it is already booked. This is because no error handling has yet been implemented, but such a check will have to be added now.

```
1  /**
2   * Books the specified car. After calling this method, the car
3   * can not be booked by any other customer. This method also
4   * permanently saves information about the current rental.
5   *
6   * @param car The car that will be booked.
7   */
8  public void bookCar(CarDTO car) {
9      rental.setRentedCar(car);
10     rentalRegistry.saveRental(rental);
11 }
```

Listing 8.1 The `bookCar` method in `Controller`

```
1  /**
2   * Specifies the car that was rented.
3   *
4   * @param rentedCar The car that was rented.
5   */
6  public void setRentedCar(CarDTO rentedCar) {
7      this.rentedCar = rentedCar;
8      carRegistry.setBookedStateOfCar(rentedCar, true);
9  }
```

Listing 8.2 The `setRentedCar` method in `Rental`

```
1  /**
2   * If there is an existing car with the registration number
3   * of the specified car, set its booked property to the
```

```
4   * specified value. Nothing is changed if the car's booked
5   * property already had the specified value.
6   *
7   * @param car         The car that shall be marked as booked.
8   * @param bookedState The new value of the booked property.
9   */
10 public void setBookedStateOfCar(CarDTO car,
11                                 boolean bookedState) {
12     CarData carToBook = findCarByRegNo(car);
13     carToBook.booked = bookedState;
14 }
```

Listing 8.3 The `bookCar` method in `CarRegistry`

Always use exceptions for error handling

This first best practice says that it is never appropriate to report an error with a return value, at least not when using a programming language that includes some kind of exception handling. One reason is that it might be difficult to find appropriate return values. Consider a method that calculates x^y, it might return any number. Therefore, whatever number is assigned to indicate errors, there is the risk that it is confused with a correct result. Another reason to always use an exception to report an error, is that no information about the error can be associated with a primitive return value. Also, if return values are used, it will be necessary to wrap each method call in an `if` statement and check for a return value indicating an error. With exceptions, all error handling is instead located in the `catch` block, separate from the main flow in the `try` block. Last, but maybe most important, using a return value to indicate an error gives low cohesion, since the same variable will be used for two completely different meanings, namely a legal return value, for example the result of calculating x^y, and an error report.

As for the car booking in the case study, this practice says that an exception must be thrown when it is not possible to book the specified car.

Use exceptions only for error handling

This seems obvious, why throw an exception when nothing has gone wrong? There are, however, cases where a bad public interface makes it mandatory to handle exceptions also in a successful execution. As an example, consider a class reading data from an array. It has only one method, `next`, which returns the first unread element. Such a class could be implemented as in listing 8.4. It is impossible to read the contents of the array without catching `ArrayIndexOutOfBoundsException`, since the reader can not tell if the end of the array is reached. A typical client of this class would contain the code in listing 8.5. To get rid of the need for an exception to find that all elements are read, `Iterator`'s public interface must be improved. This can be done by adding the method `hasNext`, which tells if there are more elements. Now `Iterator` looks as listing 8.6, and a typical client as listing 8.7.

This practice does not change our decision to throw an exception when failing to book a car.

```
1  public class Iterator {
2      //It is not relevant how the array is filled.
3      private Object[] theArray;
4      private int cursor = 0;
5
6      public Object next() throws
7          ArrayIndexOutOfBoundsException {
8          return theArray[cursor++];
9      }
10 }
```

Listing 8.4 A class that forces clients to handle exceptions even if nothing is wrong.

```
1  Iterator iter = new Iterator();
2  try {
3      while (true) {
4          iter.next();
5      }
6  } catch (ArrayIndexOutOfBoundsException done) {
7      //All elements are read
8  }
```

Listing 8.5 A client of the class in listing 8.4 is forced to handle exceptions in a successful execution.

```
1  public class Iterator {
2      //It is not relevant how the array is filled.
3      private Object[] theArray;
4      private int cursor = 0;
5
6      public Object next() throws
7          ArrayIndexOutOfBoundsException {
8          return theArray[cursor++];
9      }
10
11     public boolean hasNext() {
12         return cursor < theArray.length;
13     }
14 }
```

Listing 8.6 A better implementation of the iterator.

```
1 Iterator iter = new Iterator();
2 while (iter.hasNext()) {
3     System.out.println(iter.next());
4 }
```

Listing 8.7 A client of the improved iterator in listing 8.6.

Checked or unchecked exception?

Checked exceptions are used for business logic errors. Such errors do not indicate that the program has crashed, but that a business rule was broken. An example is a `withdraw` method in a bank account. If a user tries to withdraw more money than the current balance of the account, the method might throw an exception to indicate that the withdrawal is not allowed. Unchecked exceptions, on the other hand, are used for programming errors. A typical example is `NullPointerException`, which normally means that a method is called on an object that does not exist. This is an indication that there is a bug in the program. Ideally, such exceptions should never occur, except during development and testing. There is also a third category of errors, that are neither bugs, nor business rules. These are errors that stop the program from performing its task, but can not be eliminated during development. A typical example is if the program must call some server to perform its work, and the server is not responding. There is no consensus on which type of exception to use in this case, but it is probably more common to use unchecked exceptions. A reason often mentioned is to avoid cluttering the code with `catch` blocks that can not do much more than report the error. A possible guideline is to use a checked exception if a client can reasonably be expected to recover, in which case it is meaningful to catch the exception. If instead a client cannot do anything to recover, it is not very meaningful to catch the exception, therefore an unchecked exception is to prefer.

This practice says that a checked exception shall be used to indicate that the car is already booked. That is because booking a car that is booked would break the business rule saying *a booked car can not be booked again*. The other failure reason, that something goes wrong in the underlying storage, belongs to the category for which there is no clear advice. Let's decide to use an unchecked exception in this situation, since it is unlikely that anything can be done in response to such an exception, except to report it.

Name the exception after the error condition

An exception class shall have a name that explains what went wrong, in order to clarify why it was thrown. It is also a convention that the name ends with *Exception*. A good example is the class `java.lang.ArrayIndexOutOfBoundsException`, whose name clearly indicates that the program tried to access an index outside an array. This practice might seem to indicate that exception classes should have names explaining in detail which situation the class represents. That is, however, often not the best solution. Very detailed names of exception classes means many different classes will be needed, which in turn makes it mandatory

to mention all these classes in `catch` blocks. Also, more exception classes means a bigger public interface, and thereby a bigger risk of having to change something in it. Therefore, it might be better to have more generic exception classes, and instead provide detailed information about the particular condition in the exception message. Thus, the best solution for exception granularity is a compromise between clear information and low number of classes.

It is now time to decide class names for exceptions used in the book car system operation. First, consider the checked exception thrown when a car is already booked. It could be called something like `AlreadyBookedException`, but that might be too detailed. At the other end of the spectrum is `RentalException`, which could be used for practically any exception in the entire car rental application. A possible compromise is `CarRegistryException`, which could be used for all exceptions thrown by `CarRegistry`, but not by any other class. When deciding this, remember that whatever arrives to the view must tell that the car was already booked, or the cashier would have to say "I can not book your car and I have no clue why". This information can only have two forms, an error code contained in the exception object, or the name of the exception class. Each form has its drawbacks. Using error codes brings the risk of messy `if` statements to decide what went wrong, using class names brings the risk of both increasing the car registry's public interface and requiring many catch statements to decide what went wrong. It is not obvious which option to choose, but let's settle for the class name. Partly because it is not very object-oriented to use `if` statements to check the value of the error code, and partly because, at least this far, we have not introduced a lot of different exception classes. Later, if too many exception classes appear, we might have to change to another solution. The outcome of this discussion is thus to call the class `AlreadyBookedException`. Next, consider the case that something goes wrong in the underlying datastore. Here, `CarRegistryException` is a good name, since there is no point in trying to convey more detailed information about a database failure. All that is of interest to calling layers is that the database operation failed. We are now ready to create the exception classes, see listings 8.8 and 8.9.

```
1 /**
2  * Thrown when trying to book a car which is already booked.
3  */
4 public class AlreadyBookedException extends Exception {
5 }
```

Listing 8.8 The `AlreadyBookedException` is created.

```
1 /**
2  * Thrown when something goes wrong while performing an
3  * operation in the <code>CarRegistry</code>.
4  */
5 public class CarRegistryException extends RuntimeException {
6 }
```

Listing 8.9 The `CarRegistryException` is created.

Include information about the error condition

It is useful, both for debugging purposes and for generating error messages, that an object catching an exception can get information about the cause of the error. Therefore, such information shall be included in the exception object, remember that an exception is an ordinary object, which can have fields storing any kind of information. There are two kinds of information that shall be included in the exception. First, all data that together make up the exceptional condition. Second, a string containing an error message. This message is not intended for the end user. If a message created in the model (or integration) layer is displayed in the user interface, it effectively means that layer contains part of the view, which is definitely not appropriate. Instead, the message is intended for developers or administrators who want to understand what actually happened.

Regarding `AlreadyBookedException`, the exact error condition is known, and detailed information can be provided. The relevant data is the car that could not be booked, which could be stored as a `CarDTO` instance. The error message should state that this car was already booked, it could be *Unable to book <registration number>, since it is already booked.* Regarding `CarRegistryException`, on the other hand, all that is known is that there was a failure in the underlying datastore. It is therefore not possible to be more specific than to provide an error message stating this.

Use functionality provided in `java.lang.Exception`

All exceptions must inherit `java.lang.Exception` (consult section 1.4 if you need to repeat inheritance). In doing that, a lot of useful functionality is inherited, for example handling an error message. The `Exception` class has both a constructor that takes a message, and a method `getMessage` that returns the message.

Knowing the above, it is very easy to add message handling to the exception classes. All that is needed is to add a constructor that takes the message and passes it to the superclass' constructor. There is no need to write a get method to retrieve the message, since that method is inherited from `Exception`. After this update, the classes look like in figures 8.10 and 8.11. Note that the message is handled differently in these classes. In `CarRegistryException`, since nothing is known about the message, there is nothing to do except to store it in the object. `AlreadyBookedException`, on the other hand, represents only one particular condition, and the message will always be the same. Therefore, it can be encapsulated in the exception class. There is no reason to make it customizable, since that would only force classes throwing the exception to bother about the message, which would give them lower cohesion. The only thing that differs between different instances of `AlreadyBookedException` is which car was being booked. This is specified in the `carThatCanNotBeBooked` parameter in the constructor.

```
1 /**
2  * Thrown when trying to book a car which is already booked.
3  */
4 public class AlreadyBookedException extends Exception {
5     private CarDTO carThatCanNotBeBooked;
6
```

```
7       /**
8        * Creates a new instance with a message specifying for
9        * which car the booking failed.
10       *
11       * @param carThatCanNotBeBooked The car that could not be
12       *                              booked.
13       */
14      public AlreadyBookedException(
15              CarDTO carThatCanNotBeBooked) {
16          super("Unable to book " +
17                carThatCanNotBeBooked.getRegNo() +
18                ", since it is already booked.");
19          this.carThatCanNotBeBooked = carThatCanNotBeBooked;
20      }
21
22      /**
23       * @return The car that could not be booked.
24       */
25      public CarDTO getCarThatCanNotBeBooked() {
26          return carThatCanNotBeBooked;
27      }
28
29  }
```

Listing 8.10 Message handling is added to `AlreadyBookedException`.

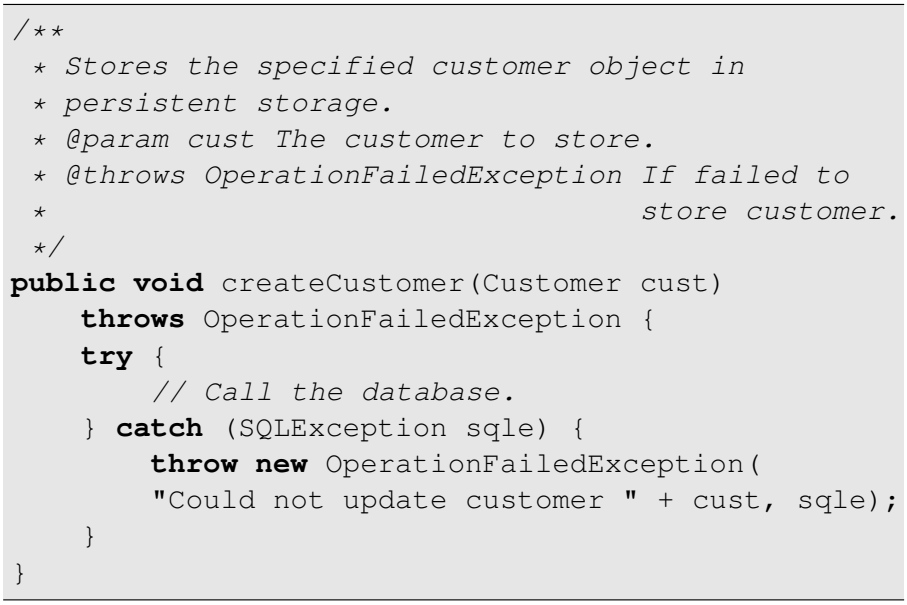
```
1   /**
2    * Thrown when something goes wrong while performing an
3    * operation in the <code>CarRegistry</code>.
4    */
5   public class CarRegistryException extends RuntimeException {
6       /**
7        * Creates a new instance representing the condition
8        * described in the specified message.
9        *
10       * @param msg A message that describes what went wrong.
11       */
12      public CarRegistryException(String msg) {
13          super(msg);
14      }
15  }
```

Listing 8.11 Message handling is added to `CarRegistryException`.

Use the correct abstraction level for exceptions

Say that a low-level layer (far from the view) throws some exception with a detailed technical description of an error condition. An example could be that the integration layer fails to perform a database operation. Should this information be displayed to the user? Most certainly not. First, it is not user friendly at all. What would you think if a cash machine said "sql syntax error" when you tried to withdraw money? It would not be possible to know if the amount had been withdrawn from the bank account, and it would probably not feel safe to use that bank any more. Second, it could create security problems if detailed information about the database was displayed to all users, including possible attackers. Now that it has been established that this exception is not of interest to the user, the next question is if it then is of any interest to the view layer? Most likely not, since anyway it shall not be displayed to the user. Also, catching a database exception in the view creates a dependency from the view to the integration layer, which is not desired. The conclusion is that it is often inappropriate that an exception traverses many layers, and better to do as in listing 8.12, where the exception from the database call is caught, and a more generic exception is thrown instead. Note that the SQLException object, `sqle`, is included in the thrown `OperationFailedException` object (line 15). That makes it possible for higher layers to retrieve the original `SQLException` object if necessary, by calling the method `getCause`. This is done on line five in listing 8.13. Listing 8.14 shows the `OperationFailedException` class.

However, note that it is not a requirement to catch an exception in each layer. Sometimes an exception *does* make sense also to the next higher layer. In that case it is best to let the exception propagate up without catching it.



```
1  /**
2   * Stores the specified customer object in
3   * persistent storage.
4   * @param cust The customer to store.
5   * @throws OperationFailedException If failed to
6   *                                  store customer.
7   */
8  public void createCustomer(Customer cust)
9      throws OperationFailedException {
10     try {
11         // Call the database.
12     } catch (SQLException sqle) {
13         throw new OperationFailedException(
14         "Could not update customer " + cust, sqle);
15     }
16 }
```

Listing 8.12 An exception that is too detailed for higher layers is caught, and a more appropriate exception is thrown instead.

```
1 public void aMethodThatMustStoreACustomer() {
2     try {
3         storage.createCustomer(customer);
4     } catch (OperationFailedException exc) {
5        exc.getCause(); // Returns the original exception.
6     }
7 }
```

Listing 8.13 How to use the method `getCause()`. The `createCustomer` method, called on line three, is the method in listing 8.12.

```
1 public class OperationFailedException
2     extends Exception {
3
4     public OperationFailedException(String msg,
5                                     Exception cause) {
6         super(msg, cause);
7     }
8 }
```

Listing 8.14 The class OperationFailedException.

We now have to decide if `CarRegistryException` is appropriate for all layers, or if it shall be wrapped in a more generic exception at some point. It is currently thrown by the method `setBookedStateOfCar` in the integration layer, which is called by the method `setRentedCar` in `Rental`, in the model. That method, in turn, is called by `bookCar` in the controller. Does it make sense to let a `CarRegistryException` propagate up to the controller? Maybe yes, since the controller anyway uses the `CarRegistry` class. That means no new dependency is introduced if `CarRegistryException` continues up to the controller. On the other hand, the code in `bookCar` in `Controller` currently does not depend on the fact that `setRentedCar` calls `CarRegistry`. Now, if a call to `setRentedCar` requires catching `CarRegistryException`, it also means that code must be changed if later `Rental` is changed, to *not* call `CarRegistry` any more. It is not easy to decide if this problem is big enough to motivate the introduction of a new exception class, that should be thrown by `Rental`. Let's choose the simplest solution, and not introduce a new exception, but instead let `CarRegistryException` propagate up to the controller. If later there really is the need to change the controller because of a change in `setRentedCar` in `Rental`, we might reconsider this decision. Thus, the changes to `setRentedCar` are to add the throws clause on line 18 in listing 8.15, to change the comment to clarify that this method books the car, and to check if the car is already booked. The private method `bookCar` is added, in order to clarify that it is here that the car is booked, and to give higher cohesion to `setRentedCar`. In fact, it might be bad design to have a set method perform the business logic of booking the car. A set method is normally expected to just store the specified value, and nothing more. Let's change the method name to `rentCar`, to avoid this confusion. Also, as can be seen on line 17, `CarDTO` has got a new method, `isBooked`, which tells if the car is booked. Furthermore, a new `CarRegistry`

method, called `getCarByRegNo`, is called on line 16. It returns a `CarDTO` instance with the current state of the car whose registration number is specified in the `CarDTO` parameter.

```
1  /**
2   * Specifies the car that was rented. The specified car is
3   * also booked, thus becoming unavailable to other rentals.
4   *
5   * @param rentedCar The car that was rented.
6   */
7  public void rentCar(CarDTO rentedCar)
8                      throws AlreadyBookedException {
9      bookCar(rentedCar);
10     this.rentedCar = rentedCar;
11 }
12
13 private void bookCar(CarDTO carToBook)
14                       throws AlreadyBookedException {
15     CarDTO currentCarState =
16         carRegistry.getCarByRegNo(carToBook);
17     if (currentCarState.isBooked()) {
18         throw new AlreadyBookedException(currentCarState);
19     }
20     carRegistry.setBookedStateOfCar(carToBook, true);
21 }
```

Listing 8.15 The `setRentedCar` method in `Rental` does not catch the `CarRegistryException`.

We are not done yet. The `CarRegistryException` is now in `bookCar` in the controller, and there the same question again arises. Shall the exception be caught, or again allowed to propagate upwards, this time to the view? Now, the answer is a definite *"no"*. It is never a good design to let the view depend on the integration layer. Therefore, the exception is caught in the controller. Still, the view must be informed of the fact that the car could not be booked. That means a new exception must be thrown by the `Controller` method `bookCar`. This exception can be very generic, in order to be appropriate whenever any operation fails, but the exact reason is not of interest to the view. Let's use the `OperationFailedException`, already introduced in listing 8.14. The `Controller` method `bookCar` is shown in listing 8.16.

```
1  /**
2   * Books the specified car. After calling this method, the car
3   * can not be booked by any other customer. This method also
4   * permanently saves information about the current rental.
5   *
6   * @param car The car that will be booked.
7   */
```

```
 8 public void bookCar(CarDTO car) throws AlreadyBookedException,
 9                                        OperationFailedException {
10     try {
11         rental.rentCar(car);
12         rentalRegistry.saveRental(rental);
13     } catch(CarRegistryException carRegExc) {
14         throw new OperationFailedException(
15                                     "Could not rent the car.",
16                                     carRegExc);
17     }
18 }
```

Listing 8.16 The `bookCar` method in `Controller`, when exception handling has been added.

This long discussion only treated `CarRegistryException`. We have not yet started to consider `AlreadyBookedException`, but luckily, that is quite easy. Since the very reason it was created was to convey information to the view, it most certainly shall propagate all the way up to the view. As can be seen in listing 8.16, it is not caught in the controller, but is instead specified in the `throws` clause, on line eight.

Write Javadoc comments for all exceptions

Whenever you write a method that throws an exception, also add a Javadoc comment that describes the condition under which the exception is thrown. Such a comment is of great use for someone trying to understand what may have happened when program execution results in the exception being thrown. One more advantage of writing a comment, is that it forces the author to really decide when the method shall throw the exception. The Javadoc tag for documenting an exception is `@throws`.

Turning to the book car case study, there are by now many methods that throw exceptions. Starting from the lowest layer, we first consider the methods in `CarRegistry`. There are two methods in that class which throw `CarRegistryException`, namely `setBookedStateOfCar` and `getCarByRegNo`. They throw the exception if the call to the datastore fails, which will never happen since there is no datastore, or if the car can not be found. One might ask why the exception is thrown when the car can not be found, is that really a failure? There is no unarguable answer to this question. The reason for letting them throw the exception is that they are supposed to perform some operation on an existing car, not to check if that car exists. On the other hand, `findAvailableCar`, which searches for a car matching a certain description, does not throw an exception if there is no matching car. This method is not supposed to handle an existing car, but instead check if there is a certain car. Therefore, it is not a failure if the car does not exist, but instead a valid outcome of the method. The javadoc comments of `setBookedStateOfCar` and `getCarByRegNo` are found in listing 8.17. The `@throws` tags are located on lines 9 and 25. Note that there is no `throws` clause in the method declarations, since `CarRegistryException` is an unchecked exception.

```
1  /**
2   * If there is an existing car with the registration number of
3   * the specified car, set its booked property to the specified
4   * value. Nothing is changed if the car's booked property
5   * already had the specified value
6   *
7   * @param car         The car that shall be marked as booked.
8   * @param bookedState The new value of the booked property.
9   * @throws CarRegistryException if the database call failed,
10  *                              or if the specified car did
11  *                              not exist.
12  */
13 public void setBookedStateOfCar(CarDTO car,
14                                  boolean bookedState) {
15     ...
16 }
17
18 /**
19  * Searches for a car with the registration number of the
20  * specified car.
21  *
22  * @param searchedCar Searches for a car with registration
23  *                    number of this object.
24  * @return A car with the specified registration number.
25  * @throws CarRegistryException if the database call failed,
26  *                              or if the specified car did
27  *                              not exist.
28  */
29 public CarDTO getCarByRegNo(CarDTO searchedCar) {
30     ...
31 }
```

Listing 8.17 The Javadoc comments for the methods `setBookedStateOfCar` and `getCarByRegNo`.

Continuing upwards through the layers, the next class is `Rental`, and the method is `rentCar`. This method does not catch a `CarRegistryException` coming from the car registry. Therefore, the same exception is thrown also by this method, and must therefore be documented. Also `AlreadyBookedException` is included in the Javadoc comment (listing 8.18), since it is thrown by this method.

```
1  /**
2   * Specifies the car that was rented. The specified car is
3   * also booked, thus becoming unavailable to other rentals.
4   *
5   * @param rentedCar The car that was rented.
```

```
6   * @throws AlreadyBookedException if the car was already
7                                      booked.
8   * @throws CarRegistryException if the database call failed,
9   *                               or if the specified car did
10  *                               not exist.
11  */
12 public void rentCar(CarDTO rentedCar)
13                      throws AlreadyBookedException {
14     ...
15 }
```

Listing 8.18 The Javadoc comment for the method `rentCar`.

The last method to document is `bookCar` in the controller. `AlreadyBookedException` just passes through this method, and its comment is identical to that in `rentCar`, since it makes perfect sense also here. `CarRegistryException` is not thrown by this method, but is wrapped in an `OperationFailedException`, which tells that the book car operation could not be performed, but the reason is unknown. The resulting Javadoc comment is in listing 8.19.

```
1  /**
2   * Books the specified car. After calling this method, the
3   * car can not be booked by any other customer. This method
4   * also permanently saves information about the current
5   * rental.
6   *
7   * @param car The car that will be booked.
8   * @throws AlreadyBookedException   if the car was already
9   *                                  booked.
10  * @throws OperationFailedException if unable to rent the car
11  *                                  for any other reason than
12  *                                  it being already booked.
13  */
14 public void bookCar(CarDTO car) throws AlreadyBookedException,
15                                  OperationFailedException {
16     ...
17 }
```

Listing 8.19 The Javadoc comment for the method `bookCar`.

An object shall not change state if it throws an exception

It is desirable that an object can still be used after it has thrown an exception. Things would be quite complicated if it was necessary to discard and recreate all objects involved in each operation that resulted in an exception. If an object shall be usable, it must be clear which

state it is in, that is, which are the values of all its data elements. Throwing an exception means the method failed, and could not perform its task. Thus, the only reasonable state of an object after a method has thrown an exception, is the state the object had before the method was called. Therefore, no field shall change value if an exception is thrown. There are many ways to reach this goal, the most common follow below, sorted more or less in the order they shall be chosen, with the most preferred strategy first.

Make the object immutable An immutable object is an object that can never change state. It is a programmers best friend, since there is never any doubt about its state, and no risk that it is accidentally changed. To make an object immutable, all of its fields must be final. Also, to be really rigid, it must not be possible to inherit the class, and change its immutable behavior. Listing 8.20 contains an immutable example, `Person`. Note that the object `address` can not just be stored on line 15 in the constructor. It is necessary to make a copy of it, otherwise also the object that created the `Person` would have a reference to the same `Address` object, and be able to change it. Alternatively, also `Address` could be immutable, in which case it would not be necessary to create the copy.

```
1  /**
2   * Objects of this class are immutable, none of
3   * the fields can ever change state.
4   */
5  public final class Person {
6      private final String name;
7      private final Address address;
8
9      /**
10      * Creates an instance with the specified
11      * name and address.
12      */
13     public Person(String name, Address address) {
14         this.name = name;
15         this.address = new Address(address);
16     }
17 }
18
19 /**
20  * Represents an address.
21  */
22 public class Address {
23     private String street;
24     // More fields.
25
26     /**
27      * Creates an instance with all fields equal
```

```
28       * to the fields of the specified address.
29       */
30      Address(Address address) {
31          this.street = address.street;
32          // Copy all other fields.
33      }
34  }
```

Listing 8.20 Objects of `Person` are immutable, their fields can never change value.

Check parameter values before changing any state A common cause of an exception is an illegal parameter value. Therefore, it is a very good practice to let each method check for illegal parameter values first, before performing its work. Consider for example the `withdraw` method in listing 8.21, which throws `OverdraftException` when the amount to withdraw is larger than the balance. Since the first thing it does is to check the parameter value (line 15), there is no risk that the state, which in this case is the balance, is changed when the exception is thrown.

```
1  /**
2   * Represents a bank account.
3   */
4  public class Account {
5      private int balance;
6
7      /**
8       * Withdraws the specified amount from this
9       * account.
10      *
11      * @param amount The amount to withdraw.
12      */
13     public void withdraw(int amount) throws
14                              OverdraftException {
15         if (amount > balance) {
16             throw new OverdraftException(balance,
17                                          amount);
18         }
19         balance = balance - amount;
20     }
21 }
```

Listing 8.21 This object checks if the amount to withdraw is illegal, before it updates the balance.

Place operations that might fail before operations that alter the state This strategy can be used if it is not possible to validate the parameters without performing a part of the methods work. Consider, for example, a method that shall insert a new element in a sorted collection of unique elements. The best way to check if the new element is unique is to find the location where it shall be inserted. This search will fail if the element is not unique, but the collection will not be updated at that point.

Use a temporary copy of the state If none of the above strategies can be used, we can save a temporary copy of the state and make sure to restore it before throwing an exception.

Now let's check if all exception throwing methods in the case study follow this practice. Starting in `CarRegistry`, the methods to consider are `getCarByRegNo` and `setBookedStateOfCar`. The first is not of interest, since it does not change any state at all. The second must check if the car exists before changing its booked state. This is an example where the strategy *Place operations that might fail before operations that alter the state* is appropriate. To be able to change the state of the car, we must find it in the datastore. If it is not found, the state can not be changed, but instead an exception is thrown. Continuing to `Rental`, the method `rentCar` uses the strategy *Check parameter values before changing any state*. The first thing the method does is to check that the specified car exists, the second is to check its booked status. Only if the car exists and is not yet booked, is the state updated. If it does not exist, or is already booked, an exception is thrown. Finally, in `Controller`, `bookCar` does never change any state.

Never write an empty `catch` block

An empty `catch` block means an exception is completely ignored, which is very seldom a good thing. If an exception is thrown in the `try` block, execution will continue in the empty `catch` block, where of course, nothing happens. Execution then continues on the first line below the `catch` block, as if no exception had been thrown. There will be no notification whatsoever about the exception, which can make it very difficult to understand the program's behavior.

The exceptions in the case study are caught in the view, which we have not yet considered. Actually, there is no real view, but just hard coded calls to the system operations in the controller. Still, since this fake view simulates an interaction with a real user, it can include error handling in the simulation. The call to the method that might throw an exception, namely `bookCar`, can be seen in the (slightly shortened) listing of the user interaction simulation in listing 8.22. As stated above, the `catch` blocks shall not be empty, but what shall happen there? Normally, there are two things to consider, information to users and information to developers and/or administrators. First, however, there are a few things worth noting about listing 8.22. One thing is that the `try` block does not contain only the call to `bookCar`, which throws exceptions, but also the rest of the method simulating the user interaction. This is because if any line fails, there is no point to go on with the simulation.

Instead, execution immediately switches to error handling in the `catch` block. Next, not only `AlreadyBookedException` and `OperationFailedException` are caught, but also `java.lang.Exception`, which is the superclass of all exceptions. This way, any exception, thrown on any line, will be caught and handled. No exception will ever leave our code. If it did, the result would be some uncontrolled and undesired stack trace in the user interface. Last, even though we have not yet decided what to write in the catch blocks, they are still not empty. Even the most badly designed and temporary program can at least call `printStackTrace` to show what exception was thrown. Now, however, it is time to improve error notification, that is the topic of the following two practices.

```
1  try {
2      CarDTO availableCar = new CarDTO(null, new Amount(1000),
3                                       "medium", true, true,
4                                       "red", false);
5
6      foundCar = contr.searchMatchingCar(availableCar);
7      System.out.println("Result of searching for available " +
8                         "car: " + foundCar);
9
10     AddressDTO address = new AddressDTO("Storgatan 2",
11                                         "12345", "Hemorten");
12     DrivingLicenseDTO drivingLicense =
13                     new DrivingLicenseDTO("982193721937213");
14     CustomerDTO customer = new CustomerDTO("Stina", address,
15                                            drivingLicense);
16     contr.registerCustomer(customer);
17     System.out.println("Customer is registered");
18
19     contr.bookCar(foundCar);
20     System.out.println("Car is booked");
21     Amount paidAmount = new Amount(1500);
22     System.out.println("----- Receipt follows -----");
23     contr.pay(paidAmount);
24     System.out.println("----- End of receipt -----");
25 } catch (AlreadyBookedException exc) {
26     exc.printStackTrace();
27 } catch (OperationFailedException exc) {
28     exc.printStackTrace();
29 } catch (Exception exc) {
30     exc.printStackTrace();
31 }
```

Listing 8.22 The `try-catch` block in the view.

How to notify the user?

The user interface must always reflect the state of the system, and clearly show what is happening. If the user has initiated a system operation, and that operation can not be successfully completed, the user interface must show this. As has been discussed previously, it is not always adequate to show very detailed error messages, for example containing database error codes. It is, instead, necessary to carefully consider what is the most appropriate error message, specifying what the user needs to know, without creating confusion. It is also important that information is always given consistently. The user must not be surprised or confused by getting different messages from different parts of the system. Messages shall always have similar format, and two similar messages must mean the same thing. As a consequence of this reasoning, it is a common design choice to have only one component generating error messages. This is also the best solution in terms of cohesion and encapsulation. Do not spread similar error handling code over many `catch` blocks, but instead just call the component responsible for error messages, and encapsulate user interface handling in there. Since error messages are a part of the user interface, this error message component is placed in the view.

The case study does not have a real user interface, but we can still create a class that prints error messages to `System.out`, see listing 8.23. It is called from the `catch` blocks, as in listing 8.24. Since there was no need for a unique error message in answer to an `OperationFailedException`, it is no longer caught explicitly, but is handled by the `catch` block for any `java.lang.Exception`. Note that the error message is created in the view, it is not the string returned by `getMessage` in `Exception`, since that would have meant some lower layer decided what to print in the user interface.

```
1  /**
2   * This class is responsible for showing error messages to
3   * the user.
4   */
5  public class ErrorMessageHandler {
6
7      /**
8       * Displays the specified error message.
9       *
10      * @param msg The error message.
11      */
12     void showErrorMsg(String msg) {
13         StringBuilder errorMsgBuilder = new StringBuilder();
14         errorMsgBuilder.append(createTime());
15         errorMsgBuilder.append(", ERROR: ");
16         errorMsgBuilder.append(msg);
17         System.out.println(errorMsgBuilder);
18     }
19
20     private String createTime() {
21         LocalDateTime now = LocalDateTime.now();
```

```
22          DateTimeFormatter formatter = DateTimeFormatter.
23                          ofLocalizedDateTime(FormatStyle.MEDIUM);
24          return now.format(formatter);
25      }
26  }
```

Listing 8.23 The class `ErrorMessageHandler`, which is responsible for showing error messages to the user.

```
1  } catch (AlreadyBookedException exc) {
2      errorMsgHandler.showErrorMsg(
3                      exc.getCarThatCanNotBeBooked().getRegNo() +
4                      " is already booked.");
5  } catch (Exception exc) {
6      errorMsgHandler.showErrorMsg("Failed to book, please" +
7                                   " try again.");
8  }
```

Listing 8.24 The calls to `ErrorMessageHandler`.

How to notify developers?

It's not enough to inform the user about the application's state. It must also be possible for developers to check if there are bugs. This requires more detailed reports about exceptions than those shown to users, since the developer must be able to see exception stack traces. These are printed to the *log*, which can be for example a text file. There is normally one dedicated component, which is responsible for creating log entries. The reasons for creating such a component are exactly the same as for creating the component generating user interface error messages. That is, messages shall be similar no matter why or where they are caused, log handling shall not be duplicated in many different locations, and detailed knowledge about the logging procedure shall be encapsulated in one component.

Which exceptions shall be logged? A developer looking for bugs is interested in exceptions indicating that the program is not working as intended, but has no interest in exceptions which are part of normal program flow. The conclusion is that *all* exceptions, *except* those representing business logic errors, shall be logged to the log file. A common practice to achieve this is to first write `catch` blocks for all business logic exceptions, and below those write a `catch` block for `java.lang.Exception`, which will catch all remaining exceptions, not representing business logic errors. This way we can be sure not to miss any exception, since `java.lang.Exception` is the superclass of all exception classes. Note that this practice applies to all catch blocks for exceptions *not* representing business logic, in all classes. Whenever such an exception is caught, it shall be logged.

The case study logs to a file called `rentcar-log.txt`. The class writing logs, listing 8.25, is placed in a `util` package, since it's useful wherever a log message shall be written. There

already exists logging APIs, for example the one in `java.util.logging` that format messages, include time and origin, and manage log files. Normally, such an API should be used, instead of, as is the case here, creating a new class performing all these tasks. The only reason to rewrite the functionality here, instead of using an existing API, is to keep this case study shorter, and not force the reader to learn a new API. Example calls of the logging component can be seen in listing 8.26. The first `catch` block, handling `AlreadyBookedException`, only shows the error message on the user interface, since trying to book a car that is already booked is a violation of a business rule, not an indication that the program has a bug. The other `catch` block, handling `java.lang.Exception`, calls the private method `writeToLogAndUI`, which also prints to the log file.

```
1  package se.leiflindback.oodbook.rentcarWithExceptions.util;
2
3  /**
4   * This class is responsible for the log.
5   */
6  public class LogHandler {
7      private static final String LOG_FILE_NAME =
8                                  "rentcar-log.txt";
9      private PrintWriter logFile;
10
11     public LogHandler() throws IOException {
12         logFile = new PrintWriter(
13                   new FileWriter(LOG_FILE_NAME), true);
14     }
15
16     /**
17      * Writes a log entry describing a thrown exception.
18      *
19      * @param exception The exception that shall be logged.
20      */
21     public void logException(Exception exception) {
22         StringBuilder logMsgBuilder = new StringBuilder();
23         logMsgBuilder.append(createTime());
24         logMsgBuilder.append(", Exception was thrown: ");
25         logMsgBuilder.append(exception.getMessage());
26         logFile.println(logMsgBuilder);
27         exception.printStackTrace(logFile);
28     }
29
30     private String createTime() {
31         LocalDateTime now = LocalDateTime.now();
32         DateTimeFormatter formatter = DateTimeFormatter.
33                         ofLocalizedDateTime(FormatStyle.MEDIUM);
34         return now.format(formatter);
35     }
```

```
36 }
```

Listing 8.25 The class `LogHandler`, which is responsible for logging.

```
1  } catch (AlreadyBookedException exc) {
2      errorMsgHandler.showErrorMsg(exc.getCarThatCanNotBeBooked()
3                                       .getRegNo()
4                                   + " is already booked.");
5  } catch (Exception exc) {
6      writeToLogAndUI("Failed to book, please try again.", exc);
7  }
8
9  private void writeToLogAndUI(String uiMsg, Exception exc) {
10     errorMsgHandler.showErrorMsg(uiMsg);
11     logger.logException(exc);
12 }
```

Listing 8.26 The calls to `LogHandler`.

Write unit tests for the exception handling

Exception handling must of course be unit tested, just like everything else. Things to test are that exceptions are not thrown during successful execution, that they are thrown when failures occur, that messages and other content of exception objects are correct, and that they are handled as intended in `catch` blocks.

As an example of unit testing exceptions, consider the class `CarRegistry` in the integration layer. The exception handling tests of `CarRegistry` calls `getCarByRegNo` and `setBookedStateOfCar` with a car that does not exist. `CarRegistryException` shall be thrown in both cases. Exception handling tests for `Rental` and `Controller` are the same, to rent a car that is already booked, and to rent a car that does not exist. Finally, there are tests also for the two classes responsible for error messages in the user interface and in the log file. The tests for `Rental` can be found in listing 8.27, as an example of exception handling unit tests. The complete tests are included in the accompanying NetBeans project [Code].

```
1  @Test
2  public void testRentBookedCar() {
3      CarRegistry carReg = new RegistryCreator().
4                               getCarRegistry();
5      Rental instance = new Rental(null, carReg);
6      CarDTO rentedCar = new CarDTO("abc123", new Amount(1000),
7                                    "medium", true,
8                                    true, "red", false);
9      try {
10         carReg.setBookedStateOfCar(rentedCar, true);
```

```
11          instance.rentCar(rentedCar);
12          fail("Could rent a booked car.");
13      } catch (AlreadyBookedException ex) {
14          assertTrue("Wrong exception message, does not " +
15                     "contain specified car: " +ex.getMessage(),
16                     ex.getMessage().contains(
17                                       rentedCar.getRegNo()));
18          assertTrue("Wrong car is specified: " +
19                     ex.getCarThatCanNotBeBooked(),
20                     ex.getCarThatCanNotBeBooked().getRegNo().
21                        equals(rentedCar.getRegNo()));
22      }
23  }
24
25  @Test
26  public void testRentCarThatDoesNotExist() throws
27                                        AlreadyBookedException {
28      CarRegistry carReg =
29                  new RegistryCreator().getCarRegistry();
30      Rental instance = new Rental(null, carReg);
31      CarDTO rentedCar = new CarDTO("wrong", new Amount(1000),
32                                    "medium", true,
33                                    true, "red", false);
34      try {
35          instance.rentCar(rentedCar);
36          fail("Could rent a non-existing car.");
37      } catch (CarRegistryException exc) {
38          assertTrue("Wrong exception message, does not " +
39                     " contain specified car: "
40                     + exc.getMessage(),
41                     exc.getMessage().contains(
42                                       rentedCar.toString()));
43      }
44  }
```

Listing 8.27 Unit testing the exception handling in `Rental`.

8.3 Complete Exception Handling in the Case Study

It is not obvious where and when to use exceptions in the rent car case study, since the code is still quite short and incomplete. For example, should all methods in `CarRegistry` throw `CarRegistryException`, to simulate database failures, even though there is no database yet? In this implementation, an exception is thrown only when there are obvious failures, in order to keep the case study reasonably short and clear. In fact, there are only two more

situations where exceptions must be used, besides the car booking described above.

The first situation requiring an exception is when `LogHandler` fails to open the log file. In this case, an `IOException` is thrown. Since this happens during startup, the exception is caught in `main` It is an open question if the program shall be allowed to continue without logging. Currently, it is terminated if the log handler fails.

The second situation where an exception is needed, is due to the fact that the controller operations `registerCustomer`, `bookCar`, and `pay`, must be called in that order. The reason why this order is required is that `registerCustomer` creates the rental object used in all three methods, therefore a `NullPointerException` will be thrown if one of the other methods is called first. Also, since each call to `registerCustomer` creates a new `Rental`, any ongoing rental will be lost when that method is called. It can of course be questioned if this implementation is correct, but that is really a question about business rules, which can only be answered by a domain expert. In the current implementation, an `IllegalStateException` is thrown if these operations are called in any other order. This is the exception usually used when methods are called in wrong order.

The implementation of these scenarios, together with its unit tests, can be found in the NetBeans project in GitHub [Code]. None of the code is listed here, since it contains nothing conceptually new, and no particularly tricky issues.

8.4 Common Mistakes

Even when following all the best practices mentioned in section 8.2, it is still possible to make mistakes. Below are two common mistakes.

Failure handling is missing The fact that the program works does not necessarily mean that failure handling is correct. It is a common mistake not handle all possible failures, for example not to check that all parameters in a method have valid values. A good way to discover this flaw is to write extensive unit tests. Unit tests shall check all possible parameter values, and all possible execution paths.

NO!

Exceptions are caught when it is not needed Do not catch an exception if nothing is done in the `catch` block. It is quite common to misunderstand the practice saying *Use the correct abstraction level for exceptions*, and always catch all exceptions in all methods. The result is code similar to listing 8.28, where, on lines four to six, the exception is caught just to be rethrown. If the exception is appropriate also in the calling method, then it is best to just let it propagate further up in the call stack, as in listing 8.29.

```
1 public void myMethod() throws MyException {
2     try {
3         methodThatThrowsMyException();
4     catch(MyException exc) {
5         throw exc;
6     }
7 }
```

Listing 8.28 There is no point in catching an exception just to immediately rethrow it, as in this listing.

```
1 public void myMethod() throws MyException {
2     methodThatThrowsMyException();
3 }
```

Listing 8.29 If the exception makes sense also in the calling method, it can continue to that method.

Chapter 9

Polymorphism and Inheritance

All design discussions up to this point have been based on the fundamental concepts coupling, cohesion and encapsulation. It is now time to introduce two more concepts, *polymorphism* and *inheritance*, and design solutions based on those. Especially polymorphism, but also inheritance, are very powerful tools for creating a design. Still, the quality of these, more advanced, designs will be completely decided by the criteria coupling, cohesion and encapsulation. The only value of polymorphism and inheritance is to make the design meet those criteria to a higher extent.

It is not possible to understand this chapter without a basic understanding of the concepts interface, implementation and inheritance. **Make sure you fully understand sections 1.7 and 1.8 before reading this chapter.**

!

9.1 UML

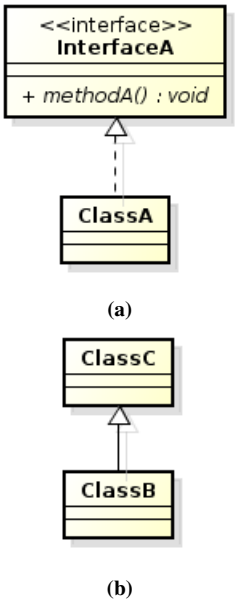


Figure 9.1 Class diagram, illustrating:
(a) interface and implementation
(b) inheritance

In order to design polymorphism and inheritance, it is necessary to model interfaces, implementations and inheritances in UML. How do this in a class diagram is illustrated in figure 9.1, where `ClassA` *implements* `InterfaceA`, and `ClassB` *inherits* `ClassC`. There are a few subtle things worth emphasizing here. First, the stereotype «`interface`» is used to tell that `InterfaceA` in an interface and not a class. A stereotype is a categorization. Here, it says that `InterfaceA` belongs to the category *interfaces*. This stereotype is the only difference between the symbols for class and interface. Another issue is that the word *implementation* does not exist in UML, instead, the same thing is called *realization*. Also, the word *inheritance* is very vaguely specified and not much used. Instead, a superclass is said to be a *generalization* of a subclass. Furthermore, note that the method `methodA` in the interface is written in italics. This means it is abstract, it consists only of a declaration and does not have any body.

It is an unfortunate fact that there is now way to illustrate implementation or inheritance in a sequence or

a communication diagram. If, for example, an object of `ClassA` in figure 9.1a is to be used in a sequence diagram, it must be of either the type `ClassA` (figure 9.2a) or `InterfaceA` (figure 9.2b). It is impossible to show that the object has both types. It is in fact a direct error to draw both types, as in figure 9.2c. That would mean there were two different objects, one of type `InterfaceA` and one of type `ClassA`.

NO!

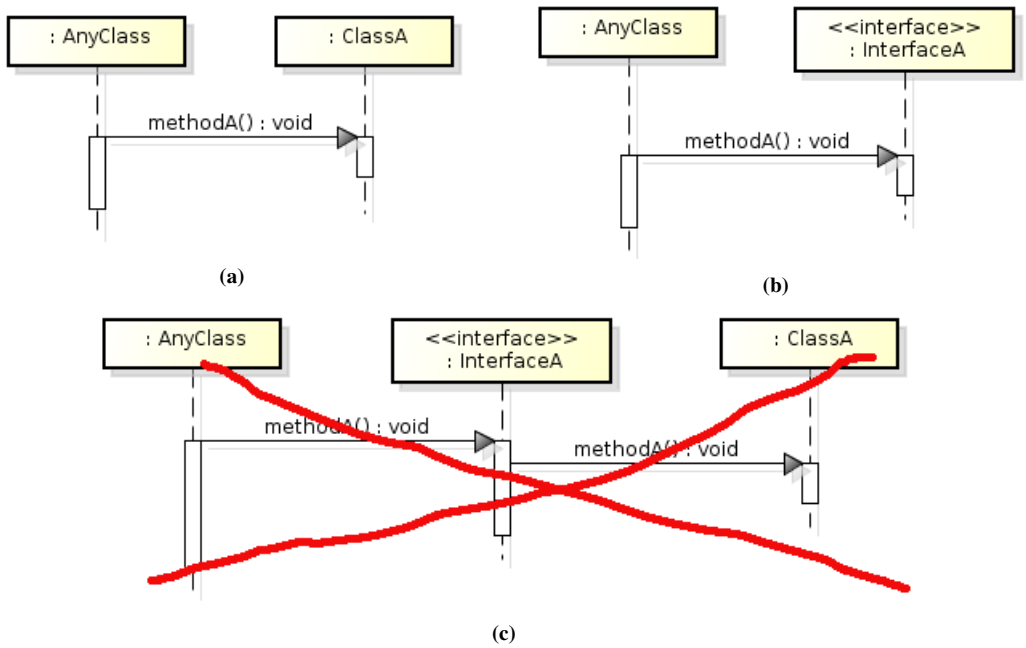


Figure 9.2 An object of a class that implements an interface can be illustrated as the class, figure (a), or as the interface, figure (b), but *not as both*, figure (c).

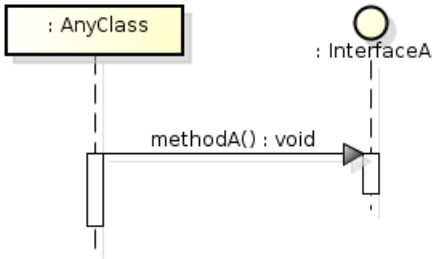


Figure 9.3 Using icon notation to illustrate an interface.

There are two different ways to draw an interface in a communication diagram and a sequence diagram. It can be drawn either as in figure 9.2, using the same symbol as in a class diagram, or it can be drawn using the symbol in figure 9.3. Examples of an interface in a communication diagram can be found in the UML cheat sheet in Appendix B, figure B.11.

9.2 Polymorphism

Polymorphism is Greek for *many forms*, the result of using polymorphism is that one single public interface can have any number of different implementations. How to achieve this will be explained with the help of a concrete example, namely a logging API. For a start, this API consists of one single class, which can write log entries to a file, listing 9.1. The public interface of this class is colored blue in the listing. It consists of the class name, the definition of the constructor, and the definition of the `log` method. Which part of this public interface is *really* necessary to know for an object that wants

Figure 9.4 Polymorphism in nature: light and dark morph (form) jaguars. Image by en:User:Cburnett - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1284082 (top), and By Ron Singer - U.S. Fish and Wildlife Service, Public Domain, https://commons.wikimedia.org/w/index.php?curid=314156 (bottom).

to log something? In fact, it is only the `log` method. That method *must* be called for a log entry to be written to the file, and therefore its declaration must be known to a client of the API. The constructor, on the other hand, is not mandatory knowledge for a log client that does not create an object of `FileLogger`. Also the class name could be hidden from a client, if it was possible to give the client a reference to *some* object, *of an unspecified class*, that has a `log` method. The client would then be able to use that object to call the `log` method, and have the entry printed to the log file, without caring about the class name. To fully understand this fact, consider listing 9.2, which contains a client of the log API. The constructor of `FileLogger` is not used anywhere, and the class name, `FileLogger`, is used only on lines eight and ten. To get rid of the dependency on this class name, it would suffice to remove it from those two places. This can be achieved by introducing an interface specifying the mandatory knowledge about the API, which, as stated above, is only the `log` method. Let's create such an interface, and call it `Logger`, listing 9.3. Also, `FileLogger` shall be changed to implement this interface, but no other change is required to `FileLogger`, since it already has the `log` method specified in the interface. Now, a client of the API does not need to know anything besides the content of this interface, which is proved by listing 9.4.

```
1  package se.leiflindback.oodbook.polyminherit.logapi;
2
3  import java.io.FileWriter;
4  import java.io.IOException;
5  import java.io.PrintWriter;
6
7  /**
8   * Prints log messages to a file. The log file will be in the
9   * current directory and will be called log.txt.
10  */
11 public class FileLogger {
12     private PrintWriter logStream;
13
14     /**
15      * Creates a new instance and also creates a new log file.
16      * An existing log file will be deleted.
17      */
18     public FileLogger() {
19         try {
20             logStream = new PrintWriter(
21                         new FileWriter("log.txt"), true);
```

```
22          } catch (IOException ioe) {
23              System.out.println("CAN NOT LOG.");
24              ioe.printStackTrace();
25          }
26      }
27
28      /**
29       * Prints the specified string to the log file.
30       *
31       * @param message The string that will be printed to the
32       *                log file.
33       */
34      public void log(String message) {
35          logStream.println(message);
36      }
37  }
```

Listing 9.1 In the first version, the logging API contains only this class. The public interface is colored blue.

```
1   package se.leiflindback.oodbook.polyminherit.logapi;
2
3   /**
4    * A client for the logger. Prints log messages to the
5    * specified logger.
6    */
7   public class AnyClassThatNeedsToLogSomething {
8       private FileLogger logger;
9
10      public void setLogger(FileLogger logger) {
11          this.logger = logger;
12      }
13
14      /**
15       * Prints to the log. The logged string includes the
16       * specified message number.
17       *
18       * @param msgNo This number is included in the logged
19       *              string.
20       */
21      public void anyMethod(int msgNo) {
22          logger.log("Important message number " + msgNo);
23      }
24  }
```

Listing 9.2 A client of the log API in listing 9.1.

```
1  package se.leiflindback.oodbook.polyminherit.logapi;
2
3  /**
4   * Specifies an object that can print to a log. This interface
5   * does not handle log locations, it is up to the implementing
6   * class to decide where the log is.
7   */
8  public interface Logger {
9
10     /**
11      * The specified message is printed to the log.
12      *
13      * @param message The message that will be logged.
14      */
15     void log(String message);
16 }
```

Listing 9.3 An interface specifying the functionality "write to the log".

```
1  package se.leiflindback.oodbook.polyminherit.logapi;
2
3  /**
4   * A client for the logger. Prints log messages to the
5   * specified logger.
6   */
7  public class AnyClassThatNeedsToLogSomething {
8      private Logger logger;
9
10     public void setLogger(Logger logger) {
11         this.logger = logger;
12     }
13
14     /**
15      * Prints to the log. The logged string includes the
16      * specified message number.
17      *
18      * @param msgNo This number is included in the logged
19      *              string.
20      */
21     public void anyMethod(int msgNo) {
22         logger.log("Important message number " + msgNo);
23     }
24 }
```

Listing 9.4 A client of the log API, when `FileLogger` implements the `Logger` interface in listing 9.3.

This code is quite amazing already now. It contains an API that has encapsulated the name of the class doing the work of the API! This is extremely low coupling, the client is using a class, `FileLogger`, without any dependency on the name of that class. Still, we are far from done, it is about to become much more amazing. Now is the time to actually make use of polymorphism, that is, to provide more than one implementation of a single public interface. The public interface for the logging API consists of the `log` method in the `logger` interface. This public interface currently has one implementation, provided by `FileLogger`. The second implementation will be a class, `ConsoleLogger` in listing 9.5, which prints the log messages to the screen, instead of to a file. Also this implementation fulfills the contract of the public interface defined by the `log` method in listing 9.3, which, according to the JavaDoc of that method is that *the specified message is printed to the log*.

```
1  package se.leiflindback.oodbook.polyminherit.logapi;
2
3  /**
4   * Prints log messages to <code>System.out</code>.
5   */
6  public class ConsoleLogger implements Logger {
7
8      /**
9       * Prints the specified string to <code>System.out</code>.
10      *
11      * @param message The string that will be printed to
12      *                <code>System.out</code>.
13      */
14     @Override
15     public void log(String message) {
16         System.out.println(message);
17     }
18 }
```

Listing 9.5 The second implementation of the `Logger` interface is this class, `ConsoleLogger`, which prints the log messages to the screen.

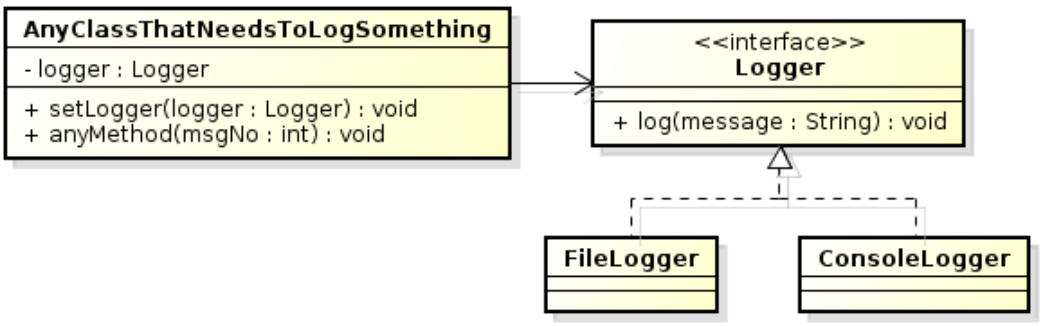


Figure 9.5 The log API and a class using it.

Figure 9.5 contains a class diagram with all classes created so far. The client of the log API, `AnyClassThatNeedsToLogSomething` depends only on the interface `Logger`. *It has no coupling whatsoever to the implementations of the interface!* This fact is the foundation of all fancy usages of polymorphism. It means that `AnyClassThatNeedsToLogSomething` does not know, and has no interest in, which class it is actually calling. All that matters is that it is a class providing an implementation of the required public interface. This is guaranteed by the compiler, since any object passed to `setLogger` must be of a class implementing `Logger`, and must therefore provide an implementation of the public interface defined in `Logger`. It is now possible to change implementation at runtime, as is done when using the `main` method in listing 9.6. It is fundamental understand this example, similar designs are extremely common in object-oriented programming.

```
1  package se.leiflindback.oodbook.polyminherit.logapi;
2
3  /**
4   * Contains the main method of the log API client.
5   */
6  public class Main {
7
8      /**
9       * @param args The program does not take any command line
10      *             parameters.
11      */
12     public static void main(String[] args) {
13         AnyClassThatNeedsToLogSomething client
14                 = new AnyClassThatNeedsToLogSomething();
15
16         client.setLogger(new FileLogger());
17         client.anyMethod(1);
18         client.anyMethod(2);
19         client.anyMethod(3);
20
21         client.setLogger(new ConsoleLogger());
22         client.anyMethod(4);
23         client.anyMethod(5);
24         client.anyMethod(6);
25     }
26 }
```

Listing 9.6 A main method of a program using the log API. The first three logged messages are printed to the file, the last three to the screen.

In conclusion, by using polymorphism the program gets *higher cohesion*, since the client of the polymorphic design, `AnyClassThatNeedsToLogSomething` in the example above, does not contain any code related to choosing implementation of the public interface. It just uses the implementation passed to it (in the method `setLogger`). There will also be *lower*

coupling, since there is no coupling from the client to any implementation. Finally, the program gets *better encapsulation*, since the names of the implementing classes are completely encapsulated inside the logging API.

The result of improving both coupling, cohesion and encapsulation, is that the behavior of the program can be changed at runtime, by switching implementation of a public interface. This was done in the above example by changing from `FileLogger` to `ConsoleLogger`. Without polymorphism, the destination of the logs would have been hard-coded in the program. The only way to change logging at runtime would have been with `if` statements, like the code in listing 9.7. Such `if` statements would have had to be repeated in every class that wanted to log something. Also, they would have to be updated whenever a new implementation was added.

```
1 if (logToFile) {
2     Write to the file
3 } else if(logToScreen) {
4     Write to the screen
5 } similar statements checking other log destinations
```

Listing 9.7 Pseudocode showing `if` statements replacing polymorphism.

Before leaving this introduction to polymorphism, as an extra exercise let's look at an example where implementations are not only changed at runtime, but also *added* at runtime. Many programming languages, including Java, have the possibility to load and call classes that did not exist when the program was started, which is illustrated in listing 9.8. The actual loading and instantiation of a new class is done on lines 32 and 33. This program will call all `Logger` implementations whose class names are specified as command line parameters. To make the class use the `FileLogger` and `ConsoleLogger`, the command line parameters shall be `se.leiflindback.oodbook.polymorphism.logapi.FileLogger se.leiflindback.oodbook.polymorphism.logapi.ConsoleLogger`. Note the space between the class names.

```
1  package se.leiflindback.oodbook.polyminherit.logapi;
2
3  /**
4   * Contains a main method of the log API client, which loads
5   * new <code>Logger</code> implementations at runtime.
6   */
7  public class LoadImplAtRuntime {
8      private int msgNo = 1;
9      private AnyClassThatNeedsToLogSomething client =
10             new AnyClassThatNeedsToLogSomething();
11
12     /**
13      * @param args Each command line parameter shall be the
14      *             fully qualified class name of a class
15      *             implementing <code>Logger</code>. This
```

```
16       *              class will be loaded and used.
17       */
18      public static void main(String[] args) throws
19                                InstantiationException,
20                                IllegalAccessException,
21                                ClassNotFoundException,
22                                NoSuchMethodException,
23                                InvocationTargetException {
24          LoadImplAtRuntime main = new LoadImplAtRuntime();
25          for (String loggerClassName : args) {
26              main.loadAndUseLogger(loggerClassName);
27          }
28      }
29
30      private void loadAndUseLogger(String logger) throws
31              InstantiationException,
32              IllegalAccessException,
33              ClassNotFoundException,
34              NoSuchMethodException,
35              InvocationTargetException {
36          Class logClass = Class.forName(loggerClassName);
37          Logger logger = (Logger) logClass.
38                          getDeclaredConstructor().newInstance();
39          client.setLogger(logger);
40          client.anyMethod(msgNo++);
41      }
42  }
```

Listing 9.8 A `main` method that loads classes at runtime. To make it load `FileLogger` (listing 9.1) and `ConsoleLogger` (listing 9.5), the command line parameters shall be `se.leiflindback.oodbook.polymorphism.logapi.FileLogger se.leiflindback.oodbook.polymorphism.logapi.ConsoleLogger`

9.3 Inheritance

Inheritance is a very special kind of relation between classes. If one class, the subclass, inherits another class, the superclass, it means that *all non-private members of the superclass also become members of the subclass*. In fact, all code in the superclass, except code with private visibility, becomes code also in the subclass. This very special kind of relation has severe consequences, as will be explained below. Before proceeding, it might adequate to repeat the basics of inheritance, by reading section 1.8.

Another thing that must be understood is *protected* visibility. This is the fourth type of visibility, besides public, private and package private. A member with protected visibility can be accessed by a subclass in any package, but by a non-subclass only in the same package. This is illustrated in figure 9.6. Note that the symbol for protected visibility is the hash character,

#. Since the protected member is visible to classes in any package, it is part of the public interface, and not of the implementation. It is thus more closely related to public visibility than to private or package private.

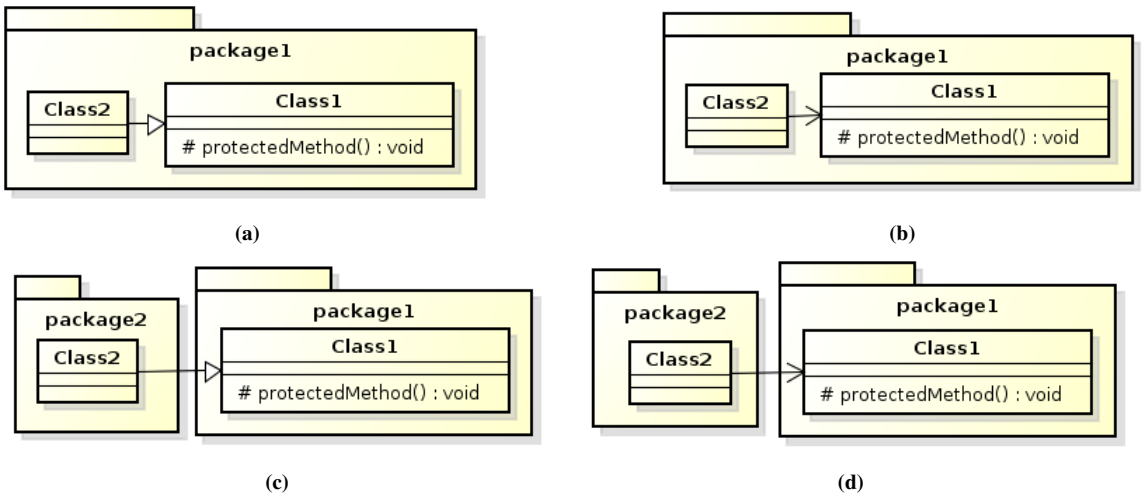


Figure 9.6 The method `protectedMethod`, which has protected visibility, can be called by a method in `Class2` in all situations except diagram (d).

When Not to Use Inheritance, And What to do Instead

It is never appropriate to use inheritance just to reuse code from another class. Code reuse is much better achieved by simply calling the methods that shall be reused, instead of inheriting them. This strategy, to use a plain, ordinary association instead of inheritance, is called *composition*. The reasons to prefer composition are explained below. Before exploring the dangers of inheritance, however, let's make clear that there are situations where inheritance *is* appropriate, as will be explained later, in the section *How to Use Inheritance Appropriately*. The point here is that code reuse alone is not a sufficient reason for using inheritance.

First Reason to Prefer Composition: Inheritance Complicates Code Reuse

Much is written on creating classical inheritance hierarchies, like the one in figure 9.7. However, they are often not as useful as it might seem, but instead make it difficult to reuse code.

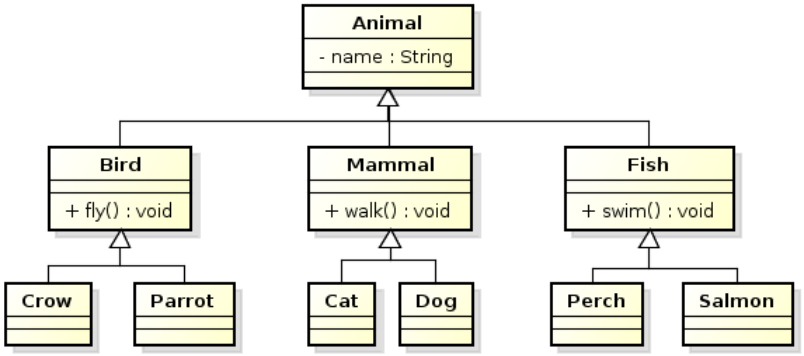


Figure 9.7 An example of a classical inheritance hierarchy, representing different species.

The above class hierarchy describes animals, now let's focus on how they move. The diagram shows that birds fly, fishes swim and mammals walk. Seems OK, but what if want to add a penguin, which is a bird that swims but does not fly? Or a mammal that can both swim and walk? And what about a creature that changes behavior as it grows, for example a bird that can not fly when it is hatched, but later learns to? This reasoning might suggest it would be better to create a class hierarchy based on movement possibilities than on taxonomy, as in figure 9.8.

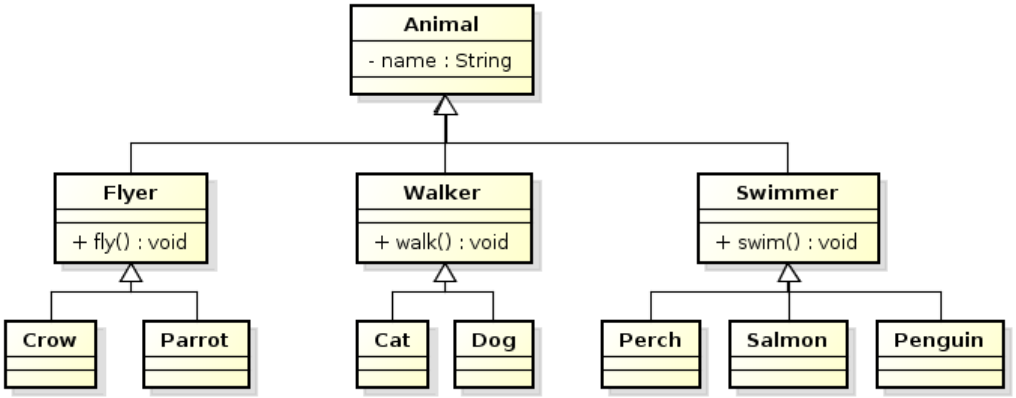


Figure 9.8 The animal inheritance hierarchy, based on how species move, instead of taxonomy.

Unfortunately, this change does not solve the problem. First, we would need multiple inheritance for the mammal that both swims and walks. Second, it would still be impossible to change movement for the bird that learns to fly. Third, there are many properties of animals which do not fit in the movement hierarchy. Consider for example wings, ostriches have wings but can not fly, so the wing property can not be placed in `Flyer`. Above all, why create any hierarchy at all? Why not use composition instead, as in figure 9.9? This way, any animal object can be connected to any movement object, or to several movement objects. Also, an animal object can, at any point in time, change the set of available movement objects. The solution should be improved further, using polymorphism for the movement, and maybe the strategy pattern described below. However, what is important here is only the advantage of

composition over inheritance. To be honest, though, it must be admitted that the diagram in figure 9.9 is messier than those in figures 9.7 and 9.8. This is the disadvantage of composition, that it becomes more difficult to understand which class has a reference to which, especially when references change at runtime, and even more so if polymorphism is used.

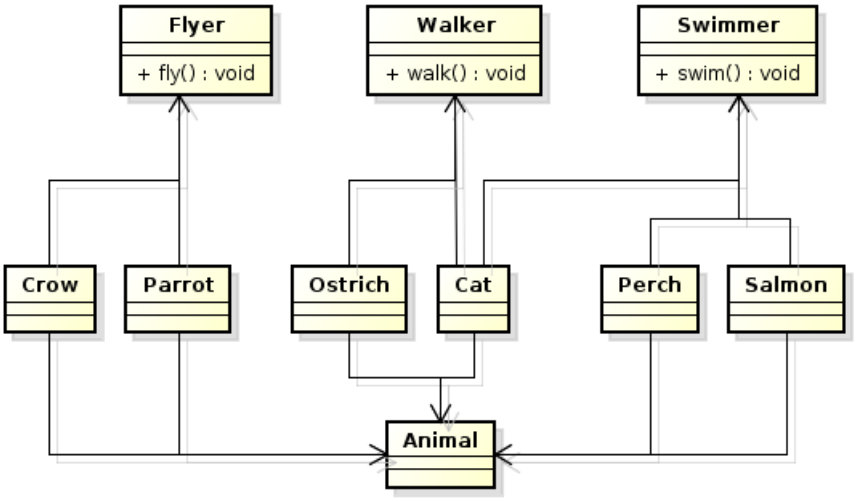


Figure 9.9 Using composition, instead of inheritance, for the animal classes.

Second Reason to Prefer Composition: Inheritance Breaks Encapsulation

Inheritance makes the code difficult to maintain, and may introduce bugs, since it breaks encapsulation. Not only the public interface of the superclass is inherited by the subclass, but also the implementation. Simply speaking, everything in the superclass becomes a part also of the subclass. It should be obvious that the mere fact of completely breaking encapsulation is something bad. Now, let's look at an example illustrating the risk of inheriting a class that was not written to be inherited.

Suppose we have at our disposal an API including a class `List`, part of which is shown in listing 9.9. Unfortunately, this class does not really meet our needs, we want a list that counts how many items have been added during the entire existence of a `List` object (not the number of items currently in the list). We therefore decide to create a new class, `CountingList`, which inherits `List` and adds the missing functionality. This can be done as in listing 9.10.

```
1 public class List {
2     private static final int INITIAL_SIZE = 100;
3     private Object[] content = new Object[INITIAL_SIZE];
4     private int firstFreeIndex = 0;
5
6     /**
7      * Adds an element last in the list.
8      * @param element The element to add.
9      */
```

```
10      public void add(Object element) {
11          content[firstFreeIndex++] = element;
12      }
13
14      /**
15       * Adds all elements in the specified list to this list.
16       * @param elemsToAdd The elements to add.
17       */
18      public void addAll(List elemsToAdd) {
19          // Add all elements in elemsToAdd.
20      }
21  }
```

Listing 9.9 Part of a list implementation.

```
1   public class CountingList extends List {
2       private int noOfAddedElems;
3
4       public void add(Object elemToAdd) {
5           noOfAddedElems++;
6           super.add(elemToAdd);
7       }
8
9       public void addAll(List elemsToAdd) {
10          noOfAddedElems = noOfAddedElems + elemsToAdd.size();
11          super.addAll(elemsToAdd);
12      }
13  }
```

Listing 9.10 An attempt to extend the list from listing 9.9, and count how many items have ever been added to the list.

This seems to be a good solution, but it does not work. When `addAll` is called, the amount of added elements is doubled. Adding five elements increases the `noOfAddedElems` counter by ten. The reason is that `addAll` in `List` was implemented as in listing 9.11. This method iterates over the list and calls `add` for each element, in order not to duplicate the code that adds a single element. However, that call is now to `add` in `CountingList`, and thereby the counter is incremented twice, both in `addAll` and `add`.

```
1   public void addAll(List elemsToAdd) {
2       for (int i=0; i<elemsToAdd.size(); i++) {
3           add(elemsToAdd.get(i));
4       }
5   }
```

Listing 9.11 The implemenation of `addAll`. The call to `add` on line three will be to `add` in `CountingList`.

There are of course different ways to get rid of this bug, for example not to increment `noOfAddedElems` in `addAll`. However, such solutions only make `CountingList` ever more dependent on particularities of `List`'s implementation. This implementation is of course not documented, since it is supposed to be encapsulated. Thus, it might very well be that the `List` implementation changes between releases, without any notice, which might introduce new bugs in `CountingList`. The conclusion is that inheritance is not appropriate here. It is much better to reuse the existing list class with composition, as in listing 9.12. Just as in the animal example above, composition makes the code a bit longer. In this case since `CountingList` must include all `List` methods that shall be available. It is, in fact, not uncommon that composition is less elegant than inheritance seems to be, at first sight. However, composition has the big advantage that it actually works, and it is much less likely to create problems in the future, when the code is changed.

```
1  /**
2   * A list that counts how many elements have ever been added.
3   */
4  public class CountingListUsingComposition {
5      private int noOfAddedElems;
6      private List list = new List();
7
8      public void add(Object elemToAdd) {
9          noOfAddedElems++;
10         list.add(elemToAdd);
11     }
12
13     public void addAll(
14                 CountingListUsingComposition elemsToAdd) {
15         noOfAddedElems = noOfAddedElems + elemsToAdd.size();
16         list.addAll(elemsToAdd.list);
17     }
18
19     /**
20      * Tells how many elements have ever been added to
21      * the list.
22      *
23      * @return the number of elements that have ever been
24      *         added to the list.
25      */
26     public int noOfAddedElems() {
27         return noOfAddedElems;
28     }
29
30     public int size() {
31         return list.size();
32     }
33
```

```
34      public Object get(int index) {
35          return list.get(index);
36      }
37  }
```

Listing 9.12 Code reuse with composition, instead of inheritance.

Third Reason to Prefer Composition: There is no Way to Control the Public Interface of the Subclass

A subclass blindly accepts the entire public interface of its superclass. If those classes have different authors, the subclass' author has effectively given up control of the public interface. Not a very nice situation. This is the case in listing 9.10, where `CountingList` has the entire public interface of `List`. This might seem appropriate, but can we really be sure we want *everything* from `List` to appear also in `CountingList`? As an example, `java.util.List` has 35 methods (in JDK 8). If the list class inherited here is of a similar size, it is quite an effort just to go through all those methods and decide if they suit the subclass. Then, it is necessary to know when new versions of the superclass are released, and check those for changes of the public interface. If a new superclass version is not checked, the public interface of the subclass might change without anyone changing its code, and without anyone knowing it changed.

How to Use Inheritance Appropriately

Given the drawbacks discussed above, it might seem that inheritance should never be used at all. This is of course not the case. Used correctly, it can give really beautiful code that is easy to understand and maintain. Below are typical situations where inheritance should be used.

Use Inheritance to Modify Behavior

It is sometimes the case that the task of one class is very similar to the task of another class, except in some detail. Consider for example a class that creates a bar chart illustrating a data set, which is read from a database. Such a class could be written as in listing 9.13

```
1   public class BarChart {
2       private String dataId;
3
4       public BarChart(String dataId) {
5           this.dataId = dataId;
6       }
7
8       public Image getChart() {
9           try {
10              // read the specified data from database.
11              // create image with bar chart.
```

```
12              return barChartImage;
13          } catch(SQLException sqle) {
14              // exception handling
15          }
16          return null;
17      }
18  }
```

Listing 9.13 Pseudocode for a class that reads data from a database, and presents it in a bar chart.

Now suppose we also want to present the data as a line chart. This would require another class, `LineChart`, which would be identical to `BarChart` except for the image creation on line eleven. Obviously there will be duplicated code, but how to remove it? The problem is that the duplicated code forms the structure of the class, for example the `try-catch` block on lines 9, 13 and 15 would be duplicated. A solution is to create a superclass that contains all common code, and place the specific code in a method that can be overridden in subclasses, as in listing 9.14. This way of placing implementation specific code in a protected method is in fact a design pattern called *template method*, which is explained in more detail below, in section 9.4.

```
1  public abstract class Chart {
2      private String dataId;
3
4      public Chart(String dataId) {
5          this.dataId = dataId;
6      }
7
8      public Image getChart() {
9          try {
10             // read the specified data from database.
11             Image chartImage = createChartImage();
12             return chartImage;
13         } catch(SQLException sqle) {
14             // exception handling
15         }
16         return null;
17     }
18
19     protected abstract Image createChartImage();
20 }
21
22 public class BarChart extends Chart {
23     public Chart(String dataId) {
24         super(dataId);
25     }
26
```

```
27      protected Image createChartImage() {
28          // create and return bar chart.
29      }
30  }
31
32  public class LineChart extends Chart {
33      public Chart(String dataId) {
34          super(dataId);
35      }
36
37      protected Image createChartImage() {
38          // create and return line chart.
39      }
40  }
```

Listing 9.14 Pseudocode for classes presenting data as bar chart and line chart, without duplicated code.

Use Inheritance to Provide Default Implementation

Inheritance can be used to specify a default implementation of a method in an interface. Consider for example the interface `java.awt.event.MouseListener`, which shall be implemented by a class that is to be notified when the user does something with the mouse. This interface has five methods, for pressing a mouse button, releasing a button, clicking a button, moving the cursor into a window on the screen and moving it out of the window. A class that is interested in mouse clicks would implement `MouseListener` and provide an implementation of the method `mouseClicked`. However, since the interface is implemented, that class would also have to provide empty implementations of the other four methods. These four empty methods will make the class more difficult to understand, and also lower its cohesion. To remedy this problem, there is a class called `MouseAdapter`, which provides empty implementations of all the five methods. Our class that reacts to mouse clicks can then extend `MouseAdapter`, instead of implementing `MouseListener`, thus not having to include the four empty methods. Note that it is not a requirement for a default method implementation to be empty, as described above. Such a method could instead provide a frequently used implementation. Since Java 8, an interface itself can have a default method implementation. This might suggest that a class like `MouseAdapter` is not needed, the implementations can instead be placed in the interface itself. Even though this is true, it is generally not a good idea. The purpose of an interface is to provide a *contract* in the form of a method declaration, nothing more. There is a clear separation of public interface and implementation if the interface contains *only declarations*, but not if it also contains implementations. In fact, the possibility to provide a default implementation in an interface was never intended to be used like this. As is stated in the Oracle documentation[Java Tutorial], the purpose of default implementation is to make it possible to add methods to an existing interface, without breaking the code of all classes that implement a previous version of that interface.

When Class Hierarchies Are Appropriate

In spite of the critique of classical inheritance hierarchies above, they can still be useful. The point is they should not be used blindly, but only under the right circumstances. First of all the following conditions must be met.

1. All members of the superclass must be meaningful also in the subclass.
2. The superclass is a generalization of the subclass, it is more abstract than the subclass, and can therefore be used in more situations than the subclass.
3. The subclass is a specialization of the superclass. It can not be used in all situations where the superclass can be used, but where it can be used, it is better suited than the superclass.
4. The subclass and the superclass must have an *is-a* relation, meaning the expression *a <subclass name> is a <superclass name>* must be true.

Unfortunately, the above conditions are only necessary, but not sufficient. In fact, the hierarchy of animals above, which prohibited code reuse, does meet all four requirements. To formulate a sufficient condition is notably more difficult. A rough rule of thumb could be that inheritance hierarchies are not very useful for real world entities, such as the animals. The reason is that such entities are complex, and do not always follow a strict generalization-specialization tree structure. Instead, inheritance is more useful for purely fabricated entities, that do not exist in the real world.

An example of a well-functioning hierarchy, not modeling real world entities, is the collection api in `java.util`. A small subset of the classes in this api is illustrated in figure 9.10. For example, there is the interface `List`, which defines the contract of a list. This interface is implemented by `AbstractList`, which provides code common for both linked lists and array lists. Then, the classes `ArrayList` and `LinkedList` contain code that is specific for a particular kind of list. When creating these classes, the author is free to decide which classes shall exist, how they relate to each other, and which functionality to put where. With the animals, there is no such freedom. A penguin is a bird and a salmon is a fish, there is no arguing about that.

9.4 Gang of Four Design Patterns

A pattern is a common solution to a recurring problem. Typically, developers realize that a particular problem in software development is solved many times, in different programs, but the solution is always more or less the same. If the solution works well, it is worth creating a formalized description covering the problem, variants of the solution, advantages and disadvantages of the solution, etc. This formalized description is a pattern. If, as is the case here, the solution concerns design, it is a design pattern. A collection of patterns is like a cookbook for software development, describing common solutions to common problems.

Gang of Four, or *GoF*, refers to the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson and Vlissides [GOF]. This book contains a collection of patterns, often called GoF patterns. These patterns are very common, and all

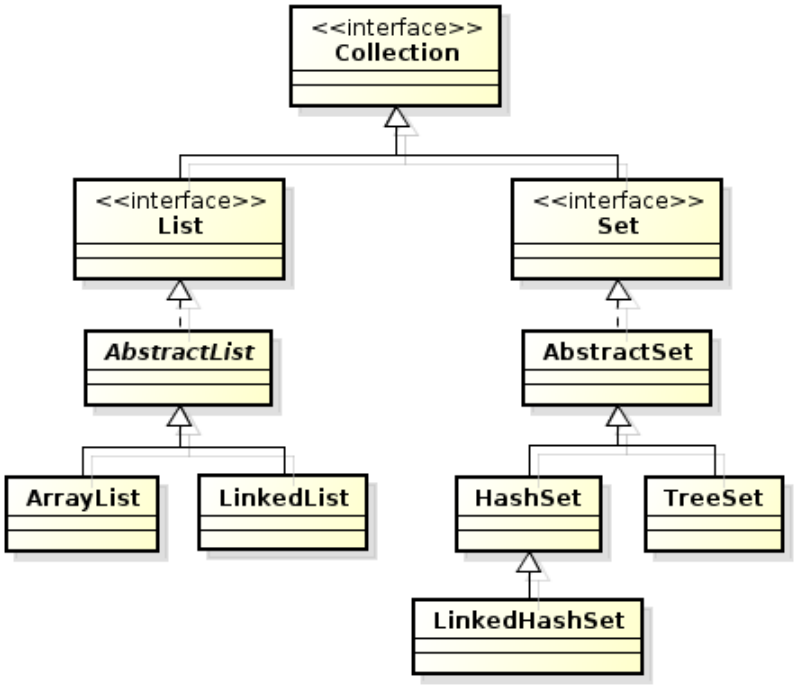


Figure 9.10 Part of the collection api in `java.util`, which is an example of a class hierarchy modeling fabricated entities instead of real world entities.

developers should be comfortable using at least the more common ones. Knowledge of these patterns also forms a vocabulary for software developers, that can be used to discuss solutions to different problems. Thus, even though they are more advanced and abstract than what we have seen so far, and may at first seem a bit scary, it is fundamental to master at least a few of the 23 patterns covered in the book. This section provides a short introduction, it covers five patterns, plus a simplified version of a sixth.

The Observer Pattern

The *observer* pattern is used where objects of one or more classes must be informed whenever a particular object changes state. This is one of the most frequently used GoF patterns, all kinds of listeners and event handlers are variants of the observer pattern.

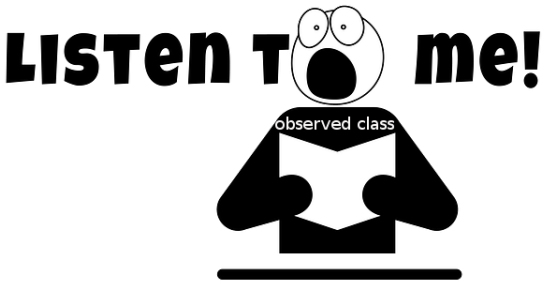


Figure 9.11 Using the observer pattern, the observing objects are immediately notified when the observed object changes state. Image by unknown creator [Public domain], via https://pixabay.com

Overview

The idea is to hide the observer object from the observed class with an interface, figure 9.12. Now, the observed class can call the observer when it changes state, but it *does not know the*

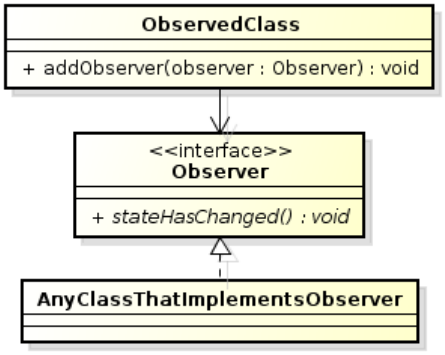


Figure 9.12 General class diagram for any observer pattern implementation. The observed class has no reference to the observing class, but only to the `Observer` interface.

class of the observer, it only knows the name of the interface. In fact, the observed class in listing 9.15 does not contain the name of the class that implements the `Observer` interface. Using an interface this way is a good practice when the caller is interested only in a particular ability of the callee. In this case, the observed class needs a class with the ability to receive notifications about updates, which is provided by the `notify` method. The observed class does not care what else the observing object can do, or what its purpose is, as long as it can receive notifications.

Case Study

Here, an observer will be used to solve a problem we have been facing since the beginning of the design, namely how to pass data from model to view, when that data can not be a return value to a method call from the view. As a case study, the car rental program is augmented with a display on the wall in the office, telling how many cars of each size the company has rented out. Whenever a rental is payed, no matter where or by who, the display shall be updated with the rented car.

```
1  package se.leiflindback.oodbook.polyminherit.observer;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  /**
7   * The observed class in a general implementation of the
8   * observer pattern.
9   */
10 public class ObservedClass {
11     private List<Observer> observers = new ArrayList<>();
12
13     /**
14      * Registers observers. Any <code>Observer</code> that is
15      * passed to this method will be notified when this object
16      * changes state.
17      *
18      * @param observer The observer that shall be registered.
19      */
20     public void addObserver(Observer observer) {
21         observers.add(observer);
22     }
23
```

```
24      // Called by any method in this class that has changed
25      // the class' state.
26      private void notifyObservers() {
27          for (Observer observer : observers) {
28              observer.stateHasChanged();
29          }
30      }
31  }
```

Listing 9.15 The observed class, note that there is no reference to the observing class, only to the `Observer` interface.

There are many reasons why this is difficult to implement. It is not easy to handle the update directly in the view, letting the user interface where the rental is payed update the display showing rented out cars. First, there might be many different ways to rent a car, in different locations, and they can not all know about the display. Second, even if there is only one single place where a customer can rent a car, it is still bad design to let one object in the view be responsible for updating other view objects. That knowledge, what to do in response to a particular user action, is not part of the user interface, and should therefore not be handled by the view. A view object shall only call the controller and update itself to reflect the result of the call. Finally, in the current rent car implementation, the view does not keep track of the rental being paid for, that is done by the controller. There is no reason to give the view the responsibility to know the size of the rented car, that would reduce cohesion in the view, since it is the responsibility of `Rental`, in the model. Also, there would certainly be some amount of duplicated code if both `Rental` and view had to know which car was being rented.

Having established that the display can not be updated by view object(s) responsible for the user interface where the car is rented, we face a brick wall. There is no other object in the view, and it would ruin MVC if the controller or any lower layer called the display to update the number of rented cars. The solution is to use an observer, which will allow the model to call the view without any dependency on the view.

Solution

First, we create the observer interface. It is convenient to include *observer* in the name, to clearly state it is part of the observer pattern. It could be called for example `CarIsHiredObserver`, but let's be a bit less specific and call it `RentalObserver`. That way it can, in the future, handle information also about other `Rental` state changes. Giving very specific names always increases the risk of change. The notification method can be called `newRental`, since it tells that a new rental is completed. The interface is listed in listing 9.16.

```
1  package se.leiflindback.oodbook.rentcarWithExAndDesPat.model;
2
3  /**
4   * A listener interface for receiving notifications about
5   * rented cars. The class that is interested in such
6   * notifications implements this interface, and the object
```

```
 * created with that class is registered with {@link
 * se.leiflindback.oodbook.rentcarWithExAndDesPat.controller.
 * Controller#addRentalObserver(RentalObserver)}. When a
 * car is rented, that object's {@link #newRental newRental}
 * method is invoked.
 */
public interface RentalObserver {
    /**
     * Invoked when a rental has been paid.
     *
     * @param rentedCar The car that was rented.
     */
    void newRental(CarDTO rentedCar);
}
```

Listing 9.16 The observer interface, `RentalObserver`.

The next decision is which class to observe. The required notification is to tell which type of car was rented when a payment is accepted. Therefore, the observed class must have knowledge about both payment and rented car. These requirements are met by `Rental`, which is also chosen. To make it observable, it must have a method that registers observers. This method is called `addObserver`, just as in the general example in listing 9.15. Also, it is good practice to add a private method, `notifyObservers`, which is called by any other method when notifications shall be sent. When shall observers be notified? In this case, the only state change of interest is that a payment is accepted, therefore, `notifyObservers` is called last in the method `pay`. Listing 9.17 contains the updated parts of `Rental`.

```
package se.leiflindback.oodbook.rentcarWithExAndDesPat.model;

public class Rental {
    private List<RentalObserver> rentalObservers =
                                 new ArrayList<>();

    public void pay(CashPayment payment) {
        payment.calculateTotalCost(this);
        this.payment = payment;
        notifyObservers();
    }

    private void notifyObservers() {
        for (RentalObserver obs : rentalObservers) {
            obs.newRental(rentedCar);
        }
    }

    public void addRentalObserver(RentalObserver obs) {
```

```
20          rentalObservers.add(obs);
21      }
22  }
```

Listing 9.17 The observed class, `Rental`.

Then it is time to create the observer implementation. It shall be a class in the view, simulating the display telling the number of rented cars. Currently, there is only one class in the view, which contains the hardcoded sample execution. To improve cohesion, a new class, `RentedCarsDisplay`, is created. This class implements `RentalObserver`, and simply prints the required information to `System.out`, see listing 9.18. As can be seen on line 28, it was appropriate to include a `CarDTO` representing the rented car as parameter in the notification method, `newRental`, since `CarDTO` contains the required information about car size.

Previously, the size of the rented car was represented as a `String`. That was never really appropriate, due to the risk of misspelling. Now that the size is being used on many places in the code, this problem is growing. Therefore, size is changed to be defined using an enumeration, which is placed inside `CarDTO`, to make it available everywhere a `CarDTO` is used, see listing 9.19.

```
1  package se.leiflindback.oodbook.rentcarWithExAndDesPat.view;
2
3  /**
4   * Shows a running total of rented cars of each type.
5   */
6  class RentedCarsDisplay implements RentalObserver {
7      private Map<CarDTO.CarType, Integer> noOfRentedCars =
8                                              new HashMap<>();
9
10     /**
11      * Creates a new instance, with the all counters of rented
12      * cars set to zero.
13      */
14     public RentedCarsDisplay() {
15         for (CarDTO.CarType type : CarDTO.CarType.values()) {
16             noOfRentedCars.put(type, 0);
17         }
18     }
19
20     @Override
21     public void newRental(CarDTO rentedCar) {
22         addNewRental(rentedCar);
23         printCurrentState();
24     }
25
26     private void addNewRental(CarDTO rentedCar) {
27         int noOfRentedCarsOfThisType =
```

```
28              noOfRentedCars.get(rentedCar.getSize()) + 1;
29          noOfRentedCars.put(rentedCar.getSize(),
30                             noOfRentedCarsOfThisType);
31      }
32
33      private void printCurrentState() {
34          System.out.println("### We have now rented out ###");
35          for (CarDTO.CarType type : CarDTO.CarType.values()) {
36              System.out.print(noOfRentedCars.get(type));
37              System.out.print(" ");
38              System.out.print(type.toString().toLowerCase());
39              System.out.println(" cars.");
40          }
41          System.out.println("##############################");
42      }
43  }
```

Listing 9.18 The observer implementation, `RentedCarsDisplay`.

```
1  public final class CarDTO {
2      // Unchanged code
3      public enum CarType {SMALL, MEDIUM, LARGE};
4      // Unchanged code
5  }
```

Listing 9.19 The definition of the enum in `CarDTO`.

The last part of the observer implementation is to create an instance of `RentedCarsDisplay`, and register it with `Rental`. There are really only two candidates for creating this object, the startup class `Main` and the view placeholder `View`. Creating it in `Main` creates more dependencies between the layers `startup` and `view`, which means unnecessary coupling. It is better to let `Main` create only one object in the view, as is the case now, and then let that object unfold the entire view. This unfolding means to create the `RentedCarsDisplay` object.

How shall the view pass the observer to the observed class in the model? There is only one allowed way for a view to communicate with a model, via controller. The solution is thus to pass the observer from view to controller, and then from controller on to `Rental` in the model, see listings 9.20 and 9.21. There are two things to note in the last listing. First there is no need to register the observer again for each new rental. It is instead saved in `Controller` and registered to all future `Rentals`. Second, since there might be more than one observer, the controller keeps all of them stored in a list, and registers them all when a new rental is initiated. That means `Rental` needs a new method, `addObservers`, which allows registering a whole list of observers.

```
1  public class View {
2      // Unchanged code
```

```
3       public View(Controller contr) throws IOException {
4           this.contr = contr;
5           contr.addRentalObserver(new RentedCarsDisplay());
6           this.logger = new LogHandler();
7       }
8       // Unchanged code
9   }
```

Listing 9.20 Creating the observer in `View`.

```
1   public class Controller {
2       private List<RentalObserver> rentalObservers =
3                                    new ArrayList<>();
4
5       public void registerCustomer(CustomerDTO customer) {
6           // Unchanged code.
7           rental.addRentalObservers(rentalObservers);
8       }
9
10      /**
11       * The specified observer will be notified when a rental
12       * has been paid. There will be notifications only for
13       * rentals that are started after this method is called.
14       *
15       * @param obs The observer to notify.
16       */
17      public void addRentalObserver(RentalObserver obs) {
18          rentalObservers.add(obs);
19      }
20  }
```

Listing 9.21 Handling observer registration in `Controller`.

Comments

There is much worth pondering about the observer pattern. Here are some issues concerning the implementation.

- **What shall the observer know about the observed class?** The observer knows nothing at all about the observed class in the case study above. In fact, there is no way for it to get a reference to the object that sent the notification. Instead, all relevant data is passed as a parameter in the call of the notification method, in this case, the data is a `CarDTO` object. This solution becomes more and more problematic as more data is required. Also, whenever the

need for data changes, the parameter list of the notification method must change, which means the public interface of `Observer` is changed.

Another solution is to not pass any data at all, but instead a reference to the object that changed state, the `Rental` object in the above case study. The observer can then call get methods in the observed class to collect needed data. This solution is more dynamic, since different observers can collect different data, as required. Also, the parameter list of the notification method consists of one single parameter, the object that changed state, and will never change. The downside of such an implementation is that the observer now depends on the observed class, thereby increasing coupling. To alleviate this problem, we can introduce yet one interface, call it `Observed`, implemented by the observed class. This interface shall contain only methods of the observed class allowed to call by the observer. Now, the observer will know only about this interface, not about the actual class being observed.

- **Is an event class needed?** It is sometimes appropriate to introduce a new class, representing the actual event. If so, an object of this class is normally the only parameter of the notification method. Any data can be included in this object, and the parameter list of the notification method is thus no longer cluttered with this data. The event object can also contain a reference to the observed object, if that is needed, and can it contain any other information of interest concerning the event.

- **When shall observers be notified?** Observers must be informed about a particular event only when that event is really completed. It would for example not be acceptable to call `notifyObservers` on line eight, instead of line ten, in listing 9.17, since the payment has not yet been handled on line eight.

- **Broadcasted method call!** Using an observer does, in fact, enable broadcasting a method call! Instead of calling a method in *one* observer object, the observed object calls `notifyObservers`, which means the method is called in *many* observer objects. This is really powerful, but, as many powerful solutions, carry a risk. The risk is that the caller can not know how many objects are called, nor what all those objects will do when called. Therefore, what seems like an innocent method call to the observed class, might in reality result in lengthy and resource consuming operations. To avoid this, event handling methods in observers shall not perform any costly operations.

The Strategy Pattern

The *strategy* pattern makes it possible to swap an algorithm without changing existing code. In fact, the logging API created in section 9.2 used strategy to change logger implementation. It was possible to change between `FileLogger` and `ConsoleLogger` while the program was running, and it was also possible to add new classes implementing `Logger` without any changes to the code. *Algorithm* is used here in a very wide sense, it can be any kind of behavior.

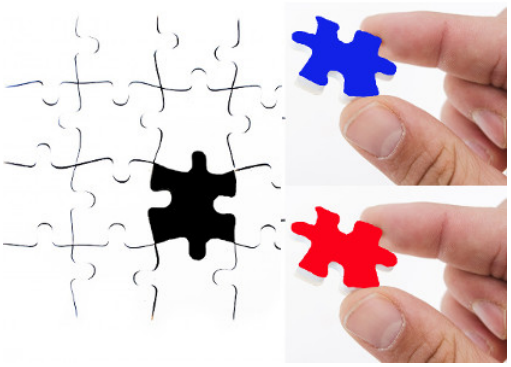


Figure 9.13 Using the strategy pattern, any algorithm can be plugged in to an existing program. Image by unknown creator [Public domain], via http://www.publicdomainpictures.net

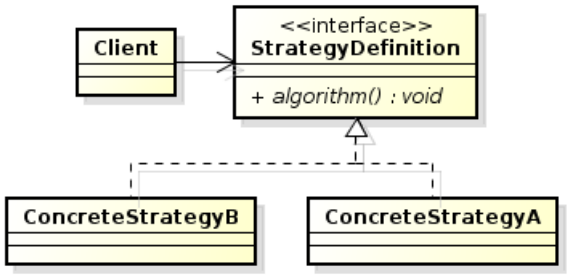


Figure 9.14 The client can use any class that implements `StrategyDefinition`, it is completely independent of the implementing classes.

Overview

The strategy pattern is a perfect example of the benefits of polymorphism. If the client using the algorithm depends only on the interface, and not on the implementations, as in figure 9.14, it is possible to change implementation without any changes to the client. Figure 9.13 also illustrates this, both the red and the blue piece fit. The surrounding pieces are not affected as long as the shape (definition) is correct.

Case Study

The logging API from section 9.2 would be a great case study. Still it is not used, since there are already many existing logging APIs. It would be a bit inappropriate to spend a lot of effort developing yet another. The case study used here is instead the search for an available car matching the wishes of a customer. This is performed in `CarRegistry`, where the current search algorithm, listing 9.22, considers a car to be a match if all properties except registration number are equal to those of the searched car. Properties equal to `null` or zero are ignored, and the matching car must not be booked. This is, however, only one of many possible ways to match existing cars against a customer's wishes.

```
1 /**
2  * Search for a car that is not booked, and that matches the
3  * specified search criteria.
4  *
5  * @param searchedCar This object contains the search
6  *                    criteria. Fields in the object that are
7  *                    set to <code>null</code> or zero are
```

```
 *                    ignored.
 * @return A car matching the searched car's
 *         description if an unbooked car with the
 *         same features as <code>searchedCar</code> was
 *         found, <code>null</code> if no such car was found.
 */
public CarDTO findAvailableCar(CarDTO searchedCar) {
    for (CarData car : cars) {
        if (matches(car, searchedCar)) {
            return new CarDTO(car.regNo,
                              new Amount(car.price), car.size,
                              car.AC, car.fourWD, car.color,
                              false);
        }
    }
    return null;
}

private boolean matches(CarData found, CarDTO searched) {
    if (searched.getPrice() != null &&
        !searched.getPrice().equals(new Amount(found.price))){
        return false;
    }
    if (searched.getSize() != null &&
        !searched.getSize().equals(found.size)) {
        return false;
    }
    if (searched.getColor() != null &&
        !searched.getColor().equals(found.color)) {
        return false;
    }
    if (searched.isAC() != found.AC) {
        return false;
    }
    if (searched.isFourWD() != found.fourWD) {
        return false;
    }
    if (found.booked) {
        return false;
    }
    return true;
}
```

Listing 9.22 The algorithm used to match cars, before using the strategy pattern.

Now, it shall be possible to switch matching algorithm. For the case study, the existing algorithm is kept, and two new algorithms are added. One exact match algorithm, where all

properties of a car in the registry must be equal to those of the searched car, and one algorithm where a specific available car always becomes the match, as long as at least one of its properties matches the search criteria. The latter algorithm is used when the rental company wants to promote a particular car. The strategy pattern is used to enable algorithm switching. Without that pattern, it would be necessary to hardcode all algorithms in `CarRegistry`, which would be a very bad solution. The code would be very messy, and it would be difficult to add new algorithms. Instead, as is dictated by the strategy pattern, a strategy interface will be created, and `CarRegistry` will be assigned an implementation of that interface performing the desired search algorithm.

Solution

Figure 9.15 and listing 9.23 show the strategy interface, `Matcher`, and its implementations handling the search for a car in the car registry. Note that `CarRegistry` has no dependency on the implementations, only on the interface. As can be seen from the definition of the `match` method, it is handed a list containing all available cars. This approach can be questioned, is it not time consuming to read all cars from the database and place them in a list? Maybe yes, the search could have been defined in the database query, in a way that only a matching car was returned. To solve the problem that way, `match` would have needed a database connection for sending the appropriate query to the database, instead of the list of all existing cars. There are, however, reasons to solve the problem as in figure 9.15 instead. Above all, there is no database yet. Also, even though the proposed solution is more time consuming than a database query, it will probably not create problems, since the list can not be extremely long. It is unlikely that the rental company has more than a few hundred available cars to choose from. Another thing worth noting is that an available car is specified by a `CarDTO` object, not by the `CarData` object used internally in `CarRegistry`. This is to maintain the distinction that `CarData` is a a replacement for a database, it is not appropriate for use outside `CarRegistry`.

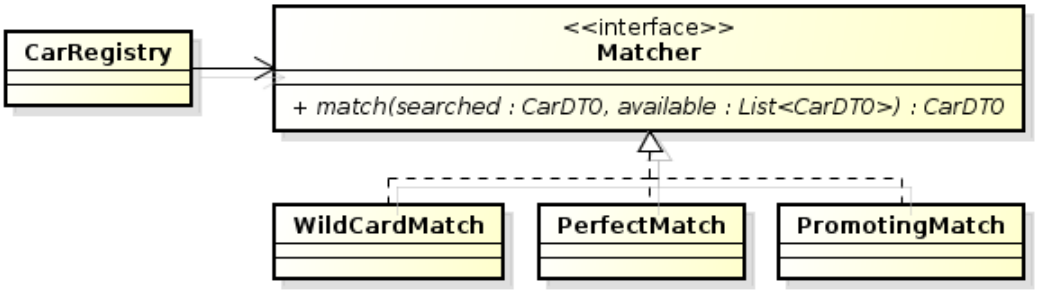


Figure 9.15 The strategy case study. There are three different algorithms for matching a car in the registry against a customer's wishes.

```
1 /**
2  * Defines the ability to match existing cars with a searched
3  * car. This interface shall be implemented by a class that
4  * provides a matching algorithm.
```

```
5   */
6  public interface Matcher {
7      /**
8       * Searches the specified available cars for an instance
9       * matching the specified search criteria.
10      *
11      * @param searched  Search criteria
12      * @param available Available cars
13      * @return A matching car, or <code>null</code> if none
14      *         was found.
15      */
16     CarDTO match(CarDTO searched, List<CarDTO> available);
17 }
```

Listing 9.23 The `Matcher` interface, wich defines the matching algorithm.

Next, it is time to create the implementations. The existing search algorithm looks for a car with all properties, except registration number, equal to the properties of the searched car. However, searched properties equal to `null` or zero are ignored. Therefore, this algorithm is called `WildCardMatch`. The algorithm that never ignores any property, but requires all properties except registration number to match, is called `PerfectMatch`. Finally, the algorithm used to promote a particular car is called `PromotingMatch`. The first two can be found in the accompanying GitHub repository [Code], only the last is listed here, listing 9.24. The registration number of the car to promote is specified in `setCarToPromote`, lines 21-23. The search algorithm is, of course, written in the method `match`, defined by the implemented interface. `PerfectMatch` is used as a fallback if the promoted car is different from the searched car in all aspects (line 59). This is in order to *not* have to tell the customer "sorry, there is no car", even if there actually is one.

```
1  /**
2   * A <code>Matcher</code> that finds the car that shall be
3   * promoted, provided it has at least one property, except
4   * registration number, matching the search criteria. If it
5   * has not, performs a <code>PerfectMatch</code>.
6   */
7  public class PromotingMatch implements Matcher {
8      private String regNoOfCarToPromote;
9
10     PromotingMatch() {
11     }
12
13     /**
14      * Specify which car to promote.
15      *
16      * @param regNo The car with this registration number will
17      *              be found by the matching algorithm, if it
```

```
18       *              exists and has at least one property equal
19       *              to the search criteria.
20       */
21      public void setCarToPromote(String regNo) {
22          this.regNoOfCarToPromote = regNo;
23      }
24
25      @Override
26      public CarDTO match(CarDTO searched,
27                          List<CarDTO> available) {
28          for (CarDTO carToMatch : available) {
29              if (!regNoOfCarToPromote.
30                  equals(carToMatch.getRegNo())) {
31                  continue;
32              }
33              if (carToMatch.getPrice() != null &&
34                  searched.getPrice() != null &&
35                  searched.getPrice().equals(
36                                       carToMatch.getPrice())) {
37                  return carToMatch;
38              }
39              if (carToMatch.getSize() != null &&
40                  searched.getSize() != null &&
41                  searched.getSize().equals(
42                                      carToMatch.getSize())) {
43                  return carToMatch;
44              }
45              if (carToMatch.getColor() != null &&
46                  searched.getColor() != null &&
47                  searched.getColor().equals(
48                                       carToMatch.getColor())) {
49                  return carToMatch;
50              }
51              if (searched.isAC() == carToMatch.isAC()) {
52                  return carToMatch;
53              }
54              if (searched.isFourWD() ==
55                  carToMatch.isFourWD()) {
56                  return carToMatch;
57              }
58          }
59          return new PerfectMatch().match(searched, available);
60      }
61  }
```

Listing 9.24 `PromotingMatch`, wich performs a matching algorithm that tries to match a particular car.

The last part of the strategy pattern is the client, `CarRegistry`. The use of the matching algorithm can be found on line 25 in listing 9.25. Unfortunately, there is a dependency on the implementation of the algorithm, not only on the definition, since the `WildCardMatch` object is created on this line. It is still worth the effort to define the `Matcher` interface, since the only change required to switch algorithm is to change `WildCardMatch` for the desired class name. However, it is not at all as beautiful as the implementation of the logging algorithm in section 9.2, where it was possible to change algorithm while the program was running. This defect will be mitigated with the next pattern, *factory*.

```
1  /**
2   * Search for a car that is not booked, and that matches the
3   * specified search criteria.
4   *
5   * @param searchedCar This object contains the search
6   *                    criteria. Fields in the object that are
7   *                    set to <code>null</code> or zero are
8   *                    ignored.
9   * @return A description matching the searched car's
10  *         description if an unbooked car with the
11  *         same features as <code>searchedCar</code> was
12  *         found, <code>null</code> if no such car was found.
13  */
14 public CarDTO findAvailableCar(CarDTO searchedCar) {
15     List<CarDTO> allCars = new ArrayList<>();
16     for (CarData car : cars) {
17         if (!car.booked) {
18             allCars.add(new CarDTO(car.regNo,
19                                    new Amount(car.price),
20                                    car.size, car.AC,
21                                    car.fourWD, car.color,
22                                    car.booked));
23         }
24     }
25     return new WildCardMatch().match(searchedCar, allCars);
26 }
```

Listing 9.25 The call from `CarRegistry` to the matching algorithm.

Comments

- **Algorithms might need a state** Concrete strategies might differ only in state, that is, the value of some property. If so, there is no need for a new class for each state. An example of this is `PromotingMatch`, which tries to match a particular car. Which car to promote is specified as a registration number. There is no need to create a new class just to promote another car.

- **All algorithms must have the same public interface** It is problematic that all concrete strategies must implement the same interface, even if different implementations of the algorithm need different data. An example in the case study could be if one matching algorithm needs to call the database, while others do not. In this case, a database connection must be passed to the `match` method, even though some classes will never use it. This might seem a small problem, the solution is to simply set the database connection parameter to `null`, when using an algorithm that does not call the database. However, it is not high cohesion, nor a very nice public interface, to have methods with parameters they do not use. The bigger the public interface, and the more the algorithms differ, the bigger this problem gets. Unfortunately, there is no solution except to unify the algorithm's public interfaces as much as possible.

- **Who shall instantiate implementations?** As stated above, it is unfortunate that, in the case study, there is a dependency on the implementation of the algorithm, not only on the definition. This dependency can be found on line 25 in listing 9.25, where the concrete strategy is instantiated. This instantiation problem is relatively common when using the strategy pattern, and a common solution is to use a factory, which is the next pattern.

The Factory Pattern

In some cases, instantiating objects worsens the design, wherever it is done. The reason could be that an unwanted coupling is created, as is the case above, where `CarRegistry` depends on the strategy implementations. It could also be that creating an object is complicated, and thereby gives bad cohesion to the class where it is placed. Whatever the reason why instantiating objects worsens the design, the solution is to create a completely new class, whose sole purpose is to create those objects. This new class is called a *factory*. There are, in fact, many GoF patterns concerning instantiation, but none of them is called Factory. The pattern described here is a simplified version of some of those GoF patterns.

Figure 9.16 The factory delivers the desired product. They all look the same on the outside (public interface), but the factory knows which has the appropriate content (implementation).

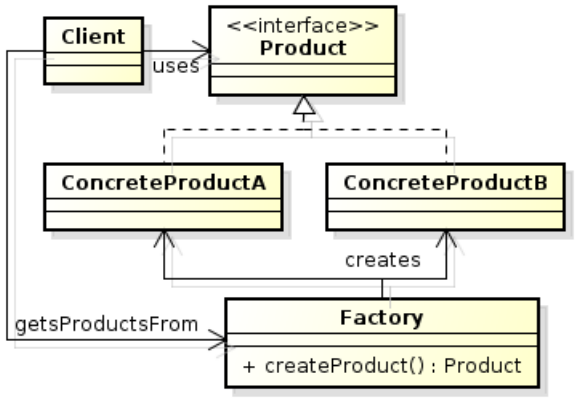


Figure 9.17 `Client` has no dependency on the concrete products, when they are created by the `Factory`.

Overview

As is illustrated in figures 9.16 and 9.17, the client does not know the class of the object it gets from the factory. The client simply asks for a product, and happily uses whatever concrete product it is handed. All that matters is that this concrete product can perform the work specified by the `Product` interface. The question that immediately arises is "How can the factory know which concrete product is appropriate for the client?" Unfortunately, there is no simple answer, but some possibilities are discussed below.

Case Study

Now it is finally time to remove `CarRegistry`'s dependency on the concrete matching algorithms, which has been troubling us since it was introduced on line 25 in listing 9.25. It is not a good design that `CarRegistry` must be changed and recompiled, when the algorithm matching a customers wishes against available cars is swapped. The whole purpose of using the strategy pattern was to make `CarRegistry` independent of those algorithms.

Solution

The solution to the undesired dependency problem is to introduce a factory. Let's call it `MatcherFactory`, since it creates matching algorithms. It needs only one method, which shall return an object of a class implementing the strategy interface `Matcher`, defined in listing 9.23. The open question is how the factory shall know which of the concrete matchers to return. The solution used here, is to let the factory read the class name of the `Matcher` implementation from a system property, see line 31-32 in listing 9.26. A system property is specified on the command line with the switch `-D`. If, for example, the car rental program is started with a command line containing `-Dse.leiflindback.rentcar.matcher.classname=se.leiflindback.oodbook.rentcarWithExAndDesPat.integration.matching.WildCardMatch`, the value returned by `System.getProperty` on line 32 would be `se.leiflindback.oodbook.rentcarWithExAndDesPat.integration.matching.WildCardMatch`, which is the class name of the desired implementation. System properties constitute a nice way to pass initialization data to the object needing it. A system property can be specified on the command line, and read using `System.getProperty` anywhere in the code. It is not necessary to pass a system property from `main` to the class using it, as is the case with a command line parameter.

```
/**
 * A Factory that creates instances of the algorithm used
 * for matching a customer's wishes against available cars.
 * All such instances must implement <code>Matcher</code>.
 */
public class MatcherFactory {
    private static final String MATCHER_CLASS_NAME_KEY =
                    "se.leiflindback.rentcar.matcher.classname";
    private static final String CAR_TO_PROMOTE_KEY =
                    "se.leiflindback.rentcar.matcher.promote";

    /**
     * Returns a <code>Matcher</code> performing the default
     * matching algorithm. The class name of the default
     * <code>Matcher</code> implementation is read from the
     * system property
     * <code>se.leiflindback.rentcar.matcher.classname</code>
     *
     * @return The default matcher
     * @throws ClassNotFoundException If unable to load the
     *         default matcher class.
     * @throws InstantiationException If unable to instantiate
     *         the default matcher class.
     * @throws IllegalAccessException If unable to instantiate
     *         the default matcher class.
     */
    public Matcher getDefaultMatcher() throws
                                        ClassNotFoundException,
                                        InstantiationException,
                                        IllegalAccessException {
        String className =
                    System.getProperty(MATCHER_CLASS_NAME_KEY);
        Class matcherClass = Class.forName(className);
        Matcher matcher =
                    (Matcher) matcherClass.
                    getDeclaredConstructor().newInstance();
        if (matcher instanceof PromotingMatch) {
            ((PromotingMatch)matcher).
            setCarToPromote(System.
            getProperty(CAR_TO_PROMOTE_KEY));
        }
        return matcher;
    }
}
```

Listing 9.26 The `MatcherFactory` class, which hands out a matching algorithm.

After the class name of the appropriate matcher is read from the system property, it still remains to create an object of that class. Remember, from listing 9.8, that it is possible to create an object of a class, whose name is the content of a `String` variable. This is done on lines 33-34 in listing 9.26.

The goal of removing all references to `Matcher` implementations from `CarRegistry` has been reached now that the factory has been introduced, as can be seen from listing 9.27. This has taken us closer to the point where we can change matching algorithm without having to restart the program, but not all the way there. To completely reach that goal, we need a mechanism to first change the value of the system property, and then to force the factory to read the new value. How to do this is discussed in the *Comments* subsection, below.

```
1  /**
2   * Search for a car that is not booked, and that matches the
3   * specified search criteria.
4   *
5   * @param searchedCar This object contains the search
6   *                    criteria. Fields in the object that are
7   *                    set to <code>null</code> or zero are
8   *                    ignored.
9   * @return A description matching the searched car's
10  *         description if an unbooked car with the
11  *         same features as <code>searchedCar</code> was
12  *         found, <code>null</code> if no such car was found.
13  * @throws CarRegistryException If unable to search for a
14  *                              matching car.
15  */
16 public CarDTO findAvailableCar(CarDTO searchedCar) {
17     List<CarDTO> allCars = new ArrayList<>();
18     for (CarData car : cars) {
19         if (!car.booked) {
20             allCars.add(new CarDTO(car.regNo,
21                         new Amount(car.price), car.size,
22                         car.AC, car.fourWD, car.color,
23                         car.booked));
24         }
25     }
26     try {
27         return new MatcherFactory().getDefaultMatcher().
28                     match(searchedCar, allCars);
29     } catch (ClassNotFoundException | InstantiationException |
30              IllegalAccessException ex) {
31         throw new CarRegistryException(
32                   "Unable to instantiate matcher", ex);
33     }
34 }
```

Listing 9.27 The call from `CarRegistry` to the factory (line 27). There is no dependency on the `Matcher` implementations

Comments

- **How shall the factory know what to create?** There are many different ways to configure the factory, for example using a system property, as in the case study above. Other alternatives are to have the factory read from a file, or to create a user interface where the product can be specified. It might also be appropriate to add a method to the factory, that can be used to define the product to create, for example `public void setDefaultProduct(Product product)`. No matter how the class name of the desired product is communicated to the factory, there must also be a way to tell the factory to read it. It is not appropriate having to decide which product to use before the program is started, we also want to be able to change product during execution. If this change is to be initiated by something outside the program, input must be accepted. There could for example be a user interface which allows a system administrator to specify a new product and make the factory read it.

  The product created by the factory in the case study did not depend on program state in any way. All matching algorithms were always allowed, no matter which cars were available or what the customer wished. This is not always the case, we could for example add another factory, that creates products for payment handling. If the customer pays cash, the factory shall create a CashPayment, if the customer pays with credit card, a CreditCardPayment shall be created. In this case, there is only one allowed product. That means the factory's method that creates a product must take a parameter that enables deciding which is the correct product. The method definition could be for example `public PaymentHandler createPaymentHandler(PaymentType type)`.

- **Try to make the factory independent of concrete products.** One of the concrete products in the case study requires initialization data, namely `PromotingMatch`, which needs the registration number of the car to promote. This data is passed on lines 36-38 in listing 9.26 above. There is a problem with these lines, the class name `PromotingMatch` is mentioned, creating a coupling to a concrete product. That is a dangerous road to take, dependency on concrete products means code must be changed when new products are added, or existing products are changed. We could get rid of this dependency by passing initialization data to all concrete products, whether they need it or not. A new method, `void init(Map<String, Object> initData)`, should be added to the `Matcher` interface. This method should have a `java.util.Map` parameter, with key-value pairs containing all data

needed by the matching algorithm. `PromotingMatch` then requires such a map where the registration number is specified. The other matching algorithms, `PerfectMatch` and `WildCardMatch`, do not need init parameters and would therefore have empty `init` methods. If we made this change, lines 33-40 of listing 9.26 would instead be as in listing 9.28, where there is no dependency on any concrete product.

```
1 Matcher matcher =
2   (Matcher)Class.forName(className).newInstance();
3     //read initialization data, for example from
4     //a file, and place it in a java.util.Map
5     //called initData.
6 matcher.init(initData);
7 return matcher;
```

Listing 9.28 Configuration of products without dependency on any concrete product

- **Products can be cached.** The suggested solution above creates a new concrete product each time `getDefaultMatcher` is called. If the product is needed often and is time consuming to create, it might be a better idea to save one instance of each class. Those instances could be created when the factory is created, in order not having to spend any time on creation when a product is requested.

- **Who creates the factory itself?** In listing 9.27, line 27, `CarRegistry` creates a new instance of the factory each time it is needed. That is fine in the case study program, but what if it is time consuming to create the factory, maybe because it creates cached instances of concrete products as suggested in the previous bullet? And what if the factory reads a configuration file as suggested above, which is also time consuming? If, for any reason, it is not appropriate to create new factory instances, there must be a way to guarantee that there is only one, single, instance. There must also be a way to share that instance between all methods where it is needed. This is the purpose of the next pattern.

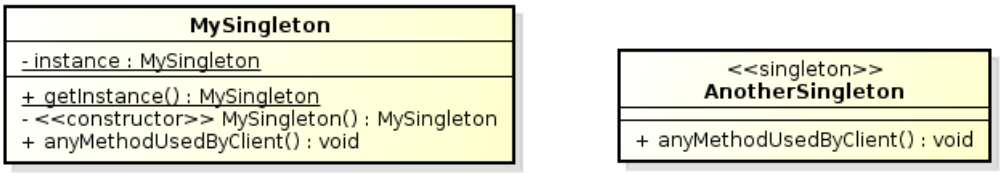
The Singleton Pattern

Sometimes it is not possible to allow more than one instance of a particular class. An example is the logging API created in section 9.2. If there is more than one instance of `ConsoleLogger`, the outputs of the loggers will be mixed on the screen, and if there is more than one instance of `FileLogger`, the outputs will be mixed in the file. To handle this situation, the program must both inhibit creating more objects, and expose the single instance to all classes needing it.

Figure 9.18 A singleton set has only one member, there are never any more. Image by unknown creator [Public domain], via https://pixabay.com

Overview

To block instantiation, the constructor of the critical object is made private. That means it can only be called by the class itself, no other class can create instances. Next, one single instance is created by the class itself, and saved in a static field. Last, a method is written, which hands out this sole instance. A class with these properties is called a *singleton*. The class diagram in figure 9.19a illustrates these features. Remember that underlining a member means it is static. Since any singleton, by definition, has such a static method and field, and private constructor, they are sometimes omitted and instead indicated with a stereotype, as in figure 9.19b.



(a) (b)

Figure 9.19 Class diagram illustrating a singleton.
(a) The static method and field, and the private constructor are depicted.
(b) The singleton-specific members from figure 9.19a are replaced with a stereotype.

Case Study

As a case study of the singleton pattern, consider the factory class created above. A factory is often a candidate to become a singleton, especially if the objects it creates shall be cached.

Solution

The solution in listing 9.29 is pretty straightforward. The static member holding the only existing instance is on line seven, the static method handing out this instance is on lines 12-14 and the private constructor on lines 16-17. Listing 9.30 shows how the client, `CarRegistry`, calls `getFactory` on line 12, in order to retrieve the only existing instance of the singleton.

```
/**
 * A Singleton that creates instances of the algorithm used
 * for matching a customer's wishes against available cars.
 * All such instances must implement <code>Matcher</code>.
 */
public class MatcherFactory {
    private static final MatcherFactory MATCHER_FACTORY =
                                            new MatcherFactory();
    /**
     * @return The only instance of this singleton.
     */
    public static MatcherFactory getFactory() {
        return MATCHER_FACTORY;
    }

    private MatcherFactory() {
    }

    // The getDefaultMatcher method has not been changed from
    // listing 9.26.
}
```

Listing 9.29 The `MatcherFactory` class, from listing 9.26, after it has been turned into a singleton.

```
public CarDTO findAvailableCar(CarDTO searchedCar) {
    List<CarDTO> allCars = new ArrayList<>();
    for (CarData car : cars) {
        if (!car.booked) {
            allCars.add(new CarDTO(car.regNo,
                        new Amount(car.price), car.size,
                        car.AC, car.fourWD, car.color,
                        car.booked));
        }
    }
    try {
            return MatcherFactory.getFactory().
                   getDefaultMatcher().match(searchedCar,
                                             allCars);
    } catch (ClassNotFoundException | InstantiationException |
             IllegalAccessException ex) {
        throw new CarRegistryException(
                  "Unable to instantiate matcher", ex);
    }
}
```

Listing 9.30 `CarRegistry` retrieves the factory instance and calls `getDefaultMatcher`. The only difference from listing 9.27 is line 12.

Comments

- **Why not make all members static, instead of creating a singleton?** This is a question that is often raised. In fact, there are many reasons, maybe the best is that a singleton is likely to have state in the form of instance variables. If all methods are static, then also those variables must be static, otherwise they can not be accessed by the static methods. Already here we are giving up object orientation, but it easily gets worse. The singleton probably has references to other objects, and when methods in those objects are called, there is a risk that the *make everything static* disease spreads also to those. Also, even if all members are static, we still need to add a private constructor to prohibit instantiating objects. In fact, the question is better asked the opposite way, *why make something static in an object-oriented program, if it can be avoided?* There are also many other reasons, for example it is easier to change a singleton into an ordinary object. Also, static methods can not be changed by inheritance, and static fields can not be serialized.

- **Is it never appropriate to create a class where all members are static?** Yes, in fact there is a situation where such a class is appropriate, namely a class that represents a purely procedural API, which will never have any state. An example of such a class is `java.lang.Math`, which contains mathematical functions, like trigonometric and exponential functions.

- **It is impossible to pass parameters to the constructor of a singleton**. Only the singleton itself can call its private constructor. For example, the singleton factory in the case study might have to read a configuration file to find out which classes to instantiate. The location of this file can not be passed to the constructor, and it seems as if it must be hard coded in the singleton itself. There is, however, a way around this problem, at least to some extent. It is possible to pass the file location as a system property, which can be used as in the solution to the factory case study above.

- **When is the singleton instance created?** The case study singleton instance is created when line seven in listing 9.29 is executed, but when is this? It is when static fields are initialized, which is when a class file is loaded by the JVM. Exactly when this happens varies, but the latest time a class can be loaded is when the JVM encounters the class name in the program.

The Composite Pattern

The *composite* pattern makes it possible to treat individual objects and groups of objects exactly the same way. This is appropriate when a certain task is to be performed, sometimes by just one object, other times by each object in a set of objects. The client shall be completely ignorant about the difference. Client code is identical when the task is performed once, by an individual implementation, and when it is performed multiple times, by different implementations.

Figure 9.20 The composite pattern removes the difference between a container and what is contained inside it. Image by unknown creator [Public domain], via https://pixabay.com

Overview

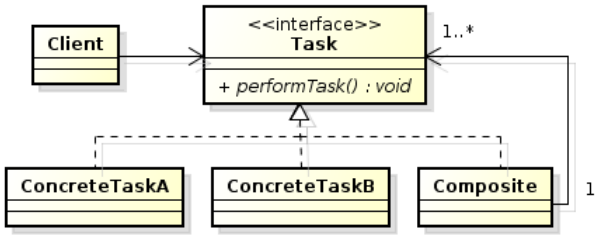


Figure 9.21 The composite implements the same interface as the concrete tasks, and wraps a number of concrete tasks.

As illustrated in figures 9.21 and 9.22, the client has a reference to an interface, and is ignorant about the implementations. As many examples above have shown, this is often the case, for example when using the strategy pattern. What is new here, is that there is one implementation of the interface, `CompositeTask`, which does not contain any concrete implementation. Instead, the composite has references to one or more concrete task. When called by the client, the composite in turn calls all its concrete tasks, and each of those perform the task. It is then up to the composite to combine the outcomes of the concrete tasks, and return the result to the client.

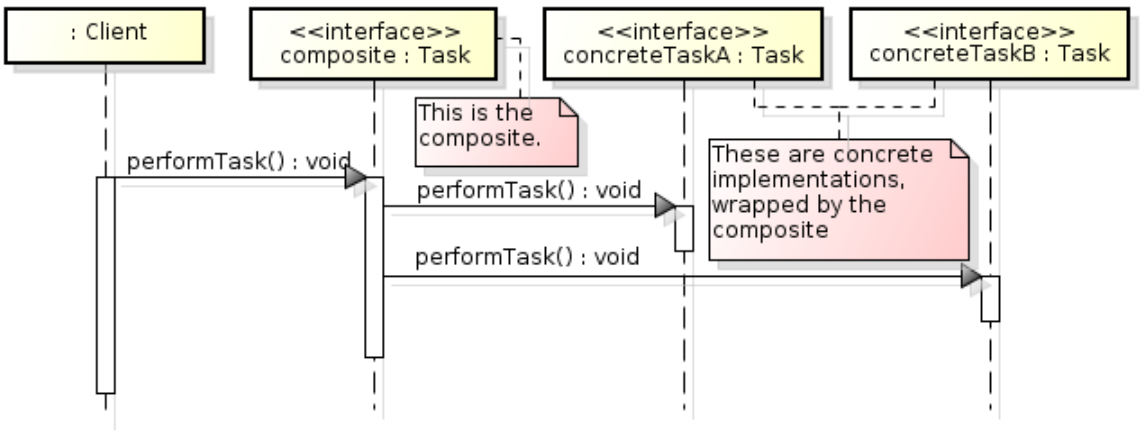


Figure 9.22 When called, the composite does not perform the task, but instead calls the concrete tasks it contains.

Case Study

Consider the car matching algorithm provided by `PromotingMatch`, which is listed in listing 9.24. This algorithm tries to match the searched car with the car that shall be promoted, and, if it fails, asks the `PerfectMatch` algorithm for a car (line 59). This is actually a hard-coded combination of two algorithms. Now, what if the fallback algorithm of `PromotingMatch` shall be changed? In this case it will be necessary to change the code on line 59 in `PromotingMatch`. And what if a new combining algorithm is introduced? Say for example `DiscountMatch`, which tries to give the customer a good deal by finding the car with the highest discount, and if no discount is found uses another algorithm. Maybe it shall be possible to change the fallback algorithm of both `DiscountMatch` and `PromotingMatch` at runtime. And maybe `DiscountMatch` uses `PromotingMatch` as fallback, or vice versa. All these situations prove it is a very bad solution to hard-code combinations of algorithms, as was done in `PromotingMatch`. Instead, a composite must be used.

Solution

Since all concrete algorithms shall be independent of each other, the hard-coded fallback algorithm in `PromotingMatch` is removed. All that is needed for this is to change line 59 in listing 9.24, to return `null` if no car was found. This indicates that the algorithm did not find a match.

The next step is to introduce a composite, which can be seen in listing 9.31. Note that this class implements the same interface, `Matcher`, as the other matching algorithms. However, no car matching is performed. Instead, `match`, which is supposed to search for cars, calls all wrapped algorithms. The results can be combined in any way. Here, the search is interrupted when an algorithm finds a car, and that car is then returned.

What about the empty `init` method on lines 53-55? This is the `init` method discussed above, in the section *Try to make the factory independent of concrete products*, on page 230. The purpose is to completely remove all hard-coded class names of `Matcher` implementations from the entire code. The addition of this method makes it possible to initialize `PromotingMatch` with the registration number of the car to promote, without mentioning the name of that class in `MatcherFactory`. Exactly how this is done can be seen in, `se.leiflindback.oodbook.rentcarWithExAndDesPat.integration.matchingWithComposite.MatcherFactory`, in the accompanying GitHub repository [Code].

```
1 /**
2  * A <code>Matcher</code>, which performs multiple matching
3  * algorithms. All matching algorithms added to this
4  * composite are executed, in the same order they were added,
5  * until an algorithm finds a car. Execution is stopped when
6  * an algorithm returns a non-null value.
7  */
8 class CompositeMatcher implements Matcher {
9     private List<Matcher> matchingAlgorithms =
```

```
10              new ArrayList<>();
11
12      CompositeMatcher() {
13      }
14
15      /**
16       * Invokes all matching algorithms added to this
17       * composite, in the same order they were added,
18       * until an algorithm finds a car. When a matching
19       * algorithm has found a car, that car is returned, and
20       * no more algorithms are called.
21       *
22       * @param searched  Search criteria
23       * @param available Available cars
24       * @return A matching car, or <code>null</code> if none
25       *         was found.
26       */
27      @Override
28      public CarDTO match(CarDTO searched,
29                          List<CarDTO> available) {
30          for (Matcher matcher : matchingAlgorithms) {
31              CarDTO found = matcher.match(searched,
32                                           available);
33              if (found != null) {
34                  return found;
35              }
36          }
37          return null;
38      }
39
40      /**
41       * Adds a matching algorithm that will be invoked when
42       * this composite is searching for a car. The newly added
43       * algorithm will be called after all previously added
44       * algorithms, provided non of the previous algorithms
45       * finds a matching car.
46       *
47       * @param matcher The new <code>Matcher</code> to add.
48       */
49      void addMatcher(Matcher matcher) {
50          matchingAlgorithms.add(matcher);
51      }
52
53      @Override
54      public void init(Map<String, String> properties) {
55      }
```

```
56 }
```

Listing 9.31 `CompositeMatcher` wraps the concrete matchers which are passed to `addMatcher` on lines 49-51. The `match` method calls these concrete matchers in the order they were added, until a matcher finds a car.

How is the composite populated with `Matchers`? This is done by the factory. When it is asked to produce a concrete product, it reads the system property `se.leiflindback.rentcar.matcher.classname`, just as previously. The difference now, is that this property may contain not just one class name of a concrete matcher, but a comma-separated list of such names. If more than one class is mentioned in the property, the factory creates a `CompositeMatcher`, and adds all specified classes to it. The private method responsible for creating the composite can be seen in listing 9.32.

```
1 private Matcher createComposite(String[] classNames) {
2     CompositeMatcher composite = new CompositeMatcher();
3     for (String className : classNames) {
4         composite.addMatcher(instantiateMatcher(className));
5     }
6     return composite;
7 }
```

Listing 9.32 Creation of composite matcher in `MatcherFactory` (exception handling is omitted).

Comments

- **But, there is still a hard-coded class name.** Yes, `CompositeMatcher` is mentioned on line two in listing 9.32. The question is if that class is considered part of the configurable algorithms, or if it is part of the infrastructure supporting those. In the previous case, it should not be mentioned, whereas it might very well be mentioned in the latter case. It can be argued that `CompositeMatcher` is, in fact, part of the infrastructure. There is only one composite, and it is used exactly the same way whenever algorithms are to be combined. If, on the other hand, there were more composites, and it should be possible to choose which to use, also the composite could be read from a system property, just like the concrete algorithms.
- **How to change the algorithm for combining outcomes of concrete algorithms wrapped by a composite?** Currently, `CompositeMatcher` always returns the car found by the first algorithm that did find a car. Maybe this way of choosing a result must be changeable. It might be desirable to have the composite collect all cars found by all its algorithms, and then choose for example the cheapest or the most expensive, depending on whether customer satisfaction or short-term income shall be maximized. In this situation, there

must be one composite `Matcher` implementation for each desired way to combine outcomes of matching algorithms. All that is needed to achieve this, is to change a few lines in `MatcherFactory`. Instead of always instantiating the same composite, the factory now must read also the class name of the composite from a system property, or whatever method is used to convey that information to the factory.

The Template Method Pattern

It is sometimes the case that the code in one class is very similar to that in another class, except in some detail. The principal strategy to remove the duplicated code is to introduce a new method, which contains the common code and can be called by both classes. Unfortunately, this strategy does not work if the duplicated code forms the structure of a method. The duplicated code might for example be a `try-catch` block or an `if` statement, while the content of the blocks differ. In this situation, the *template method* pattern is needed.

Figure 9.23 A template leaves out only the parts specific for a particular situation. Image by unknown creator [Public domain], via https://pixabay.com

Overview

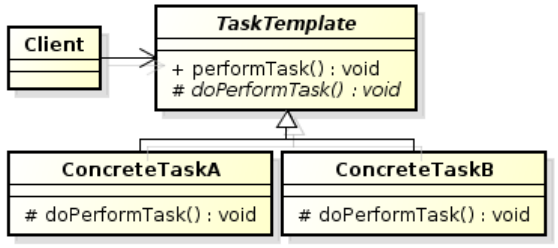
This pattern tells us to create a template, which, just like the invitation card in figure 9.23, contains everything common for all situations, and leaves out only the specific parts. The template will be an abstract class, `TaskTemplate` in figure 9.24a, containing a method, `performTask`, with the common code. This method calls an abstract method, `doPerformTask`, when it is time to execute the part of the code that differs between concrete implementations of the task. This call will arrive at one of the concrete subclasses, `ConcreteTaskA` in the example in figure 9.24b, which has overridden the abstract method and provided an implementation.

Case Study

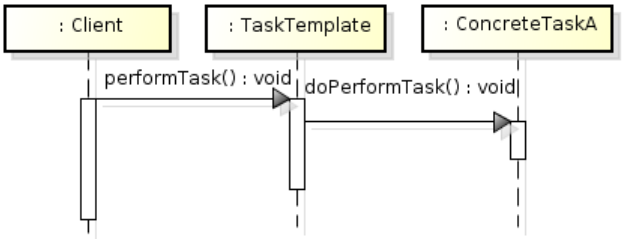
The case study concerns the display showing the number of rented cars, which was created as a case study of the observer pattern. This display shows how many cars of different types have currently been rented. The program shall now be extended to include two such displays, one with a text view and one with a GUI view.

Solution

The source code of the display class can be seen in listing 9.18. The overridden method from the `RentalObserver` interface, `newRental` on lines 20-24, calls two private meth-



(a) Code that is common for performing the task of the two concrete classes, is placed in `performTask`, in the abstract template class.



(b) The `performTask` method in the abstract template class calls a concrete implementation of `doPerformTask`, when it is time to execute code that differs between concrete implementations.

Figure 9.24 Template method class and sequence diagrams.

ods. The first, `addNewRental`, stores information about the latest rental, and is identical for the text and GUI views, which means it can be placed in the abstract superclass, `RentedCarsDisplay`, and be used in both concrete view subclasses. The second private method called from `newRental`, namely `printCurrentState`, differs between the views. Following the template method pattern, it must be promoted from `private` to `protected` visibility, and overridden in the concrete views. Listing 9.33 show the abstract template class, `RentedCarsDisplay`. Note the abstract protected method on lines 38-39. Listings 9.34 and 9.35 show the two concrete views. They contain only the implementations of `printCurrentState`, all other code is in the template superclass. Finally, listing 9.36 shows part of the main view, `View`, where the concrete classes are created.

```
1  /**
2   * Shows a running total of rented cars of each type.
3   */
4  public abstract class RentedCarsDisplay implements
5                                          RentalObserver {
6      private Map<CarDTO.CarType, Integer> noOfRentedCars =
7              new HashMap<>();
8
9      /**
10      * Creates a new instance, with the all counters of rented
11      * cars set to zero.
12      */
13     protected RentedCarsDisplay() {
14         for (CarDTO.CarType type : CarDTO.CarType.values()) {
15             noOfRentedCars.put(type, 0);
16         }
17     }
18
19     @Override
20     public void newRental(CarDTO rentedCar) {
21         addNewRental(rentedCar);
```

```
22          printCurrentState(noOfRentedCars);
23      }
24
25      private void addNewRental(CarDTO rentedCar) {
26          int noOfRentedCarsOfThisType =
27              noOfRentedCars.get(rentedCar.getSize()) + 1;
28          noOfRentedCars.put(rentedCar.getSize(),
29                              noOfRentedCarsOfThisType);
30      }
31
32      /**
33       * Shows the number of rented cars.
34       *
35       * @param noOfRentedCars Contains the number of rented
36       * cars of each type.
37       */
38      protected abstract void printCurrentState(
39                  Map<CarDTO.CarType, Integer> noOfRentedCars);
40  }
```

Listing 9.33 The template class, which leaves the adaptable parts in an abstract protected method, lines 38-39

```
1   /**
2    * Prints the number of rented cars on the console.
3    */
4   public class ConsoleRentedCarsDisplay extends
5                                       RentedCarsDisplay {
6       /**
7        * Prints the number of rented cars of each type on the
8        * console.
9        */
10      @Override
11      protected void printCurrentState(
12                  Map<CarDTO.CarType, Integer> noOfRentedCars) {
13          System.out.println("### We have now rented out ###");
14          for (CarDTO.CarType type : CarDTO.CarType.values()) {
15              System.out.print(noOfRentedCars.get(type));
16              System.out.print(" ");
17              System.out.print(type.toString().toLowerCase());
18              System.out.println(" cars.");
19          }
20          System.out.println("##############################");
21      }
22  }
```

Listing 9.34 A concrete implementation of the template. It contains only code that is specific for this particular implemenation

```
/**
 * Shows a GUI with the number of rented cars.
 */
public class GuiRentedCarsDisplay extends RentedCarsDisplay {
    private Display display;

    /**
     * Starts the GUI.
     */
    public GuiRentedCarsDisplay() {
        new Thread(() -> {
            Application.launch(Display.class, null);
        }).start();
        this.display = Display.getDisplay();
    }

    /**
     * Shows a GUI with the number of rented cars of
     * each type.
     */
    @Override
    protected void printCurrentState(
              Map<CarDTO.CarType, Integer> noOfRentedCars) {
        display.updateStatsList(noOfRentedCars);
    }

    // Code for GUI handling.
}
```

Listing 9.35 A concrete implementation of the template. It contains only code that is specific for this particular implemenation

```
/**
 * This program has no view, instead, this class is a
 * placeholder for the entire view.
 */
public class View {
    private Controller contr;
    private ErrorMessageHandler errorMsgHandler =
            new ErrorMessageHandler();
    private LogHandler logger = LogHandler.getLogger();

```

```
11      /**
12       * Creates a new instance.
13       *
14       * @param contr The controller that is used for all
15       * operations.
16       */
17      public View(Controller contr) throws IOException {
18          this.contr = contr;
19          contr.addRentalObserver(
20                  new ConsoleRentedCarsDisplay());
21          contr.addRentalObserver(new GuiRentedCarsDisplay());
22      }
23
24      // More methods, not relevant for this listing.
25  }
```

Listing 9.36 Creating the concerete implementations, lines 19-21.

Comments

- **More abstract methods** In all examples mentioned here, there was only one abstract method in the template class. This is not a requirement, we are free to add any number of abstract methods, consider for example the template class delineated in listing 9.37

```
1   /**
2    * An example of a template class with more than one abstract
3    * method.
4    */
5   public abstract class TemplateWithMoreAbstractMethods {
6       public void theTemplateMethod() {
7           try {
8               performSomeTask();
9           } catch (Exception e) {
10              errorHandling();
11          } finally {
12              cleanup();
13          }
14      }
15
16      protected abstract void performSomeTask();
17
18      protected abstract void errorHandling();
19
20      protected abstract void cleanup();
```

```
21 }
```

Listing 9.37 A template class with more than one abstract method

Chapter 10

What's Next?

After having read and practiced the contents of this book, we have a sufficient understanding of the basics of software design and architecture, and are ready to go further. There is, however, an immense amount of material on this topic, which makes it hard to see where to continue. Here follows suggestions of resources that might be good next steps for those that want to learn more. The suggestions are *not ordered*, neither by priority nor in any other sense. Before listing resources, a general advice about patterns: don't focus too much on the fact that something is presented as a pattern or whether a particular piece of code follows a particular pattern. Instead, try to understand the design advantage the pattern introduces.

One topic that is both useful and interesting is domain-driven design (DDD), which could be called a design strategy, that is an alternative to the analysis and design methods used in this book. DDD is however quite similar to the methods used in this book. The main difference is perhaps that in DDD the domain model is not just an inspiration for the design, but it is directly used as the design class diagram. There are many reasons to study DDD. First, knowing different methods to do the same thing (analysis and design) gives a deeper understanding. Second, DDD is much used and often mentioned in literature on design. Third, DDD is a bit more advanced than the methods used in this book, and provides answers to problems not covered here. The main DDD resource is the book that introduced it, E Evans: *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley Professional, 2003, ISBN: 9780321125217.

Another interesting topic is other architectural patterns, which can be used instead of MVC. Since the MVC idea has been used very extensively for a very long time, there are many variations. The same layer can have different names, and there are variations to the responsibilities of layers. Some suggestions for MVC alternatives to study are model-view-viewmodel (MVVM) and model-view-presenter (MVP). All these patterns split the system into layers the same way as in this book, but use slightly different layers. Also DDD, se above, introduces it's own layers instead of the M, V and C used here.

Something should also be said about the classics, GoF [GOF] (published 1994) and the refactorings book by Fowler [FOW] (published 2018, but the first edition, [FOW1ED], was published 1999). Are they really still relevant after almost 30 years? The answer is definitely 'yes, they are'. The GoF book introduced (or at least popularized) both design patterns and a way of thinking about design that is still a foundation of much of what is done today. Similarly, the refactorings book introduced (or popularized) refactorings, which are also still much used. However, be aware that it shows they were written many years ago, in the sense that

code examples use languages and APIs that are not much used today, for example C++ and Smalltalk in GoF.

Another important resource for those programming in Java is Joshua Bloch: *Effective Java, 3rd Ed*, Addison-Wesley, 2017, ISBN : 9780134685991. This book covers design and programming best practices for the Java language, it could be claimed that a Java program not following the suggestions mentioned here has somewhat bad smell. At least there is a big risk of bad smell if the suggestions of this book are broken without good reason. Note however that since this book is strongly connected to specifics of the Java language, only the latest edition is of interest, currently the third edition. Older editions are definitely outdated.

An interesting web page is the home page of Martin Fowler, the author of the refactoring book, `https://martinfowler.com/`. This web page contains a lot about design and architecture, of course we don't have to agree with everything, but it's definitely a great resource.

These were some tips concerning relevant next steps after having completed this book. There is of course an immense amount of resources besides those mentioned here, some better, some perhaps not very good. Also, much of the literature is domain specific, it is relevant only for a specific language or framework, or a specific kind of application. The bottom line is that no matter what software we are developing, we must always look for opinions and suggestions about *how* to write it. Preferably we should understand not just one, but different and contradicting opinions and suggestions. On the other hand, as has been mentioned many times in this book, deeper understanding comes with time and practice. We will not do everything the best way the first time.

Part III

Appendices

Appendix A

English-Swedish Dictionary

This appendix contains translations to Swedish of some English terms in the text. Be aware that these translations are terms commonly used in software development, and have a defined meaning. To know a general translation because of skills in English language is not enough, exactly the correct term must be used, not a synonym. Having said that, it is also important to point out that there is no universal agreement on these terms, do not be surprised when finding other words meaning the same thing.

English	*Swedish*
analysis	analys
architectural pattern	arkitekturellt mönster
architecture	arkitektur
class diagram	klassdiagram
code convention	kodkonvention
communication diagram	kommunikationsdiagram
design	design
design pattern	designmönster
domain model	domänmodell
encapsulation	inkapsling
entity	entitet
enumeration	uppräkningsbar typ
high cohesion	hög sammanhållning
implement	implementera
instance	instans
layer	lager
low coupling	låg koppling
modifier	modifierare
overload	överlagra
override	omdefiniera
package diagram	paketdiagram
pattern	mönster

English	***Swedish***
refactoring	omstrukturering, refaktorering, refaktorisering
sequence diagram	sekvensdiagram
system operation	systemoperation
system sequence diagram	systemsekvensdiagram
visibility	åtkomst

Appendix B

UML Cheat Sheet

Class Diagram

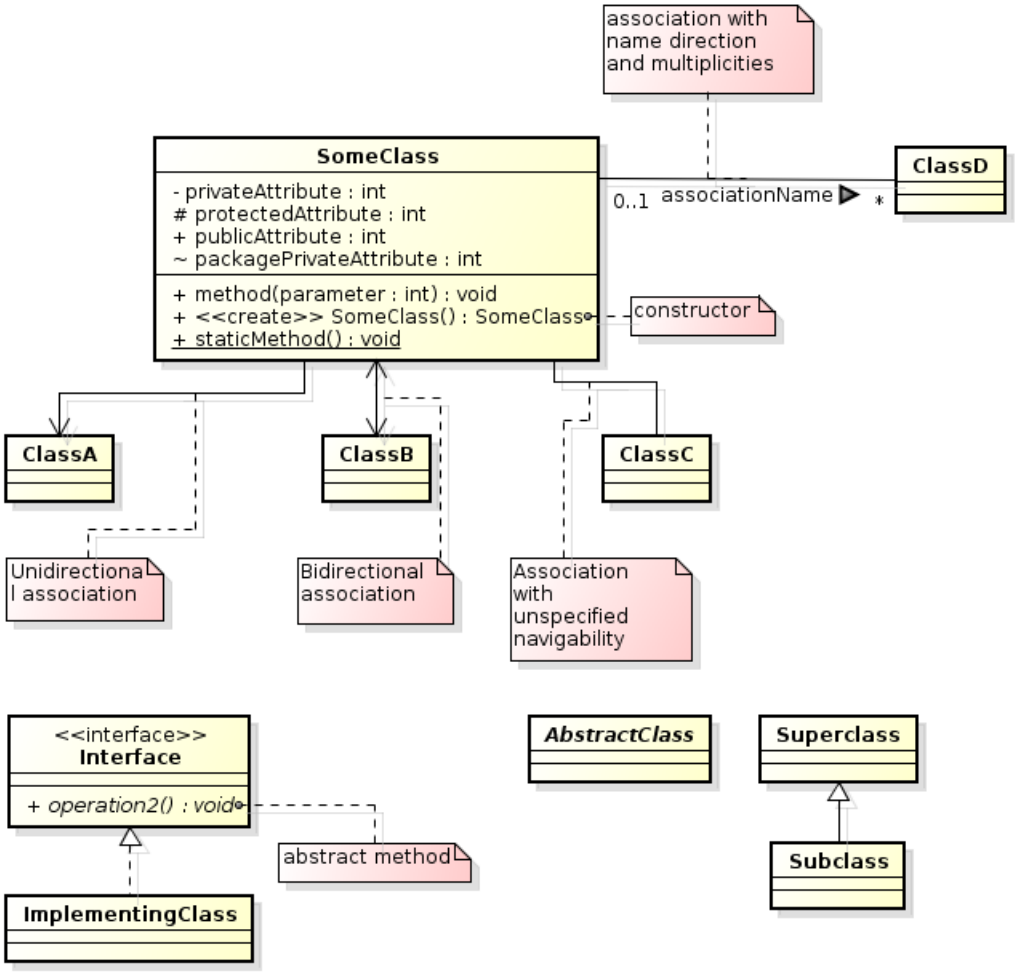


Figure B.1 Class diagram

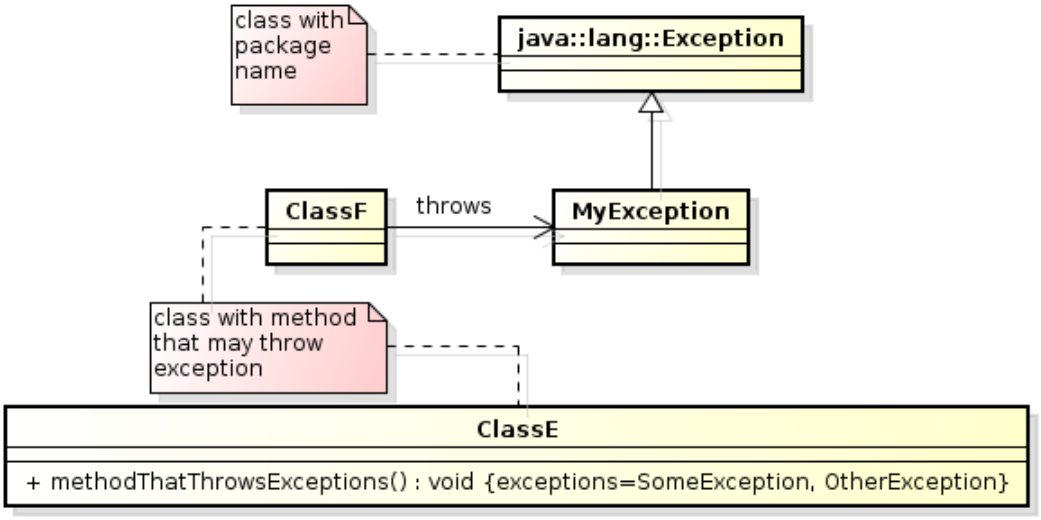


Figure B.2 Exceptions and package names in class diagram

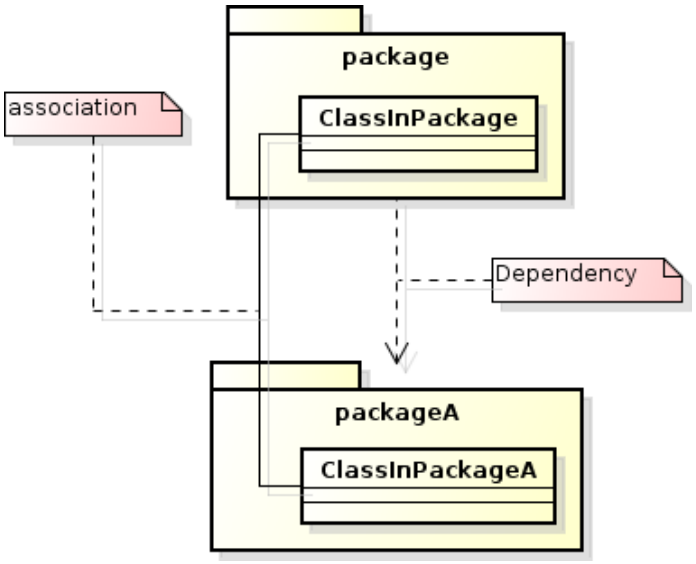


Figure B.3 Packages in class diagram

Sequence Diagram

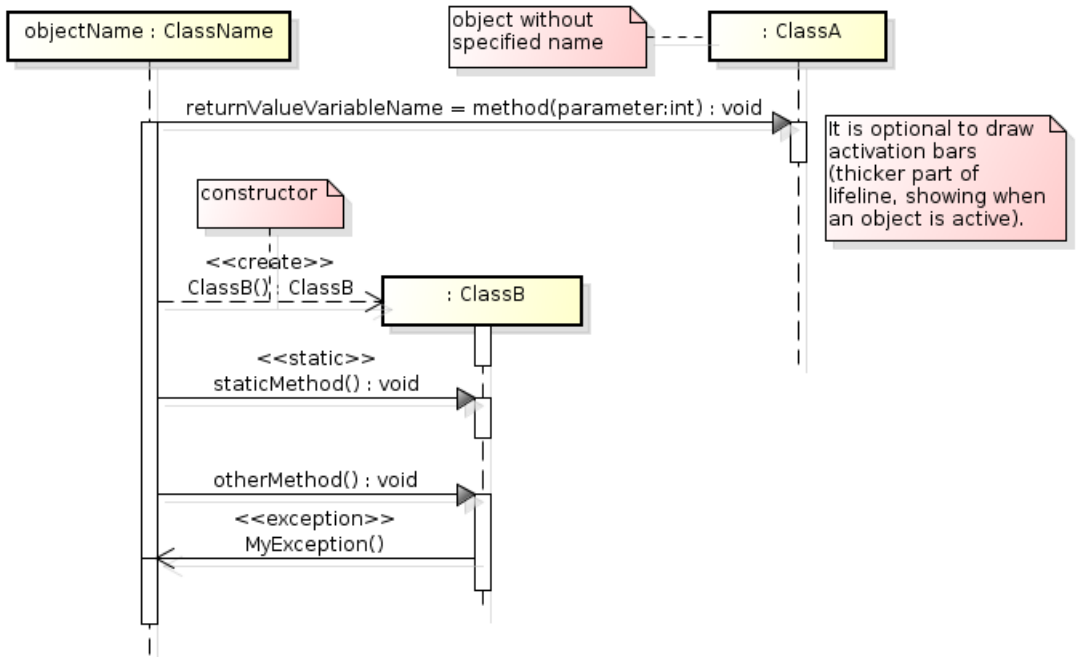


Figure B.4 Sequence diagram

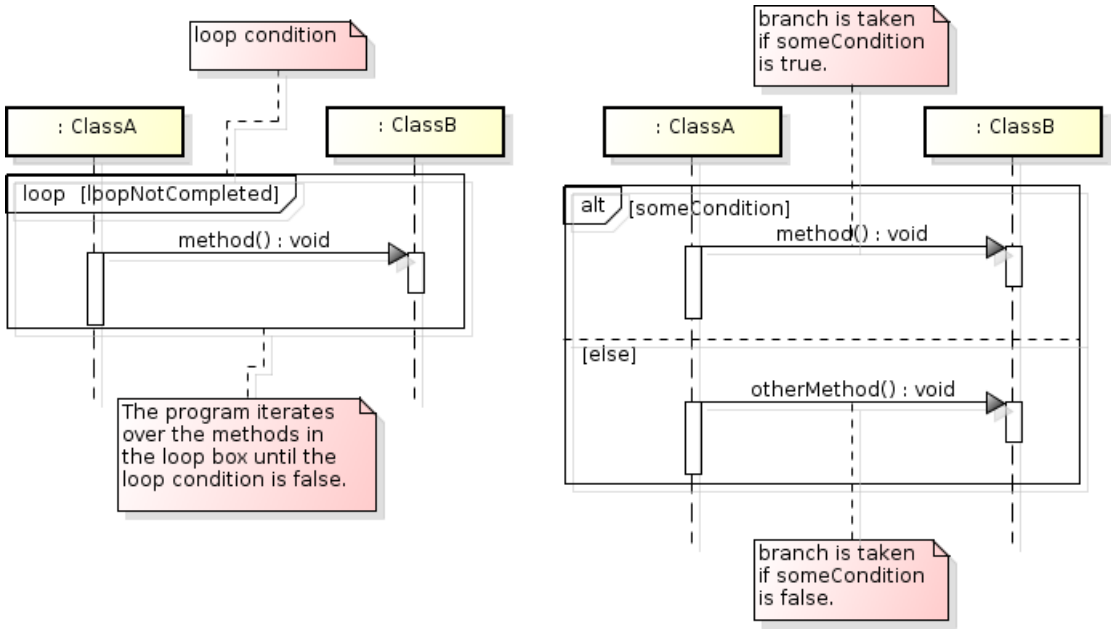


Figure B.5 Flow control in sequence diagram

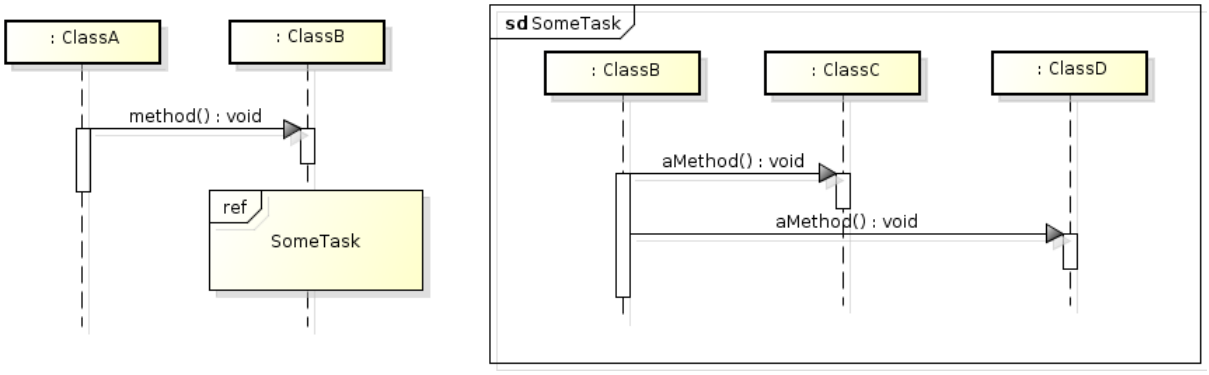


Figure B.6 Reference to other sequence diagram

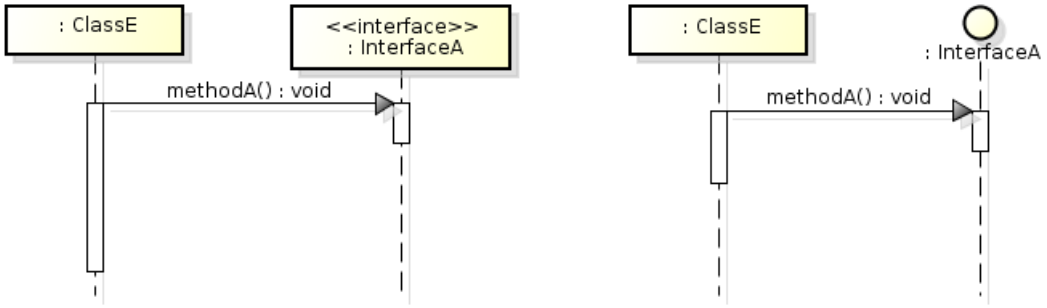


Figure B.7 A call to a method declared in an interface. The difference between the diagrams is the symbol used for the interface. Both symbols have the same meaning.

Communication Diagram

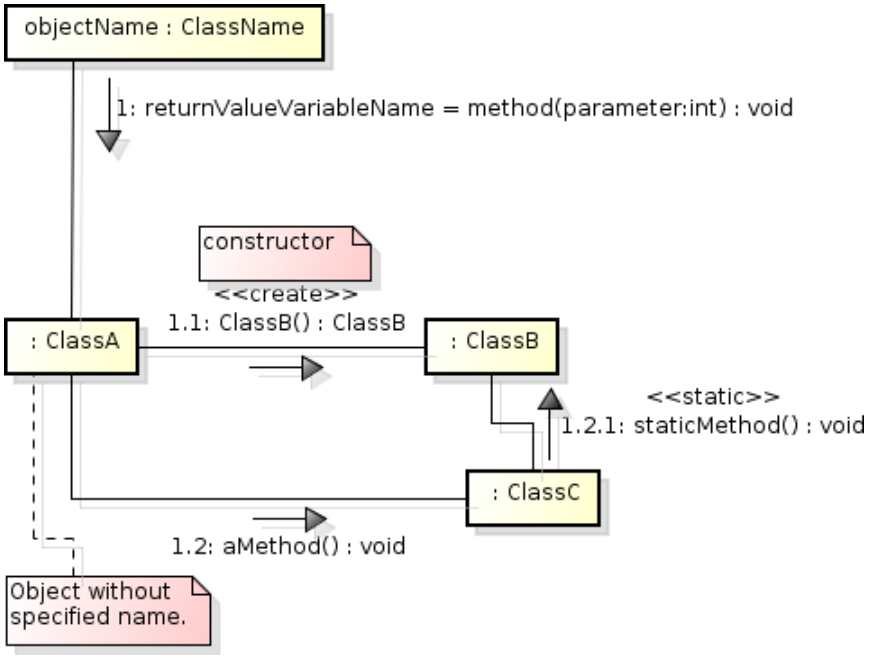


Figure B.8 Communication diagram

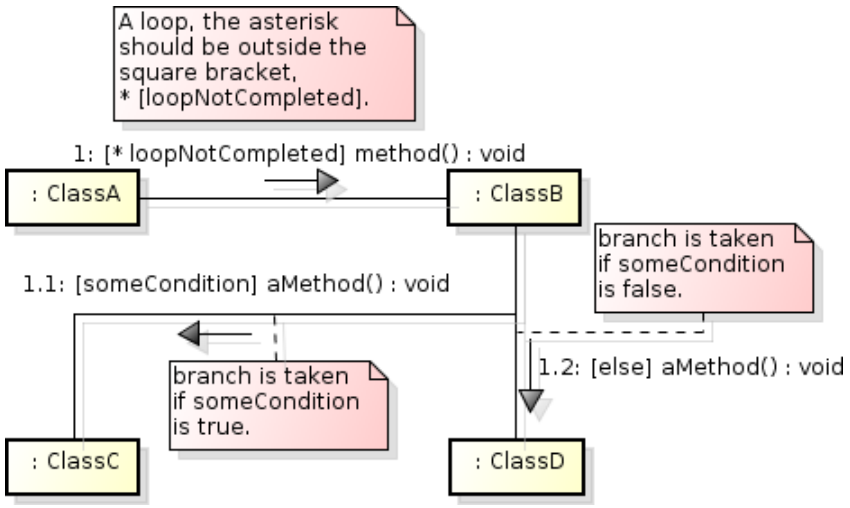


Figure B.9 Flow control in communication diagram

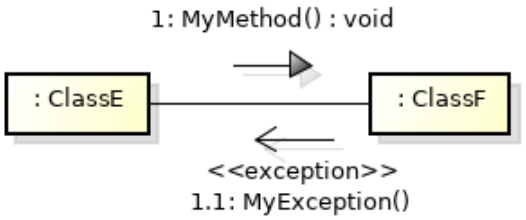


Figure B.10 Exception handling in communication diagram

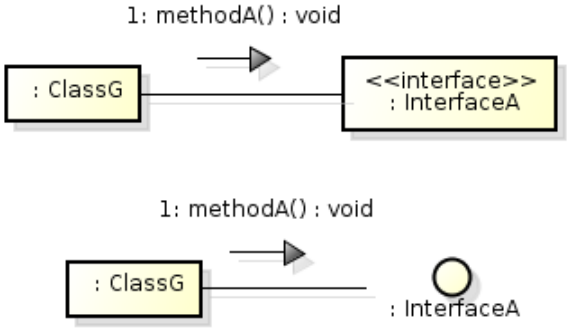


Figure B.11 A call to a method declared in an interface. The difference between the diagrams is the symbol used for the interface. Both symbols have the same meaning.

Appendix C

Implementations of UML Diagrams

This appendix contains Java implementations of all UML design diagrams in the text. The purpose is to make clearer what the diagrams actually mean. The analysis diagrams can not be implemented in code, since they do not represent programs.

C.1 Figure 5.1

Package names are not shown in the diagram, but have been added in the code.

```
1 package se.kth.ict.oodbook.design.uml;
2
3 public class AClass {
4     public void aMethod(int aParam) {
5     }
6 }
```

Listing C.1 Java code implementing the `AClass` class in figure 5.1a

```
1 package se.kth.ict.oodbook.design.uml;
2
3 public class AnotherClass {
4     private static int aStaticAttribute;
5
6     public static String aStaticMethod() {
7     }
8 }
```

Listing C.2 Java code implementing the `AnotherClass` class in figure 5.1b

```
1 package se.kth.ict.oodbook.design.uml;
2
3 public class YetAnotherClass {
4     private int privateAttribute;
5     public int publicAttribute;
```

```
6
7       private String privateMethod() {
8       }
9
10      public int publicMethod() {
11      }
12  }
```

Listing C.3 Java code implementing the `YetAnotherClass` class in figure 5.1c

C.2 Figure 5.2

The diagram tells `somePackage` in some way depends on `someOtherPackage`, but that can not be implemented in code since it does not tell how.

```
1  package somePackage;
```

Listing C.4 Java code implementing the `somePackage` package in figure 5.2

```
1  package someOtherPackage;
```

Listing C.5 Java code implementing the `someOtherPackage` package in figure 5.2

C.3 Figure 5.3

Package names are not shown in the diagram, but have been added in the code. Visibility is also not shown in the diagram. Here, attributes and methods called by objects where they are located have been assigned private visibility, while constructors and other methods have public visibility.

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class SomeClass {
4      private OtherClass otherObj;
5
6      public void firstMethod() {
7          otherObj.aMethod();
8          methodInSelf();
9      }
10
11     public void someMethod() {
12     }
```

```
13
14      private void methodInSelf() {
15      }
16  }
```

Listing C.6 Java code implementing the `SomeClass` class in figure 5.3

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class OtherClass {
4      private SomeClass someObj;
5
6      public void aMethod() {
7          someObj.someMethod();
8          ThirdClass newObj = new ThirdClass();
9      }
10 }
```

Listing C.7 Java code implementing the `OtherClass` class in figure 5.3

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class ThirdClass {
4      public ThirdClass() {
5      }
6  }
```

Listing C.8 Java code implementing the `ThirdClass` class in figure 5.3

C.4 Figure 5.4

Package names are not shown in the diagram, but have been added in the code. Visibility is also not shown in the diagram. Here, attributes and methods called by objects where they are located have been assigned private visibility, while other methods have public visibility.

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class A {
4      private B b;
5      // Somewhere in some method the following call is made:
6      // b.met1();
7  }
```

Listing C.9 Java code implementing the `A` class in figure 5.4

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class B {
4      public void met1() {
5          C.met2();
6      }
7      /*
8      * Code illustrated in a sequence diagram named 'SomeTask'.
9      */
10 }
```

Listing C.10 Java code implementing the `B` class in figure 5.4

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class C {
4      public static void met2() {
5
6      }
7      /*
8      * Code illustrated in a sequence diagram named 'SomeTask'.
9      */
10 }
```

Listing C.11 Java code implementing the `C` class in figure 5.4

C.5 Figure 5.5

Package names are not shown in the diagram, but have been added in the code. Visibility is also not shown in the diagram. Here, attributes and methods called by objects where they are located have been assigned private visibility, while constructors and methods have public visibility.

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class ClassA {
4      private ClassB objB;
5      private ClassE objE;
6
7      public void metF() {
8      }
9      /* The following lines appear somwhere in the code, in the
10      * order they are written here.
```

```
11      * objB.metA(2);
12      * int retVal = objE.metD();
13      * objE.metE();
14      */
15 }
```

Listing C.12 Java code implementing the `ClassA` class in figure 5.5

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class ClassB {
4      private ClassC objC;
5
6      public void metA(int aParam) {
7          objC.metB();
8          ClassD objD = new ClassD();
9      }
10 }
```

Listing C.13 Java code implementing the `ClassB` class in figure 5.5

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class ClassC {
4      public void metB() {
5      }
6  }
```

Listing C.14 Java code implementing the `ClassC` class in figure 5.5

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class ClassD {
4      public ClassD() {
5          myMethod();
6      }
7
8      private void myMethod() {
9      }
10 }
```

Listing C.15 Java code implementing the `ClassD` class in figure 5.5

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class ClassE {
4      private ClassA objA;
5
6      public void metE() {
7          objA.metF();
8      }
9
10     public int metD() {
11         return 0;
12     }
13 }
```

Listing C.16 Java code implementing the `ClassE` class in figure 5.5

C.6 Figure 5.6

Package names are not shown in the diagram, but have been added in the code. Visibility is also not shown in the diagram. Here, attributes and methods called by objects where they are located have been assigned private visibility, while constructors and methods have public visibility.

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class ClassF {
4      private int count;
5      private ClassG classG;
6      private ClassH classH;
7
8      /* The following code appears somewhere, in some method:
9       *   if (count == 3) {
10      *       classG.aMethod();
11      *   } else {
12      *       classH.aMethod();
13      *   }
14      */
15 }
```

Listing C.17 Java code implementing the `ClassF` class in figure 5.6

```
1  package se.kth.ict.oodbook.design.uml;
2
```

```
3  public class ClassG {
4      public void aMethod() {
5      }
6  }
```

Listing C.18 Java code implementing the `ClassG` class in figure 5.6

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class ClassH {
4      private int n;
5      private ClassK classK;
6
7      public void aMethod() {
8          for (int i = 1; i <= n; i++) {
9              classK.aMet();
10         }
11     }
12 }
```

Listing C.19 Java code implementing the `ClassH` class in figure 5.6

```
1  package se.kth.ict.oodbook.design.uml;
2
3  public class ClassK {
4      public void aMet() {
5      }
6  }
```

Listing C.20 Java code implementing the `ClassK` class in figure 5.6

C.7 Figure 5.9

The method bodies on line 17 in listing C.22 and line 17 in listing C.23 are not shown in the diagram in figure 5.9b. The code on those lines is included here since there is no reasonable alternative. The attribute on line eleven in listing C.23 is illustrated by the association in figure 5.9b. Package names are not shown in the diagram, but has been added in the code.

```
1  package se.kth.ict.oodbook.design.cohesion;
2
3  import java.util.List;
4
5  /**
```

```
 * Represents an employee.
 */
public class BadDesignEmployee {
    private String name;
    private Address address;
    private Amount salary;

    /**
     * Changes the salary of <code>employee</code> to
     * <code>newSalary</code>.
     *
     * @param employee The <code>Employee</code> whose salary will be
     *                 changed.
     * @param newSalary The new salary of <code>employee</code>.
     */
    public void changeSalary(BadDesignEmployee employee,
                             Amount newSalary) {
    }

    /**
     * Returns a list with all employees working in the same
     * department as this employee.
     */
    public List<BadDesignEmployee> getAllEmployees() {
    }
}
```

Listing C.21 Java code implementing the UML diagram in figure 5.9a

```
package se.kth.ict.oodbook.design.cohesion;

/**
 * Represents an employee.
 */
public class Employee {
    private String name;
    private Address address;
    private Amount salary;

    /**
     * Changes the salary to <code>newSalary</code>.
     *
     * @param newSalary The new salary.
     */
    public void changeSalary(Amount newSalary) {
        this.salary = newSalary;
```

```
18      }
19  }
```

Listing C.22 Java code implementing the `Employee` class in figure 5.9b

```
1  package se.kth.ict.oodbook.design.cohesion;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  /**
7   * Represents a department.
8   */
9  public class Department {
10     private String name;
11     private List<Employee> employees = new ArrayList<>();
12
13     /**
14      * Returns a list with all employees working in this department.
15      */
16     public List<Employee> getEmployees() {
17         return employees;
18     }
19 }
```

Listing C.23 Java code implementing the `Department` class in figure 5.9b

C.8 Figure 5.10

The method body on line 15 in listing C.24 is not shown in the diagram in figure 5.10a. The code on that line is included here since there is no reasonable alternative.

The method body on line 16 in listing C.25 is not shown in the diagram in figure 5.10b. The code on that line is included here since there is no reasonable alternative. The attributes on lines nine and ten in listing C.25 are illustrated by the associations in figure 5.10b. Package names are not shown in the diagram, but has been added in the code.

```
1  package se.kth.ict.oodbook.design.cohesion;
2
3  /**
4   * Represents a car.
5   */
6  public class BadDesignCar {
7      private String regNo;
8      private Person owner;
```

```
9       private String ownersPreferredRadioStation;
10
11      /**
12       * Returns the registration number of this car.
13       */
14      public String getRegNo() {
15          return regNo;
16      }
17
18      /**
19       * Accelerates the car.
20       */
21      public void accelerate() {
22
23      }
24
25      /**
26       * Breaks the car.
27       */
28      public void brake() {
29
30      }
31
32      /**
33       * Sets the car's radio to the specified station.
34       * @param station The station to which to listen.
35       */
36      public void changeRadioStation(String station) {
37
38      }
39  }
```

Listing C.24 Java code implementing the `BadDesignCar` class in figure 5.10a

```
1   package se.kth.ict.oodbook.design.cohesion;
2
3   /**
4    * Represents a car.
5    */
6   public class Car {
7       private String regNo;
8       private Person owner;
9       private Engine engine;
10      private Radio radio;
11
12      /**
```

```
13      * Returns the registration number of this car.
14      */
15     public String getRegNo() {
16         return regNo;
17     }
18 }
```

Listing C.25 Java code implementing the `Car` class in figure 5.10b

```
1  package se.kth.ict.oodbook.design.cohesion;
2
3  /**
4   * Represents a car radio.
5   */
6  public class Radio {
7      private String ownersPreferredStation;
8      /**
9       * Sets the radio to the specified station.
10      * @param station The station to which to listen.
11      */
12     public void changeStation(String station) {
13     }
14 }
```

Listing C.26 Java code implementing the `Radio` class in figure 5.10b

```
1  package se.kth.ict.oodbook.design.cohesion;
2
3  /**
4   * Represents a car engine.
5   */
6  public class Engine {
7      /**
8       * Accelerates the car.
9       */
10     public void accelerate() {
11     }
12
13     /**
14      * Breaks the car.
15      */
16     public void brake() {
17     }
18 }
```

Listing C.27 Java code implementing the `Engine` class in figure 5.10b

C.9 Figure 5.12

Package names are not shown in the diagram, but have been added in the code.

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 public class HighCouplingOrder {
4     private HighCouplingCustomer customer;
5     private HighCouplingShippingAddress shippingAddress;
6 }
```

Listing C.28 Java code implementing the `HighCouplingOrder` class in figure 5.12a

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class HighCouplingCustomer {
4     private HighCouplingShippingAddress shippingAddress;
5 }
```

Listing C.29 Java code implementing the `HighCouplingCustomer` class in figure 5.12a

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class HighCouplingShippingAddress {
4 }
```

Listing C.30 Java code implementing the `HighCouplingShippingAddress` class in figure 5.12a

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 public class Order {
4     private Customer customer;
5 }
```

Listing C.31 Java code implementing the `Order` class in figure 5.12b

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class Customer {
4     private ShippingAddress shippingAddress;
5 }
```

Listing C.32 Java code implementing the `Customer` class in figure 5.12b

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class ShippingAddress {
4 }
```

Listing C.33 Java code implementing the `ShippingAddress` class in figure 5.12b

C.10 Figure 5.13

Package names are not shown in the diagram, but have been added in the code.

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class HighCouplingBooking {
4 }
```

Listing C.34 Java code implementing the `HighCouplingBooking` class in figure 5.13a

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class HighCouplingGuest {
4 }
```

Listing C.35 Java code implementing the `HighCouplingGuest` class in figure 5.13a

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 public class HighCouplingHotel {
4     private HighCouplingBooking booking;
5     private HighCouplingGuest guest;
6     private HighCouplingAddress address;
7     private HighCouplingFloor floor;
8     private HighCouplingRoom room;
9 }
```

Listing C.36 Java code implementing the `HighCouplingHotel` class in figure 5.13a

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class HighCouplingAddress {
4 }
```

Listing C.37 Java code implementing the `HighCouplingAddress` class in figure 5.13a

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class HighCouplingFloor {
4 }
```

Listing C.38 Java code implementing the `HighCouplingFloor` class in figure 5.13a

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class HighCouplingRoom {
4 }
```

Listing C.39 Java code implementing the `HighCouplingRoom` class in figure 5.13a

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class Booking {
4     private Guest guest;
5 }
```

Listing C.40 Java code implementing the `Booking` class in figure 5.13b

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class Guest {
4 }
```

Listing C.41 Java code implementing the `Guest` class in figure 5.13b

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 public class Hotel {
4     private Booking booking;
5     private Address address;
6     private Floor floor;
```

```
7 }
```

Listing C.42 Java code implementing the `Hotel` class in figure 5.13b

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class Address {
4 }
```

Listing C.43 Java code implementing the `Address` class in figure 5.13b

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class Floor {
4     private Room room;
5 }
```

Listing C.44 Java code implementing the `Floor` class in figure 5.13b

```
1 package se.kth.ict.oodbook.design.coupling;
2
3 class Room {
4 }
```

Listing C.45 Java code implementing the `Room` class in figure 5.13b

C.11 Figure 5.15

Package name is not shown in the diagram, but has been added in the code.

```
1  package se.kth.ict.oodbook.architecture.packPriv;
2
3  /**
4   * Illustrates package private field and method. Note that it is
5   * not required to write javadoc for these, since they are not
6   * part of the public interface.
7   */
8  public class PackPriv {
9      int packagePrivateAtribute;
10
11     void packagePrivateMethod() {
12     }
13 }
```

Listing C.46 Java code implementing the `PackPriv` class in figure 5.15

C.12 Figure 5.18

```
1  package se.kth.ict.oodbook.architecture.mvc.controller;
2
3  /**
4   * This is the application's controller. All calls from view to model
5   * pass through here.
6   */
7  public class Controller {
8      /**
9       * A system operation, which means it appears in the system sequence
10      * diagram.
11      */
12     public void systemOperation1() {
13     }
14
15     /**
16      * A system operation, which means it appears in the system sequence
17      * diagram.
18      */
19     public void systemOperation2() {
20     }
21 }
```

Listing C.47 Java code implementing the `Controller` class in figure 5.18

C.13 Figure 5.19

```
1  package se.kth.ict.oodbook.architecture.mvc.view;
2
3  import se.kth.ict.oodbook.architecture.mvc.controller.Controller;
4
5  /**
6   * A class in the view.
7   */
8  public class ClassInView {
9      private Controller contr;
10
11     //Somewhere in some method.
12     contr.systemOperation1();
13 }
```

Listing C.48 Java code implementing the `ClassInView` class in figure 5.19

```
1  package se.kth.ict.oodbook.architecture.mvc.controller;
2
3  import se.kth.ict.oodbook.architecture.mvc.model.OtherClassInModel;
4  import se.kth.ict.oodbook.architecture.mvc.model.SomeClassInModel;
5
6  /**
7   * This is the application's controller. All calls from view to model
8   * pass through here.
9   */
10 public class Controller {
11     private SomeClassInModel scim;
12     private OtherClassInModel ocim;
13
14     /**
15      * A system operation, which means it appears in the system sequence
16      * diagram.
17      */
18     public void systemOperation1() {
19         scim.aMethod();
20         ocim.aMethod();
21     }
22 }
```

Listing C.49 Java code implementing the `Controller` class in figure 5.19

```
1  package se.kth.ict.oodbook.architecture.mvc.model;
2
3  /**
4   * A class in the model, performing some business logic.
5   */
6  public class SomeClassInModel {
7
8      /**
9       * Performs some business logic.
10      */
11     public void aMethod() {
12     }
13 }
```

Listing C.50 Java code implementing the `SomeClassInModel` class in figure 5.19

```
1  package se.kth.ict.oodbook.architecture.mvc.model;
2
3  /**
4   * A class in the model, performing some business logic.
5   */
6  public class OtherClassInModel {
7
8      /**
9       * Performs some business logic.
10      */
11     public void aMethod() {
12     }
13 }
```

Listing C.51 Java code implementing the `OtherClassInModel` class in figure 5.19

C.14 Figure 5.25

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.Car;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
```

```
 */
public class View {
        // Somewhere in the code. Note that the arguments to the
        // Car constructor are not specified in the UML diagram.
        Car searchedCar = new Car(0, null, false, false,
                                  null, null);
        Car foundCar = contr.searchMatchingCar(searchedCar);
    }

}
```

Listing C.52 Java code implementing the `View` class in figure 5.25

```
package se.kth.ict.oodbook.design.casestudy.controller;

import se.kth.ict.oodbook.design.casestudy.dbhandler.Car;

/**
 * This is the application's only controller class. All calls to
 * the model pass through here.
 */
public class Controller {
    public Car searchMatchingCar(Car searchedCar) {
    }
}
```

Listing C.53 Java code implementing the `Controller` class in figure 5.25

```
package se.kth.ict.oodbook.design.casestudy.dbhandler;

/**
 * Contains information about one particular car.
 */
public class Car {

    private int price;
    private String size;
    private boolean AC;
    private boolean fourWD;
    private String color;
    private String regNo;

    /**
     * Creates a new instance representing a particular car.
     *
```

```
18       * @param price The price paid to rent the car.
19       * @param size The size of the car, e.g., <code>medium
20                    hatchback</code>.
21       * @param AC      <code>true</code> if the car has air
22                       condition.
23       * @param fourWD  <code>true</code> if the car has four
24                       wheel drive.
25       * @param color The color of the car.
26       * @param regNo The car's registration number.
27       */
28      public Car(int price, String size, boolean AC, boolean fourWD,
29                 String color, String regNo) {
30          this.price = price;
31          this.size = size;
32          this.AC = AC;
33          this.fourWD = fourWD;
34          this.color = color;
35          this.regNo = regNo;
36      }
37
38      /**
39       * Get the value of regNo
40       *
41       * @return the value of regNo
42       */
43      public String getRegNo() {
44          return regNo;
45      }
46
47      /**
48       * Get the value of color
49       *
50       * @return the value of color
51       */
52      public String getColor() {
53          return color;
54      }
55
56      /**
57       * Get the value of fourWD
58       *
59       * @return the value of fourWD
60       */
61      public boolean isFourWD() {
62          return fourWD;
63      }
```

```
64
65      /**
66       * Get the value of AC
67       *
68       * @return the value of AC
69       */
70      public boolean isAC() {
71          return AC;
72      }
73
74      /**
75       * Get the value of size
76       *
77       * @return the value of size
78       */
79      public String getSize() {
80          return size;
81      }
82
83      /**
84       * Get the value of price
85       *
86       * @return the value of price
87       */
88      public int getPrice() {
89          return price;
90      }
91
92  }
```

Listing C.54 Java code implementing the `Car` class in figure 5.25

C.15 Figure 5.26

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
```

```
11          // Somewhere in the code. Note that the arguments to the
12          // CarDTO constructor are not specified in the UML
13          // diagram.
14          CarDTO searchedCar = new CarDTO(0, null, false, false,
15                                          null, null);
16          CarDTO foundCar = contr.searchMatchingCar(searchedCar);
17      }
18
19  }
```

Listing C.55 Java code implementing the `View` class in figure 5.26

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4
5  /**
6   * This is the application's only controller class. All calls to
7   * the model pass through here.
8   */
9  public class Controller {
10     public CarDTO searchMatchingCar(CarDTO searchedCar) {
11     }
12 }
```

Listing C.56 Java code implementing the `Controller` class in figure 5.26

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  /**
4   * Contains information about one particular car.
5   */
6  public class CarDTO {
7
8      private int price;
9      private String size;
10     private boolean AC;
11     private boolean fourWD;
12     private String color;
13     private String regNo;
14
15     /**
16      * Creates a new instance representing a particular car.
17      *
18      * @param price The price paid to rent the car.
```

```
19       * @param size The size of the car, e.g., <code>medium
20                    hatchback</code>.
21       * @param AC      <code>true</code> if the car has air
22                       condition.
23       * @param fourWD  <code>true</code> if the car has four
24                       wheel drive.
25       * @param color The color of the car.
26       * @param regNo The car's registration number.
27       */
28      public CarDTO(int price, String size, boolean AC,
29                    boolean fourWD, String color, String regNo) {
30          this.price = price;
31          this.size = size;
32          this.AC = AC;
33          this.fourWD = fourWD;
34          this.color = color;
35          this.regNo = regNo;
36      }
37
38      /**
39       * Get the value of regNo
40       *
41       * @return the value of regNo
42       */
43      public String getRegNo() {
44          return regNo;
45      }
46
47      /**
48       * Get the value of color
49       *
50       * @return the value of color
51       */
52      public String getColor() {
53          return color;
54      }
55
56      /**
57       * Get the value of fourWD
58       *
59       * @return the value of fourWD
60       */
61      public boolean isFourWD() {
62          return fourWD;
63      }
64
```

```
65      /**
66       * Get the value of AC
67       *
68       * @return the value of AC
69       */
70      public boolean isAC() {
71          return AC;
72      }
73
74      /**
75       * Get the value of size
76       *
77       * @return the value of size
78       */
79      public String getSize() {
80          return size;
81      }
82
83      /**
84       * Get the value of price
85       *
86       * @return the value of price
87       */
88      public int getPrice() {
89          return price;
90      }
91
92  }
```

Listing C.57 Java code implementing the `CarDTO` class in figure 5.26

C.16 Figure 5.27

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
11         // Somewhere in the code. Note that the arguments to the
```

```
12          // CarDTO constructor are not specified in the UML
13          // diagram.
14          CarDTO searchedCar = new CarDTO(0, null, false, false,
15                                          null, null);
16          CarDTO foundCar = contr.searchMatchingCar(searchedCar);
17      }
18
19  }
```

Listing C.58 Java code implementing the `View` class in figure 5.27

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4
5  /**
6   * This is the application's only controller class. All calls to
7   * the model pass through here.
8   */
9  public class Controller {
10     public CarDTO searchMatchingCar(CarDTO searchedCar) {
11         return carRegistry.findCar(searchedCar);
12     }
13 }
```

Listing C.59 Java code implementing the `Controller` class in figure 5.27

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  /**
4   * Contains information about one particular car.
5   */
6  public class CarDTO {
7
8      private int price;
9      private String size;
10     private boolean AC;
11     private boolean fourWD;
12     private String color;
13     private String regNo;
14
15     /**
16      * Creates a new instance representing a particular car.
17      *
18      * @param price The price paid to rent the car.
```

```
19          * @param size The size of the car, e.g., <code>medium
20                       hatchback</code>.
21          * @param AC      <code>true</code> if the car has air
22                          condition.
23          * @param fourWD  <code>true</code> if the car has four
24                          wheel drive.
25          * @param color The color of the car.
26          * @param regNo The car's registration number.
27          */
28         public CarDTO(int price, String size, boolean AC,
29                       boolean fourWD, String color, String regNo) {
30             this.price = price;
31             this.size = size;
32             this.AC = AC;
33             this.fourWD = fourWD;
34             this.color = color;
35             this.regNo = regNo;
36         }
37
38         /**
39          * Get the value of regNo
40          *
41          * @return the value of regNo
42          */
43         public String getRegNo() {
44             return regNo;
45         }
46
47         /**
48          * Get the value of color
49          *
50          * @return the value of color
51          */
52         public String getColor() {
53             return color;
54         }
55
56         /**
57          * Get the value of fourWD
58          *
59          * @return the value of fourWD
60          */
61         public boolean isFourWD() {
62             return fourWD;
63         }
64
```

```
65      /**
66       * Get the value of AC
67       *
68       * @return the value of AC
69       */
70      public boolean isAC() {
71          return AC;
72      }
73
74      /**
75       * Get the value of size
76       *
77       * @return the value of size
78       */
79      public String getSize() {
80          return size;
81      }
82
83      /**
84       * Get the value of price
85       *
86       * @return the value of price
87       */
88      public int getPrice() {
89          return price;
90      }
91
92  }
```

Listing C.60 Java code implementing the `CarDTO` class in figure 5.27

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  /**
4   * Contains all calls to the data store with cars that may be
5   * rented.
6   */
7  public class CarRegistry {
8      public CarDTO findCar(CarDTO searchedCar) {
9      }
10 }
```

Listing C.61 Java code implementing the `CarRegistry` class in figure 5.27

C.17 Figure 5.28

```
1  package se.kth.ict.oodbook.design.casestudy.startup;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5  import se.kth.ict.oodbook.design.casestudy.view.View;
6
7  /**
8   * Contains the <code>main</code> method. Performs all startup of
9   * the application.
10  */
11 public class Main {
12     public static void main(String[] args) {
13         CarRegistry carRegistry = new CarRegistry();
14         Controller contr = new Controller(carRegistry);
15         new View(contr);
16     }
17 }
```

Listing C.62 Java code implementing the `Main` class in figure 5.28

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  /**
4   * Contains all calls to the data store with cars that may be
5   * rented.
6   */
7  public class CarRegistry {
8  }
```

Listing C.63 Java code implementing the `CarRegistry` class in figure 5.28

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5
6  /**
7   * This is the application's only controller class. All calls to
8   * the model pass through here.
9   */
10 public class Controller {
11     public Controller(CarRegistry carRegistry) {
```

```
12      }
13  }
```

Listing C.64 Java code implementing the `Controller` class in figure 5.28

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
11     public View(Controller contr) {
12     }
13 }
```

Listing C.65 Java code implementing the `View` class in figure 5.28

C.18 Figure 5.29

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
11     private Controller contr;
12
13     /**
14      * Creates a new instance.
15      *
16      * @param contr The controller that is used for all operations.
17      */
18     public View(Controller contr) {
19     }
20 }
```

Listing C.66 Java code implementing the `View` class in figure 5.29

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5
6  /**
7   * This is the application's only controller class. All calls to
8   * the model pass through here.
9   */
10 public class Controller {
11     private CarRegistry carRegistry;
12
13     /**
14      * Creates a new instance.
15      *
16      * @param carRegistry Used to access the car data store.
17      */
18     public Controller(CarRegistry carRegistry) {
19     }
20
21     /**
22      * Search for a car matching the specified search criteria.
23      *
24      * @param searchedCar This object contains the search criteria.
25      *                    Fields in the object that are set to
26      *                    <code>null</code> or
27      *                    <code>false</code> are ignored.
28      * @return The best match of the search criteria.
29      */
30     public CarDTO searchMatchingCar(CarDTO searchedCar) {
31     }
32
33     /**
34      * Registers a new customer. Only registered customers can
35      * rent cars.
36      *
37      * @param customer The customer that will be registered.
38      */
39     public void registerCustomer(CustomerDTO customer) {
40     }
41 }
```

Listing C.67 Java code implementing the `Controller` class in figure 5.29

```
1  package se.kth.ict.oodbook.design.casestudy.startup;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5  import se.kth.ict.oodbook.design.casestudy.view.View;
6
7  /**
8   * Contains the <code>main</code> method. Performs all startup
9   * of the application.
10  */
11 public class Main {
12     public static void main(String[] args) {
13     }
14 }
```

Listing C.68 Java code implementing the `Main` class in figure 5.29

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  /**
4   * Contains all calls to the data store with cars that may be
5   * rented.
6   */
7  public class CarRegistry {
8      /**
9       * Creates a new instance.
10      */
11     public CarRegistry() {
12     }
13
14     /**
15      * Search for a car matching the specified search criteria.
16      *
17      * @param searchedCar This object contains the search criteria.
18      *                    Fields in the object that are set to
19      *                    <code>null</code> or
20      *                    <code>false</code> are ignored.
21      * @return The best match of the search criteria.
22      */
23     public CarDTO findCar(CarDTO searchedCar) {
24     }
```

```
25 }
```

Listing C.69 Java code implementing the `CarRegistry` class in figure 5.29

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  /**
4   * Contains information about one particular car.
5   */
6  public class CarDTO {
7
8      private int price;
9      private String size;
10     private boolean AC;
11     private boolean fourWD;
12     private String color;
13     private String regNo;
14
15     /**
16      * Creates a new instance representing a particular car.
17      *
18      * @param price The price paid to rent the car.
19      * @param size The size of the car, e.g.,
20      *             <code>medium hatchback</code>.
21      * @param AC      <code>true</code> if the car has
22      *                air condition.
23      * @param fourWD  <code>true</code> if the car has four
24      *                wheel drive.
25      * @param color The color of the car.
26      * @param regNo The car's registration number.
27      */
28     public CarDTO(int price, String size, boolean AC,
29                   boolean fourWD, String color, String regNo) {
30     }
31
32     /**
33      * Get the value of regNo
34      *
35      * @return the value of regNo
36      */
37     public String getRegNo() {
38     }
39
40     /**
41      * Get the value of color
42      *
```

```
43          * @return the value of color
44          */
45         public String getColor() {
46         }
47
48         /**
49          * Get the value of fourWD
50          *
51          * @return the value of fourWD
52          */
53         public boolean isFourWD() {
54         }
55
56         /**
57          * Get the value of AC
58          *
59          * @return the value of AC
60          */
61         public boolean isAC() {
62         }
63
64         /**
65          * Get the value of size
66          *
67          * @return the value of size
68          */
69         public String getSize() {
70         }
71
72         /**
73          * Get the value of price
74          *
75          * @return the value of price
76          */
77         public int getPrice() {
78         }
79
80 }
```

Listing C.70 Java code implementing the `CarDTO` class in figure 5.29

C.19 Figure 5.30

```
1 package se.kth.ict.oodbook.design.casestudy.view;
```

```
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
5  import se.kth.ict.oodbook.design.casestudy.model.AddressDTO;
6  import se.kth.ict.oodbook.design.casestudy.model.CustomerDTO;
7  import se.kth.ict.oodbook.design.casestudy.model.DrivingLicenseDTO;
8
9  /**
10  * This program has no view, instead, this class is a placeholder
11  * for the entire view.
12  */
13 public class View {
14     private Controller contr;
15
16     // Somewhere in the code. Note that the arguments to the
17     // DTO constructors are not specified in the UML
18     // diagram.
19     AddressDTO address = new AddressDTO("Storgatan 2", "12345",
20                                         "Hemorten");
21     DrivingLicenseDTO drivingLicense = new DrivingLicenseDTO(
22             "982193721937213");
23     CustomerDTO customer = new CustomerDTO("Stina", address,
24                                            drivingLicense);
25     contr.registerCustomer(customer);
26 }
```

Listing C.71 Java code implementing the `View` class in figure 5.30

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents a post address.
5   */
6  public final class AddressDTO {
7      private final String street;
8      private final String zip;
9      private final String city;
10
11     /**
12      * Creates a new instance.
13      *
14      * @param street Street name and number.
15      * @param zip    Zip code
16      * @param city   City (postort)
17      */
18     public AddressDTO(String street, String zip, String city) {
```

```
19          this.street = street;
20          this.zip = zip;
21          this.city = city;
22      }
23
24      /**
25       * Get the value of city
26       *
27       * @return the value of city
28       */
29      public String getCity() {
30          return city;
31      }
32
33      /**
34       * Get the value of zip
35       *
36       * @return the value of zip
37       */
38      public String getZip() {
39          return zip;
40      }
41
42      /**
43       * Get the value of street
44       *
45       * @return the value of street
46       */
47      public String getStreet() {
48          return street;
49      }
50
51  }
```

Listing C.72 Java code implementing the `AddressDTO` class in figure 5.30

```
1   package se.kth.ict.oodbook.design.casestudy.model;
2
3   /**
4    * Represents a driving license
5    */
6   public class DrivingLicenseDTO {
7       private final String licenseNo;
8
9       /**
10       * Creates a new instance.
```

```
11      *
12      * @param licenseNo The driving license number.
13      */
14     public DrivingLicenseDTO(String licenseNo) {
15         this.licenseNo = licenseNo;
16     }
17
18     /**
19      * Get the value of licenseNo
20      *
21      * @return the value of licenseNo
22      */
23     public String getLicenseNo() {
24         return licenseNo;
25     }
26
27 }
```

Listing C.73 Java code implementing the `DrivingLicenseDTO` class in figure 5.30

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents a customer of the car rental company.
5   */
6  public class CustomerDTO {
7      private final String name;
8      private final AddressDTO address;
9      private final DrivingLicenseDTO drivingLicense;
10
11     /**
12      * Creates a new instance.
13      *
14      * @param name           The customer's name.
15      * @param address        The customer's address.
16      * @param drivingLicense The customer's driving license.
17      */
18     public CustomerDTO(String name, AddressDTO address,
19                        DrivingLicenseDTO drivingLicense) {
20         this.name = name;
21         this.address = address;
22         this.drivingLicense = drivingLicense;
23     }
24
25     /**
26      * Get the value of drivingLicense
```

```
27          *
28          * @return the value of drivingLicense
29          */
30         public DrivingLicenseDTO getDrivingLicense() {
31             return drivingLicense;
32         }
33
34         /**
35          * Get the value of address
36          *
37          * @return the value of address
38          */
39         public AddressDTO getAddress() {
40             return address;
41         }
42
43         /**
44          * Get the value of name
45          *
46          * @return the value of name
47          */
48         public String getName() {
49             return name;
50         }
51
52 }
```

Listing C.74 Java code implementing the `CustomerDTO` class in figure 5.30

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5  import se.kth.ict.oodbook.design.casestudy.model.CustomerDTO;
6  import se.kth.ict.oodbook.design.casestudy.model.Rental;
7
8  /**
9   * This is the application's only controller class. All calls to
10  * the model pass through here.
11  */
12 public class Controller {
13     /**
14      * Registers a new customer. Only registered customers can
15      * rent cars.
16      *
17      * @param customer The customer that will be registered.
```

```
18      */
19     public void registerCustomer(CustomerDTO customer) {
20     }
21 }
```

Listing C.75 Java code implementing the `Controller` class in figure 5.30

C.20 Figure 5.31

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
5  import se.kth.ict.oodbook.design.casestudy.model.AddressDTO;
6  import se.kth.ict.oodbook.design.casestudy.model.CustomerDTO;
7  import se.kth.ict.oodbook.design.casestudy.model.DrivingLicenseDTO;
8
9  /**
10  * This program has no view, instead, this class is a placeholder
11  * for the entire view.
12  */
13 public class View {
14     private Controller contr;
15
16     // Somewhere in the code. Note that the arguments to the
17     // DTO constructors are not specified in the UML
18     // diagram.
19     AddressDTO address = new AddressDTO("Storgatan 2", "12345",
20                                         "Hemorten");
21     DrivingLicenseDTO drivingLicense = new DrivingLicenseDTO(
22             "982193721937213");
23     CustomerDTO customer = new CustomerDTO("Stina", address,
24                                            drivingLicense);
25     contr.registerCustomer(customer);
26 }
```

Listing C.76 Java code implementing the `View` class in figure 5.31

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4  * Represents a post address.
5  */
```

```
 6 public final class AddressDTO {
 7     private final String street;
 8     private final String zip;
 9     private final String city;
10 
11     /**
12      * Creates a new instance.
13      *
14      * @param street Street name and number.
15      * @param zip    Zip code
16      * @param city   City (postort)
17      */
18     public AddressDTO(String street, String zip, String city) {
19         this.street = street;
20         this.zip = zip;
21         this.city = city;
22     }
23 
24     /**
25      * Get the value of city
26      *
27      * @return the value of city
28      */
29     public String getCity() {
30         return city;
31     }
32 
33     /**
34      * Get the value of zip
35      *
36      * @return the value of zip
37      */
38     public String getZip() {
39         return zip;
40     }
41 
42     /**
43      * Get the value of street
44      *
45      * @return the value of street
46      */
47     public String getStreet() {
48         return street;
49     }
50 
51 }
```

Listing C.77 Java code implementing the `AddressDTO` class in figure 5.31

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents a driving license
5   */
6  public class DrivingLicenseDTO {
7      private final String licenseNo;
8
9      /**
10      * Creates a new instance.
11      *
12      * @param licenseNo The driving license number.
13      */
14     public DrivingLicenseDTO(String licenseNo) {
15         this.licenseNo = licenseNo;
16     }
17
18     /**
19      * Get the value of licenseNo
20      *
21      * @return the value of licenseNo
22      */
23     public String getLicenseNo() {
24         return licenseNo;
25     }
26
27 }
```

Listing C.78 Java code implementing the `DrivingLicenseDTO` class in figure 5.31

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents a customer of the car rental company.
5   */
6  public class CustomerDTO {
7      private final String name;
8      private final AddressDTO address;
9      private final DrivingLicenseDTO drivingLicense;
10
11     /**
```

```
12      * Creates a new instance.
13      *
14      * @param name           The customer's name.
15      * @param address        The customer's address.
16      * @param drivingLicense The customer's driving license.
17      */
18     public CustomerDTO(String name, AddressDTO address,
19                        DrivingLicenseDTO drivingLicense) {
20         this.name = name;
21         this.address = address;
22         this.drivingLicense = drivingLicense;
23     }
24
25     /**
26      * Get the value of drivingLicense
27      *
28      * @return the value of drivingLicense
29      */
30     public DrivingLicenseDTO getDrivingLicense() {
31         return drivingLicense;
32     }
33
34     /**
35      * Get the value of address
36      *
37      * @return the value of address
38      */
39     public AddressDTO getAddress() {
40         return address;
41     }
42
43     /**
44      * Get the value of name
45      *
46      * @return the value of name
47      */
48     public String getName() {
49         return name;
50     }
51
52 }
```

Listing C.79 Java code implementing the `CustomerDTO` class in figure 5.31

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
```

```
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5  import se.kth.ict.oodbook.design.casestudy.model.CustomerDTO;
6  import se.kth.ict.oodbook.design.casestudy.model.Rental;
7
8  /**
9   * This is the application's only controller class. All calls to
10  * the model pass through here.
11  */
12 public class Controller {
13     /**
14      * Registers a new customer. Only registered customers can
15      * rent cars.
16      *
17      * @param customer The customer that will be registered.
18      */
19     public void registerCustomer(CustomerDTO customer) {
20         rental = new Rental(customer);
21     }
22 }
```

Listing C.80 Java code implementing the `Controller` class in figure 5.31

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents one particular rental transaction, where one
5   * particular car is rented by one particular customer.
6   */
7  public class Rental {
8      private CustomerDTO customer;
9
10     /**
11      * Creates a new instance, representing a rental made by the
12      * specified customer.
13      *
14      * @param customer The renting customer.
15      */
16     public Rental(CustomerDTO customer) {
17         this.customer = customer;
18     }
19 }
```

Listing C.81 Java code implementing the `Rental` class in figure 5.31

C.21 Figure 5.32

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
11     private Controller contr;
12
13     /**
14      * Creates a new instance.
15      *
16      * @param contr The controller that is used for all operations.
17      */
18     public View(Controller contr) {
19     }
20 }
```

Listing C.82 Java code implementing the `View` class in figure 5.32

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5
6  /**
7   * This is the application's only controller class. All calls to
8   * the model pass through here.
9   */
10 public class Controller {
11     private CarRegistry carRegistry;
12
13     /**
14      * Creates a new instance.
15      *
16      * @param carRegistry Used to access the car data store.
17      */
18     public Controller(CarRegistry carRegistry) {
19     }
20
21     /**
22      * Search for a car matching the specified search criteria.
23      *
24      * @param searchedCar This object contains the search criteria.
25      *                    Fields in the object that are set to
26      *                    <code>null</code> or
27      *                    <code>false</code> are ignored.
28      * @return The best match of the search criteria.
29      */
30     public CarDTO searchMatchingCar(CarDTO searchedCar) {
31     }
32
33     /**
34      * Registers a new customer. Only registered customers can
35      * rent cars.
36      *
37      * @param customer The customer that will be registered.
38      */
39     public void registerCustomer(CustomerDTO customer) {
40     }
41 }
```

Listing C.83 Java code implementing the `Controller` class in figure 5.32

```
1  package se.kth.ict.oodbook.design.casestudy.startup;
```

```
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5  import se.kth.ict.oodbook.design.casestudy.view.View;
6
7  /**
8   * Contains the <code>main</code> method. Performs all startup
9   * of the application.
10  */
11 public class Main {
12     public static void main(String[] args) {
13     }
14 }
```

Listing C.84 Java code implementing the `Main` class in figure 5.32

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  /**
4   * Contains all calls to the data store with cars that may be
5   * rented.
6   */
7  public class CarRegistry {
8      /**
9       * Creates a new instance.
10      */
11     public CarRegistry() {
12     }
13
14     /**
15      * Search for a car matching the specified search criteria.
16      *
17      * @param searchedCar This object contains the search
18      *                    criteria. Fields in the object that
19      *                    are set to <code>null</code> or
20      *                    <code>false</code> are ignored.
21      * @return The best match of the search criteria.
22      */
23     public CarDTO findCar(CarDTO searchedCar) {
24     }
25 }
```

Listing C.85 Java code implementing the `CarRegistry` class in figure 5.32

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
```

```
2
3  /**
4   * Contains information about one particular car.
5   */
6  public class CarDTO {
7
8      private int price;
9      private String size;
10     private boolean AC;
11     private boolean fourWD;
12     private String color;
13     private String regNo;
14
15     /**
16      * Creates a new instance representing a particular car.
17      *
18      * @param price The price paid to rent the car.
19      * @param size The size of the car, e.g.,
20      *             <code>medium hatchback</code>.
21      * @param AC      <code>true</code> if the car has
22      *                air condition.
23      * @param fourWD  <code>true</code> if the car has four
24      *                wheel drive.
25      * @param color The color of the car.
26      * @param regNo The car's registration number.
27      */
28     public CarDTO(int price, String size, boolean AC,
29                   boolean fourWD, String color, String regNo) {
30     }
31
32     /**
33      * Get the value of regNo
34      *
35      * @return the value of regNo
36      */
37     public String getRegNo() {
38     }
39
40     /**
41      * Get the value of color
42      *
43      * @return the value of color
44      */
45     public String getColor() {
46     }
47
```

```
48      /**
49       * Get the value of fourWD
50       *
51       * @return the value of fourWD
52       */
53      public boolean isFourWD() {
54      }
55
56      /**
57       * Get the value of AC
58       *
59       * @return the value of AC
60       */
61      public boolean isAC() {
62      }
63
64      /**
65       * Get the value of size
66       *
67       * @return the value of size
68       */
69      public String getSize() {
70      }
71
72      /**
73       * Get the value of price
74       *
75       * @return the value of price
76       */
77      public int getPrice() {
78      }
79
80  }
```

Listing C.86 Java code implementing the `CarDTO` class in figure 5.32

```
1   package se.kth.ict.oodbook.design.casestudy.model;
2
3   /**
4    * Represents one particular rental transaction, where one
5    * particular car is rented by one particular customer.
6    */
7   public class Rental {
8       private CustomerDTO customer;
9
10      /**
```

```
11      * Creates a new instance, representing a rental made by
12      * the specified customer.
13      *
14      * @param customer The renting customer.
15      */
16     public Rental(CustomerDTO customer) {
17     }
18 }
```

Listing C.87 Java code implementing the `Rental` class in figure 5.32

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents a driving license
5   */
6  public class DrivingLicenseDTO {
7      private final String licenseNo;
8
9      /**
10      * Creates a new instance.
11      *
12      * @param licenseNo The driving license number.
13      */
14     public DrivingLicenseDTO(String licenseNo) {
15     }
16
17     /**
18      * Get the value of licenseNo
19      *
20      * @return the value of licenseNo
21      */
22     public String getLicenseNo() {
23     }
24
25 }
```

Listing C.88 Java code implementing the `DrivingLicenseDTO` class in figure 5.32

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents a customer of the car rental company.
5   */
6  public class CustomerDTO {
```

```
7       private final String name;
8       private final AddressDTO address;
9       private final DrivingLicenseDTO drivingLicense;
10
11      /**
12       * Creates a new instance.
13       *
14       * @param name           The customer's name.
15       * @param address        The customer's address.
16       * @param drivingLicense The customer's driving license.
17       */
18      public CustomerDTO(String name, AddressDTO address,
19                         DrivingLicenseDTO drivingLicense) {
20      }
21
22      /**
23       * Get the value of drivingLicense
24       *
25       * @return the value of drivingLicense
26       */
27      public DrivingLicenseDTO getDrivingLicense() {
28      }
29
30      /**
31       * Get the value of address
32       *
33       * @return the value of address
34       */
35      public AddressDTO getAddress() {
36      }
37
38      /**
39       * Get the value of name
40       *
41       * @return the value of name
42       */
43      public String getName() {
44      }
45
46  }
```

Listing C.89 Java code implementing the `CustomerDTO` class in figure 5.32

```
1   package se.kth.ict.oodbook.design.casestudy.model;
2
3   /**
```

```
4   * Represents a post address.
5   */
6  public final class AddressDTO {
7      private final String street;
8      private final String zip;
9      private final String city;
10
11     /**
12      * Creates a new instance.
13      *
14      * @param street Street name and number.
15      * @param zip    Zip code
16      * @param city   City (postort)
17      */
18     public AddressDTO(String street, String zip, String city) {
19     }
20
21     /**
22      * Get the value of city
23      *
24      * @return the value of city
25      */
26     public String getCity() {
27     }
28
29     /**
30      * Get the value of zip
31      *
32      * @return the value of zip
33      */
34     public String getZip() {
35     }
36
37     /**
38      * Get the value of street
39      *
40      * @return the value of street
41      */
42     public String getStreet() {
43     }
44
45 }
```

Listing C.90 Java code implementing the `AddressDTO` class in figure 5.32

C.22 Figure 5.33

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
5  import se.kth.ict.oodbook.design.casestudy.model.AddressDTO;
6  import se.kth.ict.oodbook.design.casestudy.model.CustomerDTO;
7  import se.kth.ict.oodbook.design.casestudy.model.DrivingLicenseDTO;
8
9  /**
10  * This program has no view, instead, this class is a placeholder
11  * for the entire view.
12  */
13 public class View {
14     private Controller contr;
15
16     //Somewhere in the code.
17     contr.bookCar(foundCar);
18 }
```

Listing C.91 Java code implementing the `View` class in figure 5.33

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5  import se.kth.ict.oodbook.design.casestudy.dbhandler.RentalRegistry;
6  import se.kth.ict.oodbook.design.casestudy.model.CustomerDTO;
7  import se.kth.ict.oodbook.design.casestudy.model.Rental;
8
9  /**
10  * This is the application's only controller class. All calls to the
11  * model pass through here.
12  */
13 public class Controller {
14     private RentalRegistry rentalRegistry;
15     private Rental rental;
16
17     /**
18      * Books the specified car. After calling this method, the car
19      * can not be booked by any other customer. This method also
20      * permanently saves information about the current rental.
21      *
22      * @param car The car that will be booked.
```

```
23          */
24         public void bookCar(CarDTO car) {
25             rental.setRentedCar(car);
26             rentalRegistry.saveRental(rental);
27         }
28     }
```

Listing C.92 Java code implementing the `Controller` class in figure 5.33

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5
6  /**
7   * Represents one particular rental transaction, where one
8   * particular car is rented by one particular customer.
9   */
10 public class Rental {
11     private CarDTO rentedCar;
12     private CarRegistry carRegistry;
13     /**
14      * Specifies the car that was rented.
15      *
16      * @param rentedCar The car that was rented.
17      */
18     public void setRentedCar(CarDTO rentedCar) {
19         this.rentedCar = rentedCar;
20         carRegistry.bookCar(rentedCar);
21     }
22 }
```

Listing C.93 Java code implementing the `Rental` class in figure 5.33

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Rental;
4
5  /**
6   * Contains all calls to the data store with performed rentals.
7   */
8  public class RentalRegistry {
9      /**
10      * Saves the specified rental permanently.
11      *
```

```
12      * @param rental The rental that will be saved.
13      */
14     public void saveRental(Rental rental) {
15     }
16 }
```

Listing C.94 Java code implementing the `RentalRegistry` class in figure 5.33

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  /**
4   * Contains all calls to the data store with cars that may
5   * be rented.
6   */
7  public class CarRegistry {
8      /**
9       * Books the specified car. After calling this method,
10      * the car can not be booked by any other customer.
11      *
12      * @param car The car that will be booked.
13      */
14     public void bookCar(CarDTO car) {
15     }
16 }
```

Listing C.95 Java code implementing the `CarRegistry` class in figure 5.33

C.23 Figure 5.34

```
1  package se.kth.ict.oodbook.design.casestudy.startup;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5  import se.kth.ict.oodbook.design.casestudy.view.View;
6
7  /**
8   * Contains the <code>main</code> method. Performs all startup of
9   * the application.
10  */
11 public class Main {
12     public static void main(String[] args) {
13         CarRegistry carRegistry = new CarRegistry();
14         RentalRegistry rentalRegistry = new RentalRegistry();
```

```
15          Controller contr = new Controller(carRegistry,
16                                            rentalRegistry);
17          new View(contr);
18      }
19  }
```

Listing C.96 Java code implementing the `Main` class in figure 5.34

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  /**
4   * Contains all calls to the data store with cars that may be
5   * rented.
6   */
7  public class CarRegistry {
8  }
```

Listing C.97 Java code implementing the `CarRegistry` class in figure 5.34

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Rental;
4
5  /**
6   * Contains all calls to the data store with performed rentals.
7   */
8  public class RentalRegistry {
9  }
```

Listing C.98 Java code implementing the `RentalRegistry` class in figure 5.34

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5
6  /**
7   * This is the application's only controller class. All calls to
8   * the model pass through here.
9   */
10 public class Controller {
11     /**
12      * Creates a new instance.
13      *
```

```
14      * @param carRegistry Used to access the car data store.
15      * @param rentalRegistry Used to access the rental data store.
16      */
17     public Controller(CarRegistry carRegistry,
18                       RentalRegistry rentalRegistry) {
19     }
20 }
```

Listing C.99 Java code implementing the `Controller` class in figure 5.34

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
11     /**
12      * Creates a new instance.
13      *
14      * @param contr The controller that is used for all
15      *              operations.
16      */
17     public View(Controller contr) {
18     }
19 }
```

Listing C.100 Java code implementing the `View` class in figure 5.34

C.24 Figure 5.35

```
1  package se.kth.ict.oodbook.design.casestudy.startup;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5  import se.kth.ict.oodbook.design.casestudy.view.View;
6
7  /**
8   * Contains the <code>main</code> method. Performs all startup of
9   * the application.
```

```
 */
public class Main {
    public static void main(String[] args) {
        RegistryCreator creator = new RegistryCreator();
        Controller contr = new Controller(creator);
        new View(contr);
    }
}
```

Listing C.101 Java code implementing the `Main` class in figure 5.35

```
package se.kth.ict.oodbook.design.casestudy.dbhandler;

/**
 * Contains all calls to the data store with cars that may be
 * rented.
 */
public class CarRegistry {
}
```

Listing C.102 Java code implementing the `CarRegistry` class in figure 5.35

```
package se.kth.ict.oodbook.design.casestudy.dbhandler;

import se.kth.ict.oodbook.design.casestudy.model.Rental;

/**
 * Contains all calls to the data store with performed rentals.
 */
public class RentalRegistry {
}
```

Listing C.103 Java code implementing the `RentalRegistry` class in figure 5.35

```
package se.kth.ict.oodbook.design.casestudy.dbhandler;

/**
 * This class is responsible for instantiating all registries.
 */
public class RegistryCreator {
    private CarRegistry carRegistry = new CarRegistry();
    private RentalRegistry rentalRegistry = new RentalRegistry();

    /**
```

```
11      * Get the value of rentalRegistry
12      *
13      * @return the value of rentalRegistry
14      */
15     public RentalRegistry getRentalRegistry() {
16         return rentalRegistry;
17     }
18
19     /**
20      * Get the value of carRegistry
21      *
22      * @return the value of carRegistry
23      */
24     public CarRegistry getCarRegistry() {
25         return carRegistry;
26     }
27 }
```

Listing C.104 Java code implementing the `RegistryCreator` class in figure 5.35

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5
6  /**
7   * This is the application's only controller class. All calls to
8   * the model pass through here.
9   */
10 public class Controller {
11     private CarRegistry carRegistry;
12     private RentalRegistry rentalRegistry;
13
14     /**
15      * Creates a new instance.
16      *
17      * @param regCreator Used to get all classes that handle
18      *                   database calls.
19      * @param printer    Interface to printer.
20      */
21     public Controller(RegistryCreator regCreator) {
22         this.carRegistry = regCreator.getCarRegistry();
23         this.rentalRegistry = regCreator.getRentalRegistry();
24     }
25 }
```

Listing C.105 Java code implementing the `Controller` class in figure 5.35

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
11     /**
12      * Creates a new instance.
13      *
14      * @param contr The controller that is used for all
15      *              operations.
16      */
17     public View(Controller contr) {
18     }
19 }
```

Listing C.106 Java code implementing the `View` class in figure 5.35

C.25 Figure 5.36

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4
5  /**
6   * This program has no view, instead, this class is a placeholder
7   * for the entire view.
8   */
9  public class View {
10     private Controller contr;
11
12     /**
13      * Creates a new instance.
14      *
15      * @param contr The controller that is used for all
```

```
16          *              operations.
17          */
18         public View(Controller contr) {
19         }
20 }
```

Listing C.107 Java code implementing the `View` class in figure 5.36

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5  import se.kth.ict.oodbook.design.casestudy.dbhandler.RegistryCreator;
6  import se.kth.ict.oodbook.design.casestudy.dbhandler.RentalRegistry;
7  import se.kth.ict.oodbook.design.casestudy.model.CustomerDTO;
8  import se.kth.ict.oodbook.design.casestudy.model.Rental;
9
10 /**
11  * This is the application's only controller class. All calls to
12  * the model pass through here.
13  */
14 public class Controller {
15     private CarRegistry carRegistry;
16     private RentalRegistry rentalRegistry;
17     private Rental rental;
18
19     /**
20      * Creates a new instance.
21      *
22      * @param regCreator Used to get all classes that handle
23      *                   database calls.
24      */
25     public Controller(RegistryCreator regCreator) {
26     }
27
28     /**
29      * Search for a car matching the specified search criteria.
30      *
31      * @param searchedCar This object contains the search criteria.
32      *                    Fields in the object that are set to
33      *                    <code>null</code> or
34      *                    <code>false</code> are ignored.
35      * @return The best match of the search criteria.
36      */
37     public CarDTO searchMatchingCar(CarDTO searchedCar) {
38     }
```

```
39
40      /**
41       * Registers a new customer. Only registered customers can
42       * rent cars.
43       *
44       * @param customer The customer that will be registered.
45       */
46      public void registerCustomer(CustomerDTO customer) {
47      }
48
49      /**
50       * Books the specified car. After calling this method, the car
51       * can not be booked by any other customer. This method also
52       * permanently saves information about the current rental.
53       *
54       * @param car The car that will be booked.
55       */
56      public void bookCar(CarDTO car) {
57      }
58  }
```

Listing C.108 Java code implementing the `Controller` class in figure 5.36

```
1   package se.kth.ict.oodbook.design.casestudy.startup;
2
3   import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4   import se.kth.ict.oodbook.design.casestudy.dbhandler.RegistryCreator;
5   import se.kth.ict.oodbook.design.casestudy.view.View;
6
7   /**
8    * Contains the <code>main</code> method. Performs all startup of
9    * the application.
10   */
11  public class Main {
12      /**
13       * Starts the application.
14       *
15       * @param args The application does not take any command line
16       *             parameters.
17       */
18      public static void main(String[] args) {
19      }
20  }
```

Listing C.109 Java code implementing the `Main` class in figure 5.36

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5
6  /**
7   * Represents one particular rental transaction, where one
8   * particular car is rented by one particular customer.
9   */
10 public class Rental {
11     private CarRegistry carRegistry;
12
13     /**
14      * Creates a new instance, representing a rental made by the
15      * specified customer.
16      *
17      * @param customer The renting customer.
18      * @param carRegistry The data store with information about
19      *                    available cars.
20      */
21     public Rental(CustomerDTO customer, CarRegistry carRegistry) {
22     }
23
24     /**
25      * Specifies the car that was rented.
26      *
27      * @param rentedCar The car that was rented.
28      */
29     public void setRentedCar(CarDTO rentedCar) {
30     }
31 }
```

Listing C.110 Java code implementing the `Rental` class in figure 5.36

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  /**
4   * This class is responsible for instantiating all registries.
5   */
6  public class RegistryCreator {
7      /**
8       * Get the value of rentalRegistry
9       *
10      * @return the value of rentalRegistry
11      */
```

```
12      public RentalRegistry getRentalRegistry() {
13      }
14
15      /**
16       * Get the value of carRegistry
17       *
18       * @return the value of carRegistry
19       */
20      public CarRegistry getCarRegistry() {
21      }
22  }
```

Listing C.111 Java code implementing the `RegistryCreator` class in figure 5.36

```
1   package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3   /**
4    * Contains all calls to the data store with cars that may be
5    * rented.
6    */
7   public class CarRegistry {
8       CarRegistry() {
9       }
10
11      /**
12       * Search for a car matching the specified search criteria.
13       *
14       * @param searchedCar This object contains the search criteria.
15       *                    Fields in the object that are set to
16       *                    <code>null</code> or <code>false</code>
17       *                    are ignored.
18       * @return The best match of the search criteria.
19       */
20      public CarDTO findCar(CarDTO searchedCar) {
21      }
22
23      /**
24       * Books the specified car. After calling this method, the car
25       * can not be booked by any other customer.
26       *
27       * @param car The car that will be booked.
28       */
29      public void bookCar(CarDTO car) {
30      }
31  }
```

Listing C.112 Java code implementing the `CarRegistry` class in figure 5.36

```
1  package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Rental;
4
5  /**
6   * Contains all calls to the data store with performed rentals.
7   */
8  public class RentalRegistry {
9      RentalRegistry() {
10     }
11
12     /**
13      * Saves the specified rental permanently.
14      *
15      * @param rental The rental that will be saved.
16      */
17     public void saveRental(Rental rental) {
18     }
19 }
```

Listing C.113 Java code implementing the `RentalRegistry` class in figure 5.36

C.26 Figure 5.37

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.model.Amount;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
11     private Controller contr;
12
13     // Somewhere in the code. The used amount (100) is not
14     // specified in the diagram.
15     Amount paidAmount = new Amount(100);
```

```
16      contr.pay(paidAmount);
17  }
```

Listing C.114 Java code implementing the `View` class in figure 5.37

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Amount;
4  import se.kth.ict.oodbook.design.casestudy.model.Rental;
5
6  /**
7   * This is the application's only controller class. All
8   * calls to the model pass through here.
9   */
10 public class Controller {
11     /**
12      * Handles rental payment. Updates the balance of
13      * the cash register where the payment was
14      * performed. Calculates change. Prints the receipt.
15      *
16      * @param amount The paid amount.
17      */
18     public void pay(Amount paidAmt) {
19     }
20 }
```

Listing C.115 Java code implementing the `Controller` class in figure 5.37

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents an amount of money
5   */
6  public final class Amount {
7      private final int amount;
8
9      public Amount(int amount) {
10         this.amount = amount;
11     }
12 }
```

Listing C.116 Java code implementing the `Amount` class in figure 5.37

C.27 Figure 5.38

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.model.Amount;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
11     private Controller contr;
12
13     // Somewhere in the code. The used amount (100) is not
14     // specified in the diagram.
15     Amount paidAmount = new Amount(100);
16     contr.pay(paidAmount);
17 }
```

Listing C.117 Java code implementing the `View` class in figure 5.38

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Amount;
4  import se.kth.ict.oodbook.design.casestudy.model.Rental;
5
6  /**
7   * This is the application's only controller class. All
8   * calls to the model pass through here.
9   */
10 public class Controller {
11     /**
12      * Handles rental payment. Updates the balance of
13      * the cash register where the payment was
14      * performed. Calculates change. Prints the receipt.
15      *
16      * @param amount The paid amount.
17      */
18     public void pay(Amount paidAmt) {
19         CashPayment payment = new CashPayment(paidAmt);
20         rental.pay(payment);
21         cashRegister.addPayment(payment);
22     }
23 }
```

Listing C.118 Java code implementing the `Controller` class in figure 5.38

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents a cash register. There shall be one instance of
5   * this class for each register.
6   */
7  public class CashRegister {
8      public void addPayment(CashPayment payment) {
9      }
10 }
```

Listing C.119 Java code implementing the `CashRegister` class in figure 5.38

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents one specific payment for one specific rental. The
5   * rental is payed with cash.
6   */
7  public class CashPayment {
8      private Amount paidAmt;
9
10     /**
11      * Creates a new instance. The customer handed over the
12      * specified amount.
13      *
14      * @param paidAmt The amount of cash that was handed over
15      *                by the customer.
16      */
17     public CashPayment(Amount paidAmt) {
18         this.paidAmt = paidAmt;
19     }
20
21     /**
22      * Calculates the total cost of the specified rental.
23      *
24      * @param paidRental The rental for which the customer is
25      *                   paying.
26      */
27     void calculateTotalCost(Rental paidRental) {
28     }
```

```
29 }
```

Listing C.120 Java code implementing the `CashPayment` class in figure 5.38

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5
6  /**
7   * Represents one particular rental transaction, where one
8   * particular car is rented by one particular customer.
9   */
10 public class Rental {
11     /**
12      * This rental is paid using the specified payment.
13      *
14      * @param payment The payment used to pay this rental.
15      */
16     public void pay(CashPayment payment) {
17         payment.calculateTotalCost(this);
18     }
19 }
```

Listing C.121 Java code implementing the `Rental` class in figure 5.38

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents an amount of money
5   */
6  public final class Amount {
7      private final int amount;
8
9      public Amount(int amount) {
10         this.amount = amount;
11     }
12 }
```

Listing C.122 Java code implementing the `Amount` class in figure 5.38

C.28 Figure 5.39

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.model.Amount;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
11     private Controller contr;
12
13     // Somewhere in the code. The used amount (100) is not
14     // specified in the diagram.
15     Amount paidAmount = new Amount(100);
16     contr.pay(paidAmount);
17 }
```

Listing C.123 Java code implementing the `View` class in figure 5.39

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Amount;
4  import se.kth.ict.oodbook.design.casestudy.integration.Printer;
5  import se.kth.ict.oodbook.design.casestudy.model.CashPayment;
6  import se.kth.ict.oodbook.design.casestudy.model.CashRegister;
7  import se.kth.ict.oodbook.design.casestudy.model.Rental;
8  import se.kth.ict.oodbook.design.casestudy.model.Receipt;
9
10 /**
11  * This is the application's only controller class. All
12  * calls to the model pass through here.
13  */
14 public class Controller {
15     private Rental rental;
16     private CashRegister cashRegister;
17     private Printer printer;
18
19         /**
20      * Handles rental payment. Updates the balance of
21      * the cash register where the payment was
22      * performed. Calculates change. Prints the receipt.
23      *
24      * @param amount The paid amount.
25      */
```

```
26      public void pay(Amount amount) {
27          CashPayment payment = new CashPayment(paidAmt);
28          rental.pay(payment);
29          cashRegister.addPayment(payment);
30          Receipt receipt = rental.getReceipt();
31          printer.printReceipt(receipt);
32      }
33  }
```

Listing C.124 Java code implementing the `Controller` class in figure 5.39

```
1   package se.kth.ict.oodbook.design.casestudy.model;
2
3   /**
4    * Represents a cash register. There shall be one instance of
5    * this class for each register.
6    */
7   public class CashRegister {
8       public void addPayment(CashPayment payment) {
9       }
10  }
```

Listing C.125 Java code implementing the `CashRegister` class in figure 5.39

```
1   package se.kth.ict.oodbook.design.casestudy.model;
2
3   /**
4    * Represents one specific payment for one specific rental. The
5    * rental is payed with cash.
6    */
7   public class CashPayment {
8       private Amount paidAmt;
9
10      /**
11       * Creates a new instance. The customer handed over the
12       * specified amount.
13       *
14       * @param paidAmt The amount of cash that was handed over
15       *                by the customer.
16       */
17      public CashPayment(Amount paidAmt) {
18          this.paidAmt = paidAmt;
19      }
20
21      /**
```

```
22      * Calculates the total cost of the specified rental.
23      *
24      * @param paidRental The rental for which the customer is
25      *                   paying.
26      */
27     void calculateTotalCost(Rental paidRental) {
28     }
29 }
```

Listing C.126 Java code implementing the `CashPayment` class in figure 5.39

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents one particular rental transaction, where one
5   * particular car is rented by one particular customer.
6   */
7  public class Rental {
8      /**
9       * This rental is paid using the specified payment.
10      *
11      * @param payment The payment used to pay this rental.
12      */
13     public void pay(CashPayment payment) {
14         payment.calculateTotalCost(this);
15     }
16
17     /**
18      * Returns a receipt for the current rental.
19      */
20     public Receipt getReceipt() {
21         return new Receipt(this);
22     }
23 }
```

Listing C.127 Java code implementing the `Rental` class in figure 5.39

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * The receipt of a rental
5   */
6  public class Receipt {
7
8      /**
```

```
9       * Creates a new instance.
10      *
11      * @param rental The rental proved by this receipt.
12      */
13     Receipt(Rental rental) {
14     }
15
16 }
```

Listing C.128 Java code implementing the `Receipt` class in figure 5.39

```
1  package se.kth.ict.oodbook.design.casestudy.integration;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Receipt;
4
5  /**
6   * The interface to the printer, used for all printouts initiated
7   * by this program.
8   */
9  public class Printer {
10     public void printReceipt(Receipt receipt) {
11
12     }
13 }
```

Listing C.129 Java code implementing the `Printer` class in figure 5.39

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents an amount of money
5   */
6  public final class Amount {
7      private final int amount;
8
9      public Amount(int amount) {
10         this.amount = amount;
11     }
12 }
```

Listing C.130 Java code implementing the `Amount` class in figure 5.39

C.29 Figure 5.40

```
1  package se.kth.ict.oodbook.design.casestudy.startup;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.integration.Printer;
5  import se.kth.ict.oodbook.design.casestudy.integration.RegistryCreator;
6  import se.kth.ict.oodbook.design.casestudy.view.View;
7
8  /**
9   * Contains the <code>main</code> method. Performs all startup of the
10  * application.
11  */
12 public class Main {
13     /**
14      * Starts the application.
15      *
16      * @param args The application does not take any command line
17      *             parameters.
18      */
19     public static void main(String[] args) {
20         RegistryCreator creator = new RegistryCreator();
21         Printer printer = new Printer();
22         Controller contr = new Controller(creator, printer);
23         new View(contr).sampleExecution();
24     }
25 }
```

Listing C.131 Java code implementing the `Main` class in figure 5.40

```
1  package se.kth.ict.oodbook.design.casestudy.integration;
2
3  /**
4   * Contains all calls to the data store with cars that may be
5   * rented.
6   */
7  public class CarRegistry {
8  }
```

Listing C.132 Java code implementing the `CarRegistry` class in figure 5.40

```
1  package se.kth.ict.oodbook.design.casestudy.integration;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Rental;
```

```
4
5  /**
6   * Contains all calls to the data store with performed rentals.
7   */
8  public class RentalRegistry {
9  }
```

Listing C.133 Java code implementing the `RentalRegistry` class in figure 5.40

```
1  package se.kth.ict.oodbook.design.casestudy.integration;
2
3  /**
4   * This class is responsible for instantiating all registries.
5   */
6  public class RegistryCreator {
7      private CarRegistry carRegistry = new CarRegistry();
8      private RentalRegistry rentalRegistry = new RentalRegistry();
9
10     /**
11      * Get the value of rentalRegistry
12      *
13      * @return the value of rentalRegistry
14      */
15     public RentalRegistry getRentalRegistry() {
16         return rentalRegistry;
17     }
18
19     /**
20      * Get the value of carRegistry
21      *
22      * @return the value of carRegistry
23      */
24     public CarRegistry getCarRegistry() {
25         return carRegistry;
26     }
27 }
```

Listing C.134 Java code implementing the `RegistryCreator` class in figure 5.40

```
1  package se.kth.ict.oodbook.design.casestudy.integration;
2
3  /**
4   * The interface to the printer, used for all printouts
5   * initiated by this program.
6   */
```

```
7 public class Printer {
8 }
```

Listing C.135 Java code implementing the `Printer` class in figure 5.40

```
1 package se.kth.ict.oodbook.design.casestudy.model;
2
3 /**
4  * Represents a cash register. There shall be one
5  * instance of this class for each register.
6  */
7 public class CashRegister {
8 }
```

Listing C.136 Java code implementing the `CashRegister` class in figure 5.40

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.integration.CarRegistry;
4  import se.kth.ict.oodbook.design.casestudy.integration.Printer;
5  import se.kth.ict.oodbook.design.casestudy.integration.RegistryCreator;
6  import se.kth.ict.oodbook.design.casestudy.integration.RentalRegistry;
7  import se.kth.ict.oodbook.design.casestudy.model.CashRegister;
8
9  /**
10  * This is the application's only controller class. All calls to the
11  * model pass through here.
12  */
13 public class Controller {
14     private CarRegistry carRegistry;
15     private RentalRegistry rentalRegistry;
16     private CashRegister cashRegister;
17     private Printer printer;
18
19     /**
20      * Creates a new instance.
21      *
22      * @param regCreator Used to get all classes that handle database
23      *                   calls.
24      * @param printer    Interface to printer.
25      */
26     public Controller(RegistryCreator regCreator, Printer printer) {
27         this.carRegistry = regCreator.getCarRegistry();
28         this.rentalRegistry = regCreator.getRentalRegistry();
29         this.printer = printer;
```

```
30          this.cashRegister = new CashRegister();
31      }
32  }
```

Listing C.137 Java code implementing the `Controller` class in figure 5.40

```
1   package se.kth.ict.oodbook.design.casestudy.view;
2
3   import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4
5   /**
6    * This program has no view, instead, this class is a placeholder
7    * for the entire view.
8    */
9   public class View {
10      private Controller contr;
11
12      /**
13       * Creates a new instance.
14       *
15       * @param contr The controller that is used for all operations.
16       */
17      public View(Controller contr) {
18          this.contr = contr;
19      }
20  }
```

Listing C.138 Java code implementing the `View` class in figure 5.40

C.30 Figure 5.41

```
1   package se.kth.ict.oodbook.design.casestudy.view;
2
3   import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4
5   /**
6    * This program has no view, instead, this class is a placeholder
7    * for the entire view.
8    */
9   public class View {
10      private Controller contr;
11
12      /**
```

```
13      * Creates a new instance.
14      *
15      * @param contr The controller that is used for all
16      *              operations.
17      */
18     public View(Controller contr) {
19     }
20 }
```

Listing C.139 Java code implementing the `View` class in figure 5.41

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Amount;
4  import se.kth.ict.oodbook.design.casestudy.integration.CarDTO;
5  import se.kth.ict.oodbook.design.casestudy.integration.CarRegistry;
6  import se.kth.ict.oodbook.design.casestudy.integration.Printer;
7  import se.kth.ict.oodbook.design.casestudy.integration.RegistryCreator;
8  import se.kth.ict.oodbook.design.casestudy.integration.RentalRegistry;
9  import se.kth.ict.oodbook.design.casestudy.model.CashPayment;
10 import se.kth.ict.oodbook.design.casestudy.model.CashRegister;
11 import se.kth.ict.oodbook.design.casestudy.model.CustomerDTO;
12 import se.kth.ict.oodbook.design.casestudy.model.Rental;
13 import se.kth.ict.oodbook.design.casestudy.model.Receipt;
14
15 /**
16  * This is the application's only controller class. All calls to
17  * the model pass through here.
18  */
19 public class Controller {
20     private CarRegistry carRegistry;
21     private RentalRegistry rentalRegistry;
22     private Rental rental;
23
24     /**
25      * Creates a new instance.
26      *
27      * @param regCreator Used to get all classes that handle
28      *                   database calls.
29      */
30     public Controller(RegistryCreator regCreator) {
31     }
32
33     /**
34      * Search for a car matching the specified search criteria.
35      *
```

```
36          * @param searchedCar This object contains the search criteria.
37          *                    Fields in the object that are set to
38          *                    <code>null</code> or
39          *                    <code>false</code> are ignored.
40          * @return The best match of the search criteria.
41          */
42         public CarDTO searchMatchingCar(CarDTO searchedCar) {
43         }
44
45         /**
46          * Registers a new customer. Only registered customers can
47          * rent cars.
48          *
49          * @param customer The customer that will be registered.
50          */
51         public void registerCustomer(CustomerDTO customer) {
52         }
53
54         /**
55          * Books the specified car. After calling this method, the car
56          * can not be booked by any other customer. This method also
57          * permanently saves information about the current rental.
58          *
59          * @param car The car that will be booked.
60          */
61         public void bookCar(CarDTO car) {
62         }
63
64         /**
65          * Handles rental payment. Updates the balance of the cash register
66          * where the payment was performed. Calculates change. Prints the
67          * receipt.
68          *
69          * @param paidAmt The paid amount.
70          */
71         public void pay(Amount paidAmt) {
72         }
73 }
```

Listing C.140 Java code implementing the `Controller` class in figure 5.41

```
1  package se.kth.ict.oodbook.design.casestudy.startup;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.integration.Printer;
5  import se.kth.ict.oodbook.design.casestudy.integration.RegistryCreator;
```

```
6  import se.kth.ict.oodbook.design.casestudy.view.View;
7
8  /**
9   * Contains the <code>main</code> method. Performs all startup of
10  * the application.
11  */
12 public class Main {
13     /**
14      * Starts the application.
15      *
16      * @param args The application does not take any command line
17      *             parameters.
18      */
19     public static void main(String[] args) {
20     }
21 }
```

Listing C.141 Java code implementing the `Main` class in figure 5.41

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  import se.kth.ict.oodbook.design.casestudy.integration.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.integration.CarRegistry;
5
6  /**
7   * Represents one particular rental transaction, where one
8   * particular car is rented by one particular customer.
9   */
10 public class Rental {
11     private CarRegistry carRegistry;
12
13     /**
14      * Creates a new instance, representing a rental made by the
15      * specified customer.
16      *
17      * @param customer The renting customer.
18      * @param carRegistry The data store with information about
19      *                    available cars.
20      */
21     public Rental(CustomerDTO customer, CarRegistry carRegistry) {
22     }
23
24     /**
25      * Specifies the car that was rented.
26      *
27      * @param rentedCar The car that was rented.
```

```
28          */
29         public void setRentedCar(CarDTO rentedCar) {
30         }
31
32         /**
33          * This rental is paid using the specified payment.
34          *
35          * @param payment The payment used to pay this rental.
36          */
37         public void pay(CashPayment payment) {
38         }
39
40         /**
41          * Returns a receipt for the current rental.
42          */
43         public Receipt getReceipt() {
44         }
45 }
```

Listing C.142 Java code implementing the `Rental` class in figure 5.41

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * The receipt of a rental
5   */
6  public class Receipt {
7      /**
8       * Creates a new instance.
9       *
10      * @param rental The rental proved by this receipt.
11      */
12     Receipt(Rental rental) {
13     }
14 }
```

Listing C.143 Java code implementing the `Receipt` class in figure 5.41

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents one specific payment for one specific rental.
5   * The rental is payed with cash.
6   */
7  public class CashPayment {
```

```
8       /**
9        * Creates a new instance. The customer handed over
10       * the specified amount.
11       *
12       * @param paidAmt The amount of cash that was handed
13       *                over by the customer.
14       */
15      public CashPayment(Amount paidAmt) {
16      }
17
18      /**
19       * Calculates the total cost of the specified rental.
20       *
21       * @param paidRental The rental for which the customer
22       *                   is paying.
23       */
24      void calculateTotalCost(Rental paidRental) {
25      }
26  }
```

Listing C.144 Java code implementing the `CashPayment` class in figure 5.41

```
1   package se.kth.ict.oodbook.design.casestudy.model;
2
3   /**
4    * Represents a cash register. There shall be one instance
5    * of this class for each register.
6    */
7   public class CashRegister {
8       public void addPayment(CashPayment payment) {
9       }
10  }
```

Listing C.145 Java code implementing the `CashRegister` class in figure 5.41

```
1   package se.kth.ict.oodbook.design.casestudy.integration;
2
3   import se.kth.ict.oodbook.design.casestudy.model.Receipt;
4
5   /**
6    * The interface to the printer, used for all printouts
7    * initiated by this program.
8    */
9   public class Printer {
10      public void printReceipt(Receipt receipt) {
```

```
11      }
12  }
```

Listing C.146 Java code implementing the `Printer` class in figure 5.41

```
1  package se.kth.ict.oodbook.design.casestudy.integration;
2
3  /**
4   * This class is responsible for instantiating all registries.
5   */
6  public class RegistryCreator {
7      /**
8       * Get the value of rentalRegistry
9       *
10      * @return the value of rentalRegistry
11      */
12     public RentalRegistry getRentalRegistry() {
13     }
14
15     /**
16      * Get the value of carRegistry
17      *
18      * @return the value of carRegistry
19      */
20     public CarRegistry getCarRegistry() {
21     }
22 }
```

Listing C.147 Java code implementing the `RegistryCreator` class in figure 5.41

```
1  package se.kth.ict.oodbook.design.casestudy.integration;
2
3  /**
4   * Contains all calls to the data store with cars that may be
5   * rented.
6   */
7  public class CarRegistry {
8      CarRegistry() {
9      }
10
11     /**
12      * Search for a car matching the specified search criteria.
13      *
14      * @param searchedCar This object contains the search criteria.
15      *                    Fields in the object that are set to
```

```
16          *                       <code>null</code> or <code>false</code>
17          *                       are ignored.
18          * @return The best match of the search criteria.
19          */
20         public CarDTO findCar(CarDTO searchedCar) {
21         }
22
23         /**
24          * Books the specified car. After calling this method, the car
25          * can not be booked by any other customer.
26          *
27          * @param car The car that will be booked.
28          */
29         public void bookCar(CarDTO car) {
30         }
31     }
```

Listing C.148 Java code implementing the `CarRegistry` class in figure 5.41

```
1  package se.kth.ict.oodbook.design.casestudy.integration;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Rental;
4
5  /**
6   * Contains all calls to the data store with performed rentals.
7   */
8  public class RentalRegistry {
9      RentalRegistry() {
10     }
11
12     /**
13      * Saves the specified rental permanently.
14      *
15      * @param rental The rental that will be saved.
16      */
17     public void saveRental(Rental rental) {
18     }
19 }
```

Listing C.149 Java code implementing the `RentalRegistry` class in figure 5.41

C.31 Figure 5.42

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.model.Amount;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
11     private Controller contr;
12
13     // Somewhere in the code. The used amount (100) is not
14     // specified in the diagram.
15     Amount paidAmount = new Amount(100);
16     contr.pay(paidAmount);
17 }
```

Listing C.150 Java code implementing the `View` class in figure 5.42

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Amount;
4  import se.kth.ict.oodbook.design.casestudy.integration.Printer;
5  import se.kth.ict.oodbook.design.casestudy.model.CashPayment;
6  import se.kth.ict.oodbook.design.casestudy.model.CashRegister;
7  import se.kth.ict.oodbook.design.casestudy.model.Rental;
8
9  /**
10  * This is the application's only controller class. All
11  * calls to the model pass through here.
12  */
13 public class Controller {
14     private Rental rental;
15     private CashRegister cashRegister;
16     private Printer printer;
17
18         /**
19      * Handles rental payment. Updates the balance of the cash
20      * register where the payment was performed. Calculates
21      * change. Prints the receipt.
22      *
23      * @param paidAmt The paid amount.
24      */
25     public void pay(Amount paidAmt) {
26         CashPayment payment = new CashPayment(paidAmt);
```

```
27          rental.pay(payment);
28          cashRegister.addPayment(payment);
29          rental.printReceipt(printer);
30      }
31  }
```

Listing C.151 Java code implementing the `Controller` class in figure 5.42

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents a cash register. There shall be one instance of
5   * this class for each register.
6   */
7  public class CashRegister {
8      public void addPayment(CashPayment payment) {
9      }
10 }
```

Listing C.152 Java code implementing the `CashRegister` class in figure 5.42

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents one specific payment for one specific rental. The
5   * rental is payed with cash.
6   */
7  public class CashPayment {
8      private Amount paidAmt;
9
10     /**
11      * Creates a new instance. The customer handed over the
12      * specified amount.
13      *
14      * @param paidAmt The amount of cash that was handed over
15      *                by the customer.
16      */
17     public CashPayment(Amount paidAmt) {
18         this.paidAmt = paidAmt;
19     }
20
21     /**
22      * Calculates the total cost of the specified rental.
23      *
24      * @param paidRental The rental for which the customer is
```

```
25          *                    paying.
26          */
27         void calculateTotalCost(Rental paidRental) {
28         }
29 }
```

Listing C.153 Java code implementing the `CashPayment` class in figure 5.42

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  import se.kth.ict.oodbook.design.casestudy.integration.Printer;
4
5  /**
6   * Represents one particular rental transaction, where one
7   * particular car is rented by one particular customer.
8   */
9  public class Rental {
10     /**
11      * This rental is paid using the specified payment.
12      *
13      * @param payment The payment used to pay this rental.
14      */
15     public void pay(CashPayment payment) {
16         payment.calculateTotalCost(this);
17     }
18
19     /**
20      * Prints a receipt for the current rental on the
21      * specified printer.
22      */
23     public void printReceipt(Printer printer) {
24         Receipt receipt = new Receipt(this);
25         printer.printReceipt(receipt);
26     }
27 }
```

Listing C.154 Java code implementing the `Rental` class in figure 5.42

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * The receipt of a rental
5   */
6  public class Receipt {
7
```

```
8       /**
9        * Creates a new instance.
10       *
11       * @param rental The rental proved by this receipt.
12       */
13      Receipt(Rental rental) {
14      }
15
16  }
```

Listing C.155 Java code implementing the `Receipt` class in figure 5.42

```
1   package se.kth.ict.oodbook.design.casestudy.integration;
2
3   import se.kth.ict.oodbook.design.casestudy.model.Receipt;
4
5   /**
6    * The interface to the printer, used for all printouts initiated
7    * by this program.
8    */
9   public class Printer {
10      public void printReceipt(Receipt receipt) {
11
12      }
13  }
```

Listing C.156 Java code implementing the `Printer` class in figure 5.42

```
1   package se.kth.ict.oodbook.design.casestudy.model;
2
3   /**
4    * Represents an amount of money
5    */
6   public final class Amount {
7       private final int amount;
8
9       public Amount(int amount) {
10          this.amount = amount;
11      }
12  }
```

Listing C.157 Java code implementing the `Amount` class in figure 5.42

C.32 Figure 5.43

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4
5  /**
6   * This program has no view, instead, this class is a placeholder
7   * for the entire view.
8   */
9  public class View {
10     private Controller contr;
11
12     /**
13      * Creates a new instance.
14      *
15      * @param contr The controller that is used for all
16      *              operations.
17      */
18     public View(Controller contr) {
19     }
20 }
```

Listing C.158 Java code implementing the `View` class in figure 5.43

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Amount;
4  import se.kth.ict.oodbook.design.casestudy.integration.CarDTO;
5  import se.kth.ict.oodbook.design.casestudy.integration.CarRegistry;
6  import se.kth.ict.oodbook.design.casestudy.integration.Printer;
7  import se.kth.ict.oodbook.design.casestudy.integration.RegistryCreator;
8  import se.kth.ict.oodbook.design.casestudy.integration.RentalRegistry;
9  import se.kth.ict.oodbook.design.casestudy.model.CashPayment;
10 import se.kth.ict.oodbook.design.casestudy.model.CashRegister;
11 import se.kth.ict.oodbook.design.casestudy.model.CustomerDTO;
12 import se.kth.ict.oodbook.design.casestudy.model.Rental;
13 import se.kth.ict.oodbook.design.casestudy.model.Receipt;
14
15 /**
16  * This is the application's only controller class. All calls to
17  * the model pass through here.
18  */
19 public class Controller {
20     private CarRegistry carRegistry;
```

```
21      private RentalRegistry rentalRegistry;
22      private Rental rental;
23
24      /**
25       * Creates a new instance.
26       *
27       * @param regCreator Used to get all classes that handle
28       *                   database calls.
29       */
30      public Controller(RegistryCreator regCreator) {
31      }
32
33      /**
34       * Search for a car matching the specified search criteria.
35       *
36       * @param searchedCar This object contains the search criteria.
37       *                    Fields in the object that are set to
38       *                    <code>null</code> or
39       *                    <code>false</code> are ignored.
40       * @return The best match of the search criteria.
41       */
42      public CarDTO searchMatchingCar(CarDTO searchedCar) {
43      }
44
45      /**
46       * Registers a new customer. Only registered customers can
47       * rent cars.
48       *
49       * @param customer The customer that will be registered.
50       */
51      public void registerCustomer(CustomerDTO customer) {
52      }
53
54      /**
55       * Books the specified car. After calling this method, the car
56       * can not be booked by any other customer. This method also
57       * permanently saves information about the current rental.
58       *
59       * @param car The car that will be booked.
60       */
61      public void bookCar(CarDTO car) {
62      }
63
64      /**
65       * Handles rental payment. Updates the balance of the cash register
66       * where the payment was performed. Calculates change. Prints the
```

```
67          * receipt.
68          *
69          * @param paidAmt The paid amount.
70          */
71         public void pay(Amount paidAmt) {
72         }
73 }
```

Listing C.159 Java code implementing the `Controller` class in figure 5.43

```
1  package se.kth.ict.oodbook.design.casestudy.startup;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.integration.Printer;
5  import se.kth.ict.oodbook.design.casestudy.integration.RegistryCreator;
6  import se.kth.ict.oodbook.design.casestudy.view.View;
7
8  /**
9   * Contains the <code>main</code> method. Performs all startup of
10  * the application.
11  */
12 public class Main {
13     /**
14      * Starts the application.
15      *
16      * @param args The application does not take any command line
17      *             parameters.
18      */
19     public static void main(String[] args) {
20     }
21 }
```

Listing C.160 Java code implementing the `Main` class in figure 5.43

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  import se.kth.ict.oodbook.design.casestudy.integration.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.integration.CarRegistry;
5  import se.kth.ict.oodbook.design.casestudy.integration.Printer;
6
7  /**
8   * Represents one particular rental transaction, where one
9   * particular car is rented by one particular customer.
10  */
11 public class Rental {
```

```
12      private CarRegistry carRegistry;
13
14      /**
15       * Creates a new instance, representing a rental made by the
16       * specified customer.
17       *
18       * @param customer The renting customer.
19       * @param carRegistry The data store with information about
20       *                    available cars.
21       */
22      public Rental(CustomerDTO customer, CarRegistry carRegistry) {
23      }
24
25      /**
26       * Specifies the car that was rented.
27       *
28       * @param rentedCar The car that was rented.
29       */
30      public void setRentedCar(CarDTO rentedCar) {
31      }
32
33      /**
34       * This rental is paid using the specified payment.
35       *
36       * @param payment The payment used to pay this rental.
37       */
38      public void pay(CashPayment payment) {
39      }
40
41      /**
42       * Prints a receipt for the current rental on the specified
43       * printer.
44       */
45      public void printReceipt(Printer printer) {
46      }
47  }
```

Listing C.161 Java code implementing the `Rental` class in figure 5.43

```
1   package se.kth.ict.oodbook.design.casestudy.model;
2
3   /**
4    * The receipt of a rental
5    */
6   public class Receipt {
7       /**
```

```
8       * Creates a new instance.
9       *
10      * @param rental The rental proved by this receipt.
11      */
12     Receipt(Rental rental) {
13     }
14 }
```

Listing C.162 Java code implementing the `Receipt` class in figure 5.43

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents one specific payment for one specific rental.
5   * The rental is payed with cash.
6   */
7  public class CashPayment {
8      /**
9       * Creates a new instance. The customer handed over
10      * the specified amount.
11      *
12      * @param paidAmt The amount of cash that was handed
13      *                over by the customer.
14      */
15     public CashPayment(Amount paidAmt) {
16     }
17
18     /**
19      * Calculates the total cost of the specified rental.
20      *
21      * @param paidRental The rental for which the customer
22      *                   is paying.
23      */
24     void calculateTotalCost(Rental paidRental) {
25     }
26 }
```

Listing C.163 Java code implementing the `CashPayment` class in figure 5.43

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents a cash register. There shall be one instance
5   * of this class for each register.
6   */
```

```
7  public class CashRegister {
8      public void addPayment(CashPayment payment) {
9      }
10 }
```

Listing C.164 Java code implementing the `CashRegister` class in figure 5.43

```
1  package se.kth.ict.oodbook.design.casestudy.integration;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Receipt;
4
5  /**
6   * The interface to the printer, used for all printouts
7   * initiated by this program.
8   */
9  public class Printer {
10     public void printReceipt(Receipt receipt) {
11     }
12 }
```

Listing C.165 Java code implementing the `Printer` class in figure 5.43

```
1  package se.kth.ict.oodbook.design.casestudy.integration;
2
3  /**
4   * This class is responsible for instantiating all registries.
5   */
6  public class RegistryCreator {
7      /**
8       * Get the value of rentalRegistry
9       *
10      * @return the value of rentalRegistry
11      */
12     public RentalRegistry getRentalRegistry() {
13     }
14
15     /**
16      * Get the value of carRegistry
17      *
18      * @return the value of carRegistry
19      */
20     public CarRegistry getCarRegistry() {
21     }
22 }
```

Listing C.166 Java code implementing the `RegistryCreator` class in figure 5.43

```
package se.kth.ict.oodbook.design.casestudy.integration;

/**
 * Contains all calls to the data store with cars that may be
 * rented.
 */
public class CarRegistry {
    CarRegistry() {
    }

    /**
     * Search for a car matching the specified search criteria.
     *
     * @param searchedCar This object contains the search criteria.
     *                    Fields in the object that are set to
     *                    <code>null</code> or <code>false</code>
     *                    are ignored.
     * @return The best match of the search criteria.
     */
    public CarDTO findCar(CarDTO searchedCar) {
    }

    /**
     * Books the specified car. After calling this method, the car
     * can not be booked by any other customer.
     *
     * @param car The car that will be booked.
     */
    public void bookCar(CarDTO car) {
    }
}
```

Listing C.167 Java code implementing the `CarRegistry` class in figure 5.43

```
package se.kth.ict.oodbook.design.casestudy.integration;

import se.kth.ict.oodbook.design.casestudy.model.Rental;

/**
 * Contains all calls to the data store with performed rentals.
 */
```

```
8  public class RentalRegistry {
9      RentalRegistry() {
10     }
11
12     /**
13      * Saves the specified rental permanently.
14      *
15      * @param rental The rental that will be saved.
16      */
17     public void saveRental(Rental rental) {
18     }
19 }
```

Listing C.168 Java code implementing the `RentalRegistry` class in figure 5.43

C.33 Figure 5.44

```
1  package se.kth.ict.oodbook.design.casestudy.view;
2
3  import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4  import se.kth.ict.oodbook.design.casestudy.model.Amount;
5
6  /**
7   * This program has no view, instead, this class is a placeholder
8   * for the entire view.
9   */
10 public class View {
11     private Controller contr;
12
13     // Somewhere in the code. The used amount (100) is not
14     // specified in the diagram.
15     Amount paidAmount = new Amount(100);
16     contr.pay();
17 }
```

Listing C.169 Java code implementing the `View` class in figure 5.44

```
1  package se.kth.ict.oodbook.design.casestudy.controller;
2
3  import se.kth.ict.oodbook.design.casestudy.model.Amount;
4  import se.kth.ict.oodbook.design.casestudy.model.Rental;
5
6  /**
```

```
 * This is the application's only controller class. All
 * calls to the model pass through here.
 */
public class Controller {
    /**
     * Handles rental payment. Updates the balance of
     * the cash register where the payment was
     * performed. Calculates change. Prints the receipt.
     *
     * @param amount The paid amount.
     */
    public void pay() {
        CashPayment payment = new CashPayment(paidAmt);
        rental.pay(payment);
        cashRegister.addPayment(payment);
    }
}
```

Listing C.170 Java code implementing the `Controller` class in figure 5.44. Note that this is code won't compile, since it's an implementation of a diagram illustrating a mistake.

```
package se.kth.ict.oodbook.design.casestudy.model;

/**
 * Represents a cash register. There shall be one instance of
 * this class for each register.
 */
public class CashRegister {
    public void addPayment(CashPayment payment) {
    }
}
```

Listing C.171 Java code implementing the `CashRegister` class in figure 5.44

```
package se.kth.ict.oodbook.design.casestudy.model;

/**
 * Represents one specific payment for one specific rental. The
 * rental is payed with cash.
 */
public class CashPayment {
    private Amount paidAmt;

    /**
     * Creates a new instance. The customer handed over the
     * specified amount.
```

```
13          *
14          * @param paidAmt The amount of cash that was handed over
15          *                by the customer.
16          */
17         public CashPayment(Amount paidAmt) {
18             this.paidAmt = paidAmt;
19         }
20
21         /**
22          * Calculates the total cost of the specified rental.
23          *
24          * @param paidRental The rental for which the customer is
25          *                   paying.
26          */
27         void calculateTotalCost(Rental paidRental) {
28         }
29     }
```

Listing C.172 Java code implementing the `CashPayment` class in figure 5.44

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5
6  /**
7   * Represents one particular rental transaction, where one
8   * particular car is rented by one particular customer.
9   */
10 public class Rental {
11     /**
12      * This rental is paid using the specified payment.
13      *
14      * @param payment The payment used to pay this rental.
15      */
16     public void pay(CashPayment payment) {
17         payment.calculateTotalCost(this);
18     }
19 }
```

Listing C.173 Java code implementing the `Rental` class in figure 5.44

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
```

```
 * Represents an amount of money
 */
public final class Amount {
    private final int amount;

    public Amount(int amount) {
        this.amount = amount;
    }
}
```

Listing C.174 Java code implementing the `Amount` class in figure 5.44

C.34 Figure 5.45

```
package se.kth.ict.oodbook.design.casestudy.view;

import se.kth.ict.oodbook.design.casestudy.controller.Controller;
import se.kth.ict.oodbook.design.casestudy.model.Amount;

/**
 * This program has no view, instead, this class is a placeholder
 * for the entire view.
 */
public class View {
    private Controller contr;

    // Somewhere in the code. The used amount (100) is not
    // specified in the diagram.
    Amount paidAmount = new Amount(100);
    contr.pay(paidAmount);
}
```

Listing C.175 Java code implementing the `View` class in figure 5.45

```
package se.kth.ict.oodbook.design.casestudy.controller;

import se.kth.ict.oodbook.design.casestudy.model.Amount;
import se.kth.ict.oodbook.design.casestudy.model.Rental;

/**
 * This is the application's only controller class. All
 * calls to the model pass through here.
 */
```

```
10 public class Controller {
11     /**
12      * Handles rental payment. Updates the balance of
13      * the cash register where the payment was
14      * performed. Calculates change. Prints the receipt.
15      *
16      * @param amount The paid amount.
17      */
18     public void pay(Amount paidAmt) {
19         CashPayment payment = new CashPayment(paidAmt);
20         rental.pay();
21         cashRegister.addPayment(payment);
22     }
23 }
```

Listing C.176 Java code implementing the `Controller` class in figure 5.45

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents a cash register. There shall be one instance of
5   * this class for each register.
6   */
7  public class CashRegister {
8      public void addPayment(CashPayment payment) {
9      }
10 }
```

Listing C.177 Java code implementing the `CashRegister` class in figure 5.45

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents one specific payment for one specific rental. The
5   * rental is payed with cash.
6   */
7  public class CashPayment {
8      private Amount paidAmt;
9
10     /**
11      * Creates a new instance. The customer handed over the
12      * specified amount.
13      *
14      * @param paidAmt The amount of cash that was handed over
15      *                by the customer.
```

```
16      */
17     public CashPayment(Amount paidAmt) {
18         this.paidAmt = paidAmt;
19     }
20
21     /**
22      * Calculates the total cost of the specified rental.
23      *
24      * @param paidRental The rental for which the customer is
25      *                   paying.
26      */
27     void calculateTotalCost(Rental paidRental) {
28     }
29 }
```

Listing C.178 Java code implementing the `CashPayment` class in figure 5.45

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4  import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5
6  /**
7   * Represents one particular rental transaction, where one
8   * particular car is rented by one particular customer.
9   */
10 public class Rental {
11     /**
12      * This rental is paid using the specified payment.
13      *
14      * @param payment The payment used to pay this rental.
15      */
16     public void pay() {
17         payment.calculateTotalCost(this);
18     }
19 }
```

Listing C.179 Java code implementing the `Rental` class in figure 5.45. Note that this is code won't compile, since it's an implementation of a diagram illustrating a mistake.

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents an amount of money
5   */
6  public final class Amount {
```

```
7       private final int amount;
8
9       public Amount(int amount) {
10          this.amount = amount;
11      }
12  }
```

Listing C.180 Java code implementing the `Amount` class in figure 5.45

C.35 Figure 5.47

```
1   package se.kth.ict.oodbook.design.casestudy.view;
2
3   import se.kth.ict.oodbook.design.casestudy.controller.Controller;
4   import se.kth.ict.oodbook.design.casestudy.model.Amount;
5
6   /**
7    * This program has no view, instead, this class is a placeholder
8    * for the entire view.
9    */
10  public class View {
11      private Controller contr;
12
13      // Somewhere in the code. The used amount (100) is not
14      // specified in the diagram.
15      Amount paidAmount = new Amount(100);
16      contr.pay(paidAmount);
17  }
```

Listing C.181 Java code implementing the `View` class in figure 5.47

```
1   package se.kth.ict.oodbook.design.casestudy.controller;
2
3   import se.kth.ict.oodbook.design.casestudy.model.Amount;
4   import se.kth.ict.oodbook.design.casestudy.model.Rental;
5
6   /**
7    * This is the application's only controller class. All
8    * calls to the model pass through here.
9    */
10  public class Controller {
11      /**
12       * Handles rental payment. Updates the balance of
```

```
13      * the cash register where the payment was
14      * performed. Calculates change. Prints the receipt.
15      *
16      * @param amount The paid amount.
17      */
18     public void pay(Amount paidAmt) {
19         new CashPayment(paidAmt);
20         rental.pay(payment);
21         cashRegister.addPayment(payment);
22     }
23 }
```

Listing C.182 Java code implementing the `Controller` class in figure 5.47. Note that this is code won't compile, since it's an implementation of a diagram illustrating a mistake.

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents a cash register. There shall be one instance of
5   * this class for each register.
6   */
7  public class CashRegister {
8      public void addPayment(CashPayment payment) {
9      }
10 }
```

Listing C.183 Java code implementing the `CashRegister` class in figure 5.47

```
1  package se.kth.ict.oodbook.design.casestudy.model;
2
3  /**
4   * Represents one specific payment for one specific rental. The
5   * rental is payed with cash.
6   */
7  public class CashPayment {
8      private Amount paidAmt;
9
10     /**
11      * Creates a new instance. The customer handed over the
12      * specified amount.
13      *
14      * @param paidAmt The amount of cash that was handed over
15      *                by the customer.
16      */
17     public CashPayment(Amount paidAmt) {
18         this.paidAmt = paidAmt;
```

```
19      }
20
21      /**
22       * Calculates the total cost of the specified rental.
23       *
24       * @param paidRental The rental for which the customer is
25       *                   paying.
26       */
27      void calculateTotalCost(Rental paidRental) {
28      }
29  }
```

Listing C.184 Java code implementing the `CashPayment` class in figure 5.47

```
1   package se.kth.ict.oodbook.design.casestudy.model;
2
3   import se.kth.ict.oodbook.design.casestudy.dbhandler.CarDTO;
4   import se.kth.ict.oodbook.design.casestudy.dbhandler.CarRegistry;
5
6   /**
7    * Represents one particular rental transaction, where one
8    * particular car is rented by one particular customer.
9    */
10  public class Rental {
11      /**
12       * This rental is paid using the specified payment.
13       *
14       * @param payment The payment used to pay this rental.
15       */
16      public void pay(CashPayment payment) {
17          payment.calculateTotalCost(this);
18      }
19  }
```

Listing C.185 Java code implementing the `Rental` class in figure 5.47.

```
1   package se.kth.ict.oodbook.design.casestudy.model;
2
3   /**
4    * Represents an amount of money
5    */
6   public final class Amount {
7       private final int amount;
8
9       public Amount(int amount) {
```

```
10          this.amount = amount;
11      }
12  }
```

Listing C.186 Java code implementing the `Amount` class in figure 5.47

C.36 Figure 8.2

Package names are not shown in the diagram, but have been added in the code.

```
1  package se.kth.ict.oodbook.exception.uml;
2
3  public class MyClass {
4      public void myMethod() throws MyException, AnotherException {
5      }
6  }
```

Listing C.187 Java code implementing the `MyClass` class in figure 8.2a

```
1  package se.kth.ict.oodbook.exception.uml;
2
3  public class MyException extends Exception {
4  }
```

Listing C.188 Java code implementing the `MyException` class in figure 8.2a

```
1  package se.kth.ict.oodbook.exception.uml;
2
3  public class AnotherException extends Exception {
4  }
```

Listing C.189 Java code implementing the `AnotherException` class in figure 8.2a

```
1  package se.kth.ict.oodbook.exception.uml;
2
3  public class MyClass {
4      // The diagram shows that one or more of the methods in
5      // this class throws MyException, but it does not show
6      // which method(s).
7      public void myMethod() {
8      }
9
```

```
10      public void anotherMethod() {
11      }
12  }
```

Listing C.190 Java code implementing the `MyClass` class in figure 8.2b

```
1  package se.kth.ict.oodbook.exception.uml;
2
3  public class MyException extends Exception {
4  }
```

Listing C.191 Java code implementing the `MyException` class in figure 8.2b

C.37 Figure 8.3

Package names are not shown in the diagram, but have been added in the code. The code is exactly the same for figures 8.3a and 8.3b.

```
1  package se.kth.ict.oodbook.exception.uml;
2
3  public class MyClass {
4      public void myMethod() throws MyException {
5      }
6  }
```

Listing C.192 Java code implementing the `MyClass` class in figure 8.3

```
1  package se.kth.ict.oodbook.exception.uml;
2
3  public class MyException extends Exception {
4  }
```

Listing C.193 Java code implementing the `MyException` class in figure 8.3

```
1  package se.kth.ict.oodbook.exception.uml;
2
3  public class SomeClass {
4      private MyClass myClass;
5
6          // Somewhere, in some method in this class:
7          try {
8              myClass.myMethod();
```

```
9           } catch (MyException exception) {
10
11          }
12  }
```

Listing C.194 Java code implementing the `SomeClass` class in figure 8.3. Strictly speaking, the UML diagram does not tell whether there is a `try-catch` block or not.

C.38 Figure 9.1

Package names are not shown in the diagram, but have been added in the code.

```
1  package se.leiflindback.oodbook.polymorphism.uml;
2
3  interface InterfaceA {
4      public void methodA();
5  }
```

Listing C.195 Java code implementing the `InterfaceA` interface in figure 9.1a

```
1  package se.leiflindback.oodbook.polymorphism.uml;
2
3  public class ClassA implements InterfaceA {
4      @Override
5      public void methodA() {
6      }
7  }
```

Listing C.196 Java code implementing the `ClassA` class in figure 9.1a

```
1  package se.leiflindback.oodbook.polymorphism.uml;
2
3  public class ClassB extends ClassC{
4  }
```

Listing C.197 Java code implementing the `ClassB` class in figure 9.1b

```
1  package se.leiflindback.oodbook.polymorphism.uml;
2
3  public class ClassC {
4  }
```

Listing C.198 Java code implementing the `ClassC` class in figure 9.1b

C.39 Figure 9.2

Package names are not shown in the diagram, but have been added in the code.

```
1 package se.leiflindback.oodbook.polymorphism.uml.seqClass;
2
3 public class AnyClass {
4     private ClassA callee;
5
6     //Somewhere in some method.
7     callee.methodA();
8 }
```

Listing C.199 Java code implementing the `AnyClass` object in figure 9.2a

```
1 package se.leiflindback.oodbook.polymorphism.uml.seqClass;
2
3 public class ClassA {
4     public void methodA() {
5     }
6 }
```

Listing C.200 Java code implementing the `ClassA` object in figure 9.2a

```
1 package se.leiflindback.oodbook.polymorphism.uml.seqInterf;
2
3 public class AnyClass {
4     private InterfaceA callee;
5
6     //Somewhere in some method.
7     callee.methodA();
8 }
```

Listing C.201 Java code implementing the `AnyClass` object in figure 9.2b

```
1 package se.leiflindback.oodbook.polymorphism.uml.seqInterf;
2
3 interface InterfaceA {
4     public void methodA();
5 }
```

Listing C.202 Java code implementing the `InterfaceA` object in figure 9.2b

```
1 package se.leiflindback.oodbook.polymorphism.uml.seqInterfClass;
```

```
2
3 public class AnyClass {
4     private InterfaceA callee;
5
6     //Somewhere in some method.
7     callee.methodA();
8 }
```

Listing C.203 Java code implementing the `AnyClass` object in figure 9.2c

```
1  package se.leiflindback.oodbook.polymorphism.uml.seqInterfClass;
2
3  public class SomeUnknownClass implements InterfaceA {
4      private ClassA callee;
5
6      @Override
7      public void methodA() {
8          callee.methodA();
9      }
10 }
```

Listing C.204 Java code implementing the `InterfaceA` object in figure 9.2c

```
1  package se.leiflindback.oodbook.polymorphism.uml.seqInterfClass;
2
3  public class ClassA {
4      public void methodA() {
5      }
6  }
```

Listing C.205 Java code implementing the `ClassA` object in figure 9.2c

C.40 Figure 9.3

The implementation of this diagram identical to that of diagram 9.2b above (listings C.201 and C.202), since these two figures are different ways to show exactly the same thing.

C.41 Figure 9.6

```
1 package package1;
2
```

```
3 public class Class1 {
4     protected void protectedMethod() {
5
6     }
7 }
```

Listing C.206 Java code implementing `Class1` in figure 9.6a.

```
1 package package1;
2
3 public class Class2 extends Class1 {
4
5 }
```

Listing C.207 Java code implementing `Class2` in figure 9.6a.

```
1 package package1;
2
3 public class Class1 {
4     protected void protectedMethod() {
5
6     }
7 }
```

Listing C.208 Java code implementing `Class1` in figure 9.6b.

```
1 package package1;
2
3 public class Class2 {
4     private Class1 class1;
5 }
```

Listing C.209 Java code implementing `Class2` in figure 9.6b.

```
1 package package1;
2
3 public class Class1 {
4     protected void protectedMethod() {
5
6     }
7 }
```

Listing C.210 Java code implementing `Class1` in figure 9.6c.

```
1 package package2;
2
3 import package1.Class1;
4
5 public class Class2 extends Class1 {
6
7 }
```

Listing C.211 Java code implementing `Class2` in figure 9.6c.

```
1 package package1;
2
3 public class Class1 {
4     protected void protectedMethod() {
5
6     }
7 }
```

Listing C.212 Java code implementing `Class1` in figure 9.6d.

```
1 package package2;
2
3 import package1.Class1;
4
5 public class Class2 {
6     private Class1 class1;
7 }
```

Listing C.213 Java code implementing `Class2` in figure 9.6d.

C.42 Figure 9.7

```
1  public class Animal {
2      private String name;
3  }
4
5  public class Bird extends Animal {
6      public void fly() {
7      }
8  }
9
10 public class Fish extends Animal {
```

```
11      public void swim() {
12      }
13  }
14
15  public class Mammal extends Animal {
16      public void walk() {
17      }
18  }
19
20  public class Crow extends Bird {
21  }
22
23  public class Parrot extends Bird {
24  }
25
26  public class Salmon extends Fish {
27  }
28
29  public class Perch extends Fish {
30  }
31
32  public class Dog extends Mammal {
33  }
34
35  public class Cat extends Mammal {
36  }
```

Listing C.214 Java code implementing all classes in figure 9.7.

C.43 Figure 9.8

```
1   public class Animal {
2       private String name;
3   }
4
5   public class Flyer extends Animal {
6       public void fly() {
7       }
8   }
9
10  public class Swimmer extends Animal {
11      public void swim() {
12      }
13  }
```

```
14
15 public class Walker extends Animal {
16     public void walk() {
17     }
18 }
19
20 public class Parrot extends Flyer {
21 }
22
23 public class Crow extends Flyer {
24 }
25
26 public class Salmon extends Swimmer {
27 }
28
29 public class Perch extends Swimmer {
30 }
31
32 public class Penguin extends Swimmer {
33 }
34
35 public class Cat extends Walker {
36 }
37
38 public class Dog extends Walker {
39 }
```

Listing C.215 Java code implementing all classes in figure 9.8.

C.44 Figure 9.9

```
1  public class Animal {
2      private String name;
3  }
4
5  public class Flyer {
6      public void fly() {
7      }
8  }
9
10 public class Swimmer {
11     public void swim() {
12     }
13 }
```

```
14
15 public class Walker {
16     public void walk() {
17     }
18 }
19
20 public class Parrot {
21     private Animal animal;
22     private Flyer flyer;
23
24     public void fly() {
25         flyer.fly();
26     }
27 }
28
29 public class Crow {
30     private Animal animal;
31     private Flyer flyer;
32
33     public void fly() {
34         flyer.fly();
35     }
36 }
37
38 public class Salmon {
39     private Animal animal;
40     private Swimmer swimmer;
41
42     public void swim() {
43         swimmer.swim();
44     }
45 }
46
47 public class Perch {
48     private Animal animal;
49     private Swimmer swimmer;
50
51     public void swim() {
52         swimmer.swim();
53     }
54 }
55
56 public class Ostrich {
57     private Animal animal;
58     private Walker walker;
59
```

```
60      public void walk() {
61          walker.walk();
62      }
63  }
64
65  public class Cat {
66      private Animal animal;
67      private Walker walker;
68      private Swimmer swimmer;
69
70      public void walk() {
71          walker.walk();
72      }
73
74      public void swim() {
75          swimmer.swim();
76      }
77  }
```

Listing C.216 Java code implementing all classes in figure 9.9.

C.45 Figure 9.12

Package names are not shown in the diagram, but have been added in the code.

```
1   package se.leiflindback.oodbook.despat.observer;
2
3   import java.util.ArrayList;
4   import java.util.List;
5
6   /**
7    * The observed class in a general implementation of the
8    * observer pattern.
9    */
10  public class ObservedClass {
11      private List<Observer> observers = new ArrayList<>();
12
13      /**
14       * Registers observers. Any <code>Observer</code> that is
15       * passed to this method will be notified when this object
16       * changes state.
17       *
18       * @param observer The observer that shall be registered.
19       */
20      public void addObserver(Observer observer) {
```

```
21          observers.add(observer);
22      }
23
24      // Called by any method in this class that has changed the
25      // class' state.
26      private void notifyObservers() {
27          for (Observer observer : observers) {
28              observer.stateHasChanged();
29          }
30      }
31  }
```

Listing C.217 Java code implementing `ObservedClass` in figure 9.12. The method `notifyObservers` is not shown in the diagram.

```
1  package se.leiflindback.oodbook.despat.observer;
2
3  /**
4   * The observer interface in a general implementation of the
5   * observer pattern.
6   */
7  public interface Observer {
8      /**
9       * Called when the observed class changes state.
10      */
11     void stateHasChanged();
12 }
```

Listing C.218 Java code implementing `Observer` in figure 9.12.

```
1  package se.leiflindback.oodbook.despat.observer;
2
3  /**
4   * The observing class in a general implementation of the
5   * observer pattern.
6   */
7  public class AnyClassThatImplementsObserver implements
8                                                  Observer {
9      @Override
10     public void stateHasChanged() {
11     }
12 }
```

Listing C.219 Java code implementing `AnyClassThatImplementsObserver` in figure 9.12.

C.46 Figure 9.14

Package names are not shown in the diagram, but have been added in the code.

```
1  package se.leiflindback.oodbook.polymInherit.strategy;
2
3  /**
4   * A class that uses a concrete strategy.
5   */
6  public class Client {
7      private StrategyDefinition strategy;
8  }
```

Listing C.220 Java code implementing `Client` in figure 9.14.

```
1  package se.leiflindback.oodbook.polymInherit.strategy;
2
3  /**
4   * A general example of definition of a strategy.
5   */
6  public interface StrategyDefinition {
7      /**
8       * The work performed by this strategy.
9       */
10     public void algorithm();
11 }
```

Listing C.221 Java code implementing `StrategyDefinition` in figure 9.14.

```
1  package se.leiflindback.oodbook.polymInherit.strategy;
2
3  /**
4   * A concrete implementation of a strategy.
5   */
6  public class ConcreteStrategyA implements StrategyDefinition {
7      @Override
8      public void algorithm() {
9      }
10 }
```

Listing C.222 Java code implementing `ConcreteStrategyA` in figure 9.14.

```
1  package se.leiflindback.oodbook.polymInherit.strategy;
2
```

```
3  /**
4   * A concrete implementation of a strategy.
5   */
6  public class ConcreteStrategyB implements StrategyDefinition {
7      @Override
8      public void algorithm() {
9      }
10 }
```

Listing C.223 Java code implementing `ConcreteStrategyB` in figure 9.14.

C.47 Figure 9.17

Package names are not shown in the diagram, but have been added in the code.

```
1  package se.leiflindback.oodbook.polymInherit.factory;
2
3  /**
4   * A class that uses a concrete strategy.
5   */
6  public class Client {
7      private Product product = new Factory().createProduct();
8  }
```

Listing C.224 Java code implementing `Client` in figure 9.17.

```
1  package se.leiflindback.oodbook.polymInherit.factory;
2
3  /**
4   * A generic example of a definition of products created by
5   * a factory.
6   */
7  public interface Product {
8  }
```

Listing C.225 Java code implementing `Product` in figure 9.17.

```
1  package se.leiflindback.oodbook.polymInherit.factory;
2
3  /**
4   * An example of a concrete product
5   */
6  public class ConcreteProductA implements Product {
```

```
7 }
```

Listing C.226 Java code implementing `ProductA` in figure 9.17.

```
1 package se.leiflindback.oodbook.polymInherit.factory;
2
3 /**
4  * An example of a concrete product
5  */
6 public class ConcreteProductB implements Product {
7 }
```

Listing C.227 Java code implementing `ProductB` in figure 9.17.

```
1  package se.leiflindback.oodbook.polymInherit.factory;
2
3  /**
4   * A generic example of a factory.
5   */
6  public class Factory {
7
8      /**
9       * Creates a new product.
10      * @return The newly created product.
11      */
12     public Product createProduct() {
13         return new ConcreteProductA();
14     }
15 }
```

Listing C.228 Java code implementing `Factory` in figure 9.17.

C.48 Figures 9.21 and 9.22

Diagrams 9.21 and 9.22 illustrate exactly the same code. Package names are not shown in the diagrams, but have been added in the code.

```
1 package se.leiflindback.oodbook.polyminherit.composite;
2
3 /**
4  * The client of the algorithm.
5  */
6 public class Client {
```

```
7       private Task task;
8
9       public void anyMethodInThisClass() {
10          // Task shall be a reference to Composite, but the
11          // diagrams do not show how that reference is created.
12          task.performTask();
13      }
14  }
```

Listing C.229 Java code implementing `Client` in figurs 9.21 and 9.22.

```
1   package se.leiflindback.oodbook.polyminherit.composite;
2
3   /**
4    * A definition of an algorithm.
5    */
6   public interface Task {
7       /**
8        * Performs the algorithm.
9        */
10      void performTask();
11  }
```

Listing C.230 Java code implementing `Task` in figurs 9.21 and 9.22.

```
1   package se.leiflindback.oodbook.polyminherit.composite;
2
3   /**
4    * An implementation of the algorithm.
5    */
6   public class ConcreteTaskA implements Task {
7       @Override
8       public void performTask() {
9       }
10  }
```

Listing C.231 Java code implementing `ConcreteTaskA` in figurs 9.21 and 9.22.

```
1   package se.leiflindback.oodbook.polyminherit.composite;
2
3   /**
4    * An implementation of the algorithm.
5    */
6   public class ConcreteTaskB implements Task {
```

```
7       @Override
8       public void performTask() {
9       }
10  }
```

Listing C.232 Java code implementing `ConcreteTaskB` in figurs 9.21 and 9.22.

```
1   package se.leiflindback.oodbook.polyminherit.composite;
2
3   import java.util.List;
4
5   /**
6    * A composite algorithm, containing concrete implementations
7    * of the algorithm.
8    */
9   public class Composite implements Task {
10      // The diagrams do not show how concrete tasks are added
11      // to the list.
12      private List<Task> tasks;
13
14      @Override
15      public void performTask() {
16          for (Task task : tasks) {
17              task.performTask();
18          }
19      }
20  }
```

Listing C.233 Java code implementing `Composite` in figurs 9.21 and 9.22.

C.49 Figure 9.24

Diagrams 9.24a and 9.24b illustrate exactly the same code. Package names are not shown in the diagrams, but have been added in the code.

```
1   package se.leiflindback.oodbook.polyminherit.tempmet;
2
3   /**
4    * A class that uses a concrete implementation of a template
5    * method.
6    */
7   public class Client {
8       private TaskTemplate task;
9       public void anyMethod() {
```

```
10          task.performTask();
11      }
12  }
```

Listing C.234 Java code implementing `Client` in figure 9.24.

```
1  package se.leiflindback.oodbook.polyminherit.tempmet;
2
3  /**
4   * The abstract superclass providing the template method.
5   */
6  public abstract class TaskTemplate {
7      public void performTask() {
8          // Some code, which is common for all concrete
9          // subclasses.
10         doPerformTask();
11         // Some code, which is common for all concrete
12         // subclasses.
13     }
14
15     /**
16      * Subclasses must provide an implementation of
17      * this method.
18      */
19     protected abstract void doPerformTask();
20 }
```

Listing C.235 Java code implementing `TaskTemplate` in figure 9.24.

```
1  package se.leiflindback.oodbook.polyminherit.tempmet;
2
3  /**
4   * A concrete implementation of the template.
5   */
6  public class ConcreteTaskA extends TaskTemplate {
7      @Override
8      protected void doPerformTask() {
9          // Some code, which is specific for ConcreteTaskA.
10     }
11 }
```

Listing C.236 Java code implementing `ConcreteTaskA` in figure 9.24.

```
1  package se.leiflindback.oodbook.polyminherit.tempmet;
```

```
2
3  /**
4   * A concrete implementation of the template.
5   */
6  public class ConcreteTaskB extends TaskTemplate {
7      @Override
8      protected void doPerformTask() {
9          // Some code, which is specific for ConcreteTaskB.
10     }
11 }
```

Listing C.237 Java code implementing `ConcreteTaskB` in figure 9.24.

Appendix D

Solutions to Exercises

This appendix contains solutions to all exercises.

D.1 Solutions to Exercises in Chapter 7, Testing

Here are suggested solutions to exercises on testing, chapter 7. A correct solution doesn't have to look exactly as these solutions. As always when concerning design and programming, there are more than one correct solution, but remember that not all solutions are correct.

Exercise 1

```
1  package se.leiflindback.oodbook.tests.exercises.acct;
2
3  import org.junit.jupiter.api.AfterEach;
4  import org.junit.jupiter.api.BeforeEach;
5  import org.junit.jupiter.api.Test;
6  import static org.junit.jupiter.api.Assertions.*;
7
8  public class HolderTest {
9
10     private String holderNameNisse = "Nisse";
11     private Holder holderNisse;
12
13     @BeforeEach
14     public void setUp() {
15         holderNisse = new Holder(holderNameNisse);
16     }
17
18     @AfterEach
19     public void tearDown() {
20         holderNisse = null;
21     }
22
23     @Test
```

```
24      public void testGetName() {
25          String expResult = holderNameNisse;
26          String result = holderNisse.getName();
27          assertEquals(expResult, result,
28                       "Wrong name returned by getName");
29      }
30  }
```

Listing D.1 The test for the `getName` method in `Holder`.

Exercise 2

Here are suggested tests for the `Holder` class. They will not cover one hundred percent of the code, nor do they include all possible input values. There are for example no tests for constructors or setters that just store a value, and no tests for getters that just return a value. Still, these tests do give a reasonable test coverage.

constructor Test that a holder number is generated, and that each holder gets a unique holder number. It's not possible to know exactly which holder number shall be generated, but we know that it's zero if no number was generated, since the default value of a long is zero. Theoretically, the holder number generator could generate the number zero, but that risk is vanishingly small.

`addAccount` and `getAccounts` These are a setter and a getter, but it's best to test them anyway since there's some handling of the account collection in `getAccounts`. Test that the correct accounts are returned both when no accounts have been added and when some accounts have been added.

`hashCode` Test that the expected hash code is returned. It could be argued that such a test is meaningless, since exactly the same calculation will be done to generate the hash code in the test and in the SUT. This is however not the whole truth. What we actually do is to write the desired algorithm for hash code generation in the test, then the test verifies that's the algorithm actually used in the SUT. If instead we just copy the hash code generation from the SUT to the test, then it's truly meaningless since the same bugs (if there are any) will exist in both the SUT and the test.

`equals` Test that two equal objects are equal according to the `equals` method. Also test that an object is not equal to `null`, to an object of another class, and to a different `Holder`. This will cover all if-branches in the `equals` method.

`toString` Verify that all data is present in the string representation. In this case, the data is holder name, holder number and account numbers. We can also check that the string "Holder" is present, since the string representation tells that this is a holder.

Exercise 3

Here's the complete test for the `Holder` class, it can also be found in the accompanying Git repository [Code].

```
1  package se.leiflindback.oodbook.tests.exercises.acct;
2
3  import java.util.HashSet;
4  import java.util.Set;
5  import org.junit.jupiter.api.AfterEach;
6  import org.junit.jupiter.api.BeforeEach;
7  import org.junit.jupiter.api.Test;
8  import static org.junit.jupiter.api.Assertions.*;
9
10 public class HolderTest {
11
12     private String holderNameNisse = "Nisse";
13     private Holder holderNisse;
14
15     @BeforeEach
16     public void setUp() {
17         holderNisse = new Holder(holderNameNisse);
18     }
19
20     @AfterEach
21     public void tearDown() {
22         holderNisse = null;
23     }
24
25     @Test
26     public void testHolderNoIsGenerated() {
27         long holderNo = holderNisse.getHolderNo();
28         assertNotEquals(holderNo, 0,
29         "No holder no was generated");
30     }
31
32     @Test
33     public void testHolderNoIsNotSameEachTime() {
34         long holderNo =
35             new Holder(holderNameNisse).getHolderNo();
36         assertNotEquals(holderNo, holderNisse.getHolderNo(),
37                         "Two holders got same holder no");
38     }
39
40     @Test
41     public void testGetName() {
42         String expResult = holderNameNisse;
```

```
43          String result = holderNisse.getName();
44          assertEquals(expResult, result,
45                       "Wrong name returned by getName");
46      }
47
48      @Test
49      public void testAcctHandling() {
50          int noOfAcctsOwnedByNisse = 12;
51          Set<Account> expResult =
52              addAcctsToHolderNisse(noOfAcctsOwnedByNisse);
53          Set<Account> result = holderNisse.getAccounts();
54          assertEquals(result, expResult,
55                       "Wrong accounts owned by holder.");
56      }
57
58      @Test
59      public void testAcctHandlingWhenThereAreNoAccts() {
60          Set<Account> expResult = new HashSet<>();
61          Set<Account> result = holderNisse.getAccounts();
62          assertTrue(result.equals(expResult),
63                     "Wrong accounts owned by holder.");
64      }
65
66      @Test
67      public void testHashCode() {
68          assertEquals(holderNisse.hashCode(),
69             Long.valueOf(holderNisse.getHolderNo()).hashCode(),
70                          "Wrong hash code");
71      }
72
73      @Test
74      public void testHolderIsEqualToItself() {
75          assertTrue(holderNisse.equals(holderNisse),
76                     "Holder is not equal to itself");
77      }
78
79      @Test
80      public void testHolderIsNotEqualToNull() {
81          assertFalse(holderNisse.equals(null),
82                      "Holder was equal to null");
83      }
84
85      @Test
86      public void testHolderIsNotEqualToOtherType() {
87          assertFalse(holderNisse.equals(new Object()),
88                      "Holder was equal to java.lang.Object");
```

```
89      }
90
91      @Test
92      public void testHolderIsNotEqualToOtherHolder() {
93          assertFalse(holderNisse.
94                      equals(new Holder(holderNameNisse)),
95                             "Holder was equal to other holder");
96      }
97
98      @Test
99      public void testStringRepContainsState() {
100         int noOfAcctsOwnedByNisse = 7;
101         Set<Account> acctsOwnedByNisse =
102             addAcctsToHolderNisse(noOfAcctsOwnedByNisse);
103         String strRep = holderNisse.toString();
104         assertTrue(strRep.contains("Holder"),
105                    "wrong class name in string");
106         assertTrue(strRep.contains(holderNameNisse),
107                    "wrong holder name in string");
108         assertTrue(strRep.contains(Long.toString(
109                        holderNisse.getHolderNo())),
110                        "wrong holder no in string");
111         for (Account acct : acctsOwnedByNisse) {
112             assertTrue(strRep.contains(
113                             Long.toString(acct.getAcctNo())),
114                             "Acct missing in string");
115         }
116     }
117
118     private Set<Account> addAcctsToHolderNisse(
119                               int noOfAcctsOwnedByNisse) {
120         Set<Account> accountsOwnedByNisse = new HashSet<>();
121         for (int i = 0; i < noOfAcctsOwnedByNisse; i++) {
122             Account acct = new Account(holderNisse);
123             holderNisse.addAccount(acct);
124             accountsOwnedByNisse.add(acct);
125         }
126         return accountsOwnedByNisse;
127     }
128 }
```

Listing D.2 The complete test for the `Holder` class.

Exercise 4

Here are suggested tests for the `Account` class. They don't cover one hundred percent of the code, nor do they include all possible input values. There are for example no tests for constructors or setters that just store a value, and no tests for getters that just return a value. Still, these tests give a reasonable test coverage.

constructor Test that the constructor with fewer arguments creates an account with the balance zero. Also test that an account number is generated, and that each account gets a unique number. It's not possible to know exactly which account number that shall be generated, but we know that it's zero if no number was generated, since the default value of a long is zero. Theoretically, the account number generator could generate the number zero, but that risk is vanishingly small.

`withdraw` Test successful withdrawal, both with positive balance after the withdrawal and with the balance zero after the withdrawal. There shouldn't be any differences between these two tests, both are handled the same way. It's still a good idea to test a withdrawal that gives the resulting balance zero, since that can be considered a possible corner case. It's also necessary to test withdrawal of a negative amount, withdrawal of the amount zero, and overdraft, but all three involves exception handling and are therefore left to chapter 8.

`deposit` Since depositing a negative amount, or zero, involves exception handling, which will be tested in chapter 8, the only test here is of a successful withdrawal.

`hashCode` Test that the expected hash code is returned. It could be argued that such a test is meaningless, since exactly the same calculation will be done to generate the hash code in the test and in the SUT. This is however not the whole truth. What we actually do is to write the desired algorithm for hash code generation in the test, then the test verifies that algorithm is actually used in the SUT. If instead we just copy the hash code generation from the SUT to the test, it's truly meaningless since the same bugs (if there are any) will exist in both the SUT and the test.

`equals` Test that two equal objects are equal according to the `equals` method. Also test that an object is not equal to `null`, to an object of another class, and to a different `Account`. This will cover all if-branches in the `equals` method.

`toString` Verify that all data is present in the string representation. In this case, the data is holder number, balance and account number. We can also check that the string "Account" is present, since the string representation tells that this is an account.

Exercise 5

Here's the complete test for the `Account` class, it can also be found in the accompanying Git repository [Code].

```
1  package se.leiflindback.oodbook.tests.exercises.acct;
2
3  import org.junit.jupiter.api.AfterEach;
4  import org.junit.jupiter.api.BeforeEach;
5  import org.junit.jupiter.api.Test;
6  import static org.junit.jupiter.api.Assertions.*;
7
8  public class AccountTest {
9      private String holderNameFia = "Fia";
10     private Holder holderFia = new Holder(holderNameFia);
11     private int initBalance10 = 10;
12     private Account acctFia10;
13
14     @BeforeEach
15     public void setUp() {
16         acctFia10 = new Account(holderFia, initBalance10);
17     }
18
19     @AfterEach
20     public void tearDown() {
21         acctFia10 = null;
22     }
23
24     @Test
25     public void testConstrWithoutBalance() {
26         Account instance = new Account(holderFia);
27         int expResult = 0;
28         int result = instance.getBalance();
29         assertEquals(expResult, result,
30                      "Wrong balance returned when acct created"
31                      + " with no-balance constructor");
32     }
33
34     @Test
35     public void testAcctNoIsGenerated() {
36         long result = acctFia10.getAcctNo();
37         assertNotEquals(result, 0,
38                         "No acct no was generated");
39     }
40
41     @Test
42     public void testAcctNoIsNotSameEachTime() {
```

```
43          long acctNo = new Account(holderFia).getAcctNo();
44          assertNotEquals(acctNo, acctFia10.getAcctNo(),
45                          "Two accounts got same acct no");
46      }
47
48      @Test
49      public void testSuccessfulWithdrawal() {
50          try {
51              int amtToWithdraw = 3;
52              int expResult = initBalance10 - amtToWithdraw;
53              acctFia10.withdraw(amtToWithdraw);
54              int result = acctFia10.getBalance();
55              assertEquals(expResult, result,
56                           "Wrong balance after withdrawal");
57          } catch (IllegalBankTransactionException ex) {
58              fail("Got exception from successful withdrawal");
59          }
60      }
61
62      @Test
63      public void testWithdrawToBalanzeZero() {
64          try {
65              int expResult = 0;
66              acctFia10.withdraw(initBalance10);
67              int result = acctFia10.getBalance();
68              assertEquals(expResult, result,
69                           "Wrong balance after withdrawal");
70          } catch (IllegalBankTransactionException ex) {
71              fail("Got exception from successful withdrawal");
72          }
73      }
74
75      @Test
76      public void testWithdrawNegAmt() {
77          int amtToWithdraw = -3;
78          try {
79              acctFia10.withdraw(amtToWithdraw);
80              fail("Managed to withdraw neg amount");
81          } catch (IllegalBankTransactionException ex) {
82              assertTrue(ex.getMessage().
83                  contains(Integer.toString(amtToWithdraw)),
84                           "Wrong error message");
85          }
86      }
87
88      @Test
```

```
89      public void testWithdrawZero() {
90          int amtToWithdraw = 0;
91          try {
92              acctFia10.withdraw(amtToWithdraw);
93              fail("Managed to withdraw zero");
94          } catch (IllegalBankTransactionException ex) {
95              assertTrue(ex.getMessage().
96                  contains(Integer.toString(amtToWithdraw)),
97                          "Wrong error message");
98          }
99      }
100
101     @Test
102     public void testOverdraft() {
103         int amtToWithdraw = 11;
104         try {
105             acctFia10.withdraw(amtToWithdraw);
106             fail("Managed to overdraft");
107         } catch (IllegalBankTransactionException ex) {
108             assertTrue(ex.getMessage().
109                 contains(Integer.toString(amtToWithdraw)),
110                         "Wrong error message");
111             assertTrue(ex.getMessage().
112                 contains(Integer.toString(initBalance10)),
113                         "Wrong error message");
114         }
115     }
116
117     @Test
118     public void testSuccessfulDeposit() {
119         try {
120             int amtToDeposit = 3;
121             int expResult = initBalance10 + amtToDeposit;
122             acctFia10.deposit(amtToDeposit);
123             int result = acctFia10.getBalance();
124             assertEquals(expResult, result,
125                         "Wrong balance after deposit");
126         } catch (IllegalBankTransactionException ex) {
127             fail("Got exception from successful deposit");
128         }
129     }
130
131     @Test
132     public void testDepositNegAmt() {
133         int amtToDeposit = -3;
134         try {
```

```
135             acctFia10.deposit(amtToDeposit);
136             fail("Managed to deposit neg amount");
137         } catch (IllegalBankTransactionException ex) {
138             assertTrue(ex.getMessage().
139                 contains(Integer.toString(amtToDeposit)),
140                         "Wrong error message");
141         }
142     }
143
144     @Test
145     public void testDepositZero() {
146         int amtToDeposit = 0;
147         try {
148             acctFia10.deposit(amtToDeposit);
149             fail("Managed to deposit zero");
150         } catch (IllegalBankTransactionException ex) {
151             assertTrue(ex.getMessage().
152                 contains(Integer.toString(amtToDeposit)),
153                         "Wrong error message");
154         }
155     }
156
157     @Test
158     public void testHashCode() {
159         assertEquals(acctFia10.hashCode(),
160            Long.valueOf(acctFia10.getAcctNo()).hashCode(),
161                        "Wrong hash code");
162     }
163
164     @Test
165     public void testAccountIsEqualToItself() {
166         assertTrue(acctFia10.equals(acctFia10),
167                    "Acct is not equal to itself");
168     }
169
170     @Test
171     public void testAcctIsNotEqualToNull() {
172         assertFalse(acctFia10.equals(null),
173                     "Acct was equal to null");
174     }
175
176     @Test
177     public void testAcctIsNotEqualToOtherType() {
178         assertFalse(acctFia10.equals(new Object()),
179                     "Acct was equal to java.lang.Object");
180     }
```

```
181
182     @Test
183     public void testAcctIsNotEqualToOtherAcct() {
184         assertFalse(acctFia10.equals(new Account(holderFia)),
185                     "Acct was equal to other acct");
186     }
187
188     @Test
189     public void testStringRepContainsState() {
190         String strRep = acctFia10.toString();
191         assertTrue(strRep.contains("Account"),
192                    "wrong class name in string");
193         assertTrue(strRep.contains(
194                         Long.toString(holderFia.getHolderNo())),
195                         "wrong holder no in string");
196         assertTrue(strRep.contains(
197                         Integer.toString(initBalance10)),
198                         "wrong balance in string");
199         assertTrue(strRep.contains(
200                         Long.toString(acctFia10.getAcctNo())),
201                         "wrong acct no in string");
202     }
203 }
```

Listing D.3 The complete test for the `Account` class.

Exercise 6

Here's the complete test for the `IllegalBankTransactionException` class, it can also be found in the accompanying Git repository [Code].

```
1  package se.leiflindback.oodbook.tests.exercises.acct;
2
3  import org.junit.jupiter.api.Test;
4  import static org.junit.jupiter.api.Assertions.*;
5
6  public class IllegalBankTransactionExceptionTest {
7
8      @Test
9      public void testMsgIsStored() {
10         String msg = "This is the message";
11         IllegalBankTransactionException instance =
12             new IllegalBankTransactionException(msg);
13         assertEquals(msg, instance.getMessage(),
14                      "Message wasn't stored correctly.");
```

```
15      }
16  }
```

Listing D.4 The complete test for the `IllegalBankTransactionException` class.

Exercise 7

The test for `ClassInModel` is found below, in listing D.5. As always, once the first test has been written it's very easy to add similar tests with other input values.

```
1  package se.leiflindback.oodbook.tests.exercises.contr.model;
2
3  import org.junit.jupiter.api.AfterEach;
4  import org.junit.jupiter.api.Test;
5  import static org.junit.jupiter.api.Assertions.*;
6  import org.junit.jupiter.api.BeforeEach;
7
8  public class ClassInModelTest {
9      private ClassInModel instance;
10
11     @BeforeEach
12     public void setUp() {
13         instance = new ClassInModel();
14     }
15
16     @AfterEach
17     public void tearDown() {
18         instance = null;
19     }
20
21     @Test
22     public void testAddingToZero() {
23         int expResult = 2;
24         int result = instance.addTwo(expResult - 2);
25         assertEquals(expResult, result,
26                      "Wrong result when adding to zero.");
27     }
28
29     @Test
30     public void testAddingToNegOperand() {
31         int expResult = -2;
32         int result = instance.addTwo(expResult - 2);
33         assertEquals(expResult, result,
34                      "Wrong result when adding to negative"
35                      + " operand.");
```

```
36      }
37
38      @Test
39      public void testAdditionThatGivesResultZero() {
40          int expResult = 0;
41          int result = instance.addTwo(expResult - 2);
42          assertEquals(expResult, result,
43                       "Wrong result when result is supposed to"
44                       + " be zero.");
45      }
46  }
```

Listing D.5 The complete test for `ClassInModel`.

Here's the test for `Controller`, there are some things worth noting. First, the actual tests are very similar to those for `ClassInModel`. That's very typical, tests for higher layers are often very similar to tests for the same functionality in lower layers. Second, the method `createObjInModel` can not be tested in isolation, since it's only outcome is to create an object internally in the controller. This outcome is, however, verified in all tests in `ControllerTest` below, since no test would have passed if `ClassInModel` hadn't been instantiated in the calls to `createObjInModel`. Finally, in order to create an object of `Controller`, it's necessary to first create all objects required by `Controller`'s constructor, as is done on lines 21 and 22.

```
1   package
2       se.leiflindback.oodbook.tests.exercises.contr.controller;
3
4   import org.junit.jupiter.api.AfterEach;
5   import org.junit.jupiter.api.Test;
6   import static org.junit.jupiter.api.Assertions.*;
7   import org.junit.jupiter.api.BeforeEach;
8   import se.leiflindback.oodbook.tests.exercises.contr.
9          integration.ClassInInteg;
10  import se.leiflindback.oodbook.tests.exercises.contr.
11         integration.FileHandler;
12
13  public class ControllerTest {
14      private String fileName = "the-file-name";
15      private FileHandler fileHandler;
16      private ClassInInteg classInInteg;
17      private Controller instance;
18
19      @BeforeEach
20      public void setUp() {
21          fileHandler = new FileHandler(fileName);
22          classInInteg = new ClassInInteg();
```

```
23          instance = new Controller(classInInteg, fileHandler);
24      }
25
26      @AfterEach
27      public void tearDown() {
28          fileHandler = null;
29          classInInteg = null;
30          instance = null;
31      }
32
33      @Test
34      public void testAddingToZero() {
35          instance.createObjInModel();
36          int expResult = 2;
37          int result = instance.addTwo(expResult - 2);
38          assertEquals(expResult, result,
39                       "Wrong result when adding to zero.");
40      }
41
42      @Test
43      public void testAddingToNegOperand() {
44          instance.createObjInModel();
45          int expResult = -2;
46          int result = instance.addTwo(expResult - 2);
47          assertEquals(expResult, result,
48                       "Wrong result when adding to negative"
49                       + " operand.");
50      }
51
52      @Test
53      public void testAdditionThatGivesResultZero() {
54          instance.createObjInModel();
55          int expResult = 0;
56          int result = instance.addTwo(expResult - 2);
57          assertEquals(expResult, result,
58                       "Wrong result when result is supposed to"
59                       + " be zero.");
60      }
61  }
```

Listing D.6 The complete test for `Controller`.

Appendix E

Tools, Features and Possibilities Not Covered Elsewhere

This appendix covers topics useful for object-oriented development, which are not covered anywhere else. The reason for them not being covered elsewhere is to minimize the content and make it easy to access, since this is a first course in object-oriented development. The reader shall be able to to get started with object-oriented development without having to understand anything too advanced, too language-specific, or simply less important.

The topics covered here are not ordered by priority. Instead, more general things are covered first, and more language-specific features are covered later.

Overloaded Methods

Many programming languages, for example Java, identifies a method both by its name and its parameters. This means there can be two methods with the same name, in the same class, if their parameter lists differ. When that happens, the method name is said to be *overloaded*. A typical example of overloading is when the same operation is done on parameters of different types, as in listing E.1. Any difference regarding parameter type or count is sufficient to avoid compile errors. The only thing that's not allowed is when methods with the same name have the same number of parameters, and all parameters in the same position in both parameter lists have the same type.

```
1  package se.leiflindback.oodbook.overloading;
2
3  public class Arithmetic {
4      public int sum(int term1, int term2) {
5          return term1 + term2;
6      }
7
8      public float sum(float term1, float term2) {
9          return term1 + term2;
10     }
11 }
```

Listing E.1 The method `sum` is overloaded, since there are two method definitions with the same name. Javadoc is omitted to make the listing shorter.

Regarding design or code smells, there's not much to say about overloaded methods. Exactly the same design guidelines and refactorings are valid for overloaded methods as for other methods, which is why they're not mentioned in any other chapter. It is, however, worth mentioning that in some situations there's a risk of duplicated code in overloading methods, and the solution to that is to let the overloading methods call each other. This is illustrated in listing E.2, where the constructor is overloaded. Without overloading, each constructor would have duplicated a subset of the assignments on lines 26-30.

```
1  package se.leiflindback.oodbook.overloading;
2
3  public class Person {
4
5      private String firstName;
6      private String lastName;
7      private String email;
8      private String phone;
9      private String pnr;
10
11     public Person() {
12         this(null, null);
13     }
14
15     public Person(String firstName, String lastName) {
16         this(firstName, lastName, null, null);
17     }
18
19     public Person(String firstName, String lastName,
20                   String email, String phone) {
21         this(firstName, lastName, email, phone, null);
22     }
23
24     public Person(String firstName, String lastName,
25                   String email, String phone, String pnr) {
26         this.firstName = firstName;
27         this.lastName = lastName;
28         this.email = email;
29         this.phone = phone;
30         this.pnr = pnr;
31     }
32
33 }
```

Listing E.2 The constructor is overloaded, in order to avoid duplicated code. Javadoc is omitted to make the listing shorter.

Java Records

Record Classes, or just *records*, is a very useful feature existing since JDK 16. The functionality offered by records is autogeneration of a constructor, and also of getter, equals, hashCode and toString methods. Consider the very short declaration of a DTO representing a book, in listing E.3. Note the keyword `record` on line 12. If we write this code, the compiler will generate a class similar to that in listing E.4, which relieves us from a lot of repetitive coding.

One particular detail worth noting is that the prefix `get` is omitted from the names of the generated getters. For example, the method that returns the value of the `title` attribute is called `title`, not `getTitle`, see lines 21-23 in listing E.4.

The obvious use of records is for creating DTOs, as in listing E.3. We should in fact always use a record when creating a DTO, unless there's some very specific reason no to, but it's hard to imagine such a reason. There are also other use cases for records, and there's also a bit more to say about what's allowed to write in a record. The purpose here is, however, not to cover records in detail, but just to introduce the feature, and to show that it's very useful for writing DTOs.

```
1  package se.leiflindback.oodbook.records;
2
3  /**
4   * A DTO representing a book.
5   *
6   * @param title The title of the book.
7   * @param author The author of the book.
8   * @param publisher The company that publishes the book.
9   * @param year The year the book was released.
10  * @param ISBN The book's ISBN number.
11  */
12 public record BookDTO(String title, String author,
13                       String publisher, String year,
14                       String ISBN) { }
```

Listing E.3 An example of records, here used to create DTO representing a book.

```
1  package se.leiflindback.oodbook.records;
2
3  public class BookDTO {
4
5      private final String title;
6      private final String author;
7      private final String publisher;
8      private final String year;
9      private final String ISBN;
10
11     public BookDTOWithoutRecords(String title, String author,
```

```
12                                  String publisher, String year,
13                                  String ISBN) {
14          this.title = title;
15          this.author = author;
16          this.publisher = publisher;
17          this.year = year;
18          this.ISBN = ISBN;
19      }
20
21      String title() {
22          return this.title;
23      }
24
25      String author() {
26          return this.author;
27      }
28
29      String publisher() {
30          return this.publisher;
31      }
32
33      String year() {
34          return this.year;
35      }
36
37      String ISBN() {
38          return this.ISBN;
39      }
40
41      @Override
42      public boolean equals(Object other) {
43          // Implementation of equals, which specify that two
44          // objects are equal if they are of the same  type,
45          // and all their attributes have the same value.
46      }
47
48      @Override
49      public int hashCode() {
50         // Implementation of hashCode, which generates the same
51         // hash code for equal objects.
52      }
53
54      public String toString() {
55          return "BookDTO[title=" + title
56                  + ", author=" + author
57                  + ", publisher=" + publisher
```

```
58                  + ", year=" + year + ", ISBN=" + ISBN + "]";
59      }
60  }
```

Listing E.4 The class generated from the record declaration in listing E.3. The code might not look exactly like this, but the functionality is as illustrated here.

Bibliography

[LAR] C. Larman: *Applying UML and Patterns*, third edition, Prentice-Hall 2004, ISBN:0131489062

[JCC] The original Java code convention `http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html`

[JCS] Java Coding Standars from Ambysoft Inc. `http://www.ambysoft.com/downloads/javaCodingStandardsSummary.pdf`

[FOW1ED] M. Fowler: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley 1999, ISBN: 9780201485677

[FOW] M. Fowler: *Refactoring: Improving the Design of Existing Code, 2nd ed*, Addison-Wesley 2018, ISBN: 9780134757599

[JU] Home page for the JUnit unit testing framework `http://junit.org/junit5/`

[NB] Home page for the NetBeans IDE `http://netbeans.apache.org/`

[IJ] Home page for the IntelliJ IDE `http://www.jetbrains.com/idea/`

[Code] A Git repository containing a NetBeans project with all the source code in this text, `https://github.com/oodbook/code` The complete project can also be downloaded as a zip file, `https://github.com/oodbook/code/archive/master.zip`

[GOF] E. Gamma, R. Heml, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1994, ISBN: 0201633612

[Java Tutorial] `https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html`, accessed Feb 22, 2017

Index