

# **A First Course in Object Oriented Development**

**A Hands-On Approach**

Leif Lindbäck

June 7, 2016

# Revision History

Date	Description	Page(s) (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 <code>Amount</code> 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	<i>Create</i> 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 naïv domain model.	31
2016-06-07	Added chapter seven.	130-149
2016-06-07	Split chapter eight.	150-153

# License

Except for figures 5.7, 5.8, 5.11, 5.14, 7.1, 7.2, and 7.6, *A First Course in Object Oriented Development, A Hands-On Approach* by Leif Lindbäck is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License, see <http://creativecommons.org/licenses/by-sa/4.0/>.



# Preface

This text replaces lecture notes for the course IV1350, Object Oriented Design.

It is assumed that the reader has basic knowledge of Java programming, corresponding to one 7.5 hp course, for example ID1018, Programming I. Important concepts, in particular objects and references, are repeated in chapter 1.

Each chapter has an initial section indicating parts of [LAR] covering the same topics. That is not mandatory reading, this text contains all course material. It is intended only for those who wish to dig deeper in a particular subject.

Paragraphs that are crucial to remember when solving seminar tasks are marked with an exclamation mark, like this paragraph. Forgetting the information in such paragraphs might lead to severe misunderstandings.

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

There are Java implementatons of all UML design diagrams in Appendix C. 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. There is a NetBeans project with all Java code in this book, that can be downloaded from the course web [CW].

!

NO!

# Contents

<b>Revision History</b>	<b>i</b>
<b>License</b>	<b>ii</b>
<b>Preface</b>	<b>iii</b>
<b>I Background</b>	<b>1</b>
<b>1 Java Essentials</b>	<b>2</b>
1.1 Further Reading	2
1.2 Objects	2
1.3 References	4
1.4 Arrays and Lists	5
1.5 Exceptions	6
1.6 Javadoc	7
1.7 Annotations	9
1.8 Interfaces	9
1.9 Inheritance	10
<b>2 Introduction</b>	<b>13</b>
2.1 Further Reading	13
2.2 Why Bother About Object Oriented Design?	13
2.3 Software Development Methodologies	13
2.4 Activities During an Iteration	14
2.5 Unified Modeling Language, UML	16
<b>3 The Case Study</b>	<b>17</b>
3.1 Basic Flow	17
3.2 Alternative Flows	18
<b>II Course Content</b>	<b>19</b>
<b>4 Analyses</b>	<b>20</b>
4.1 Further Reading	20
4.2 UML	20

## Contents

4.3	Domain Model . . . . .	24
4.4	System Sequence Diagram . . . . .	33
<b>5</b>	<b>Design . . . . .</b>	<b>39</b>
5.1	Further Reading . . . . .	39
5.2	UML . . . . .	39
5.3	Design Concepts . . . . .	43
5.4	Architecture . . . . .	50
5.5	A Design Method . . . . .	57
5.6	Designing the RentCar Case Study . . . . .	58
5.7	Common Mistakes . . . . .	78
<b>6</b>	<b>Programming . . . . .</b>	<b>80</b>
6.1	Further Reading . . . . .	80
6.2	Dividing the Code in Packages . . . . .	80
6.3	Code Conventions . . . . .	81
6.4	Comments . . . . .	82
6.5	Code Smell and Refactoring . . . . .	83
6.6	Coding Case Study . . . . .	104
6.7	Common Mistakes When Implementing the Design . . . . .	119
<b>7</b>	<b>Testing . . . . .</b>	<b>121</b>
7.1	Unit Tests and The JUnit Framework . . . . .	122
7.2	Unit Testing Best Practices . . . . .	127
7.3	When Testing is Difficult . . . . .	129
7.4	Unit Testing Case Study . . . . .	130
7.5	Common Mistakes When Writing Unit Tests . . . . .	148
<b>8</b>	<b>Exception Handling . . . . .</b>	<b>150</b>
<b>9</b>	<b>Polymorphism and Design Patterns . . . . .</b>	<b>151</b>
<b>10</b>	<b>Inheritance . . . . .</b>	<b>152</b>
<b>11</b>	<b>Inner Classes . . . . .</b>	<b>153</b>
<b>III</b>	<b>Appendices . . . . .</b>	<b>154</b>
<b>A</b>	<b>English-Swedish Dictionary . . . . .</b>	<b>155</b>
<b>B</b>	<b>UML Cheat Sheet . . . . .</b>	<b>157</b>
<b>C</b>	<b>Implementations of UML Diagrams . . . . .</b>	<b>161</b>

## *Contents*

<b>Bibliography</b> . . . . .	<b>254</b>
<b>Index</b> . . . . .	<b>255</b>

# **Part I**

## **Background**



# Chapter 1

## Java Essentials

This text assumes previous knowledge of an object oriented programming language. All code listings are written in Java. This chapter repeats important concepts of Java, but does not cover the whole language.

### 1.1 Further Reading

The code examples are available in a NetBeans project, which can be downloaded from the course web [CW]. Video lectures, which, in addition to explaining concepts, give an introduction to NetBeans, are also available on the course web [CW].

### 1.2 Objects

The goal of object-oriented programming is to declare classes, which are groups of 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. Since fields can not be static, neither can methods since static methods can access only static fields. Static fields and methods are normally not used at all, except in a few, very special, cases.

## 1.3 References

When the `new` operator is used to create an object, it returns a reference to that object. A reference can, like any other value, be stored in variables, sent to methods, sent to constructors, etc. This is illustrated in listing 1.4, which contains a program where a person places food in a dog's bowl. First, a `Bowl` object is created on line three. The reference to that object is stored in the `bowl` variable and passed to the constructor of a `Person` object on line four. On line 16, the `Person` object stores the reference in the `bowl` field, declared on line 12. Then, on line six, the `main` method calls the `feedDog` method in `person`. In `feedDog`, the method `addFood` is called in the previously created `bowl`, on line 20. 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         for (int i = 0; i < 9; i++) {
6             person.feedDog();
7         }
8     }
9 }
10
11 public class Person {
12     private Bowl bowl;
13     private int gramsToAdd = 200;
14
15     public Person(Bowl bowl) {
16         this.bowl = bowl;
17     }
18
19     public void feedDog() {
20         bowl.addFood(gramsToAdd);
21     }
22 }
23
24 public class Bowl {
25     private int gramsOfFoodInBowl;
26
27     public void addFood(int grams) throws Exception {
28         gramsOfFoodInBowl = gramsOfFoodInBowl + grams;
```

```

29     }
30 }

```

**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.

## 1.4 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);

```

**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. 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`.

```

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");

```

**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

## 1.5 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 exception if it inherits the class `java.lang.RuntimeException`.

## 1.6 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

## 1.7 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.

## 1.8 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. The method implementations in the implementing class must do what is intended in the method declarations in the interface. This should be documented in javadoc comments. 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 accidentally naming the method `logg` instead of `log` in the implementing class in listing 1.11.

## 1.9 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.

```

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

```

```

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

```

**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*

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.

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

These topics are covered in chapters one to eight in [LAR], but those chapters are much more extensive than what is required for this course.

### 2.2 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.

### 2.3 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 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. 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 certain way to tell if it still works.

## 2.4 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 important 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 will consist of classes, attributes, etc. However, it shall not describe the program that shall be developed, but rather describe the reality in which the program shall operate. The purpose is to gain a better understanding of this reality, *before* thinking about the program. Analysis is covered 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 introduced in chapter 5.

**Coding** is of course the most important part of development, it is code quality alone that decides if the program works as 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 if 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.

## **2.5 Unified Modeling Language, UML**

Both analysis and design result in plans. The results of analysis are plans of the reality modeled by 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 class diagrams 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 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.

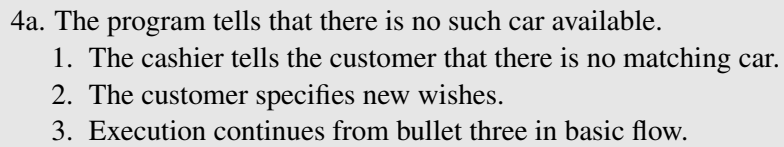
1. The customer arrives and asks to rent a car.
2. The customer describes the desired car.
3. The cashier registers the customer's wishes.
4. The program tells that such a car is available.
5. The cashier describes the car to the customer.
6. The customer agrees to rent the described car.
7. The cashier 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

## Analyses

The purpose of analysis is to create a model, a simplified view, of the reality in which the system under development shall operate. That model will consist of classes, attributes, method calls, etc. However, it shall not describe the program being developed, but rather the reality in which that program operates. The purpose is to gain a better understanding of this reality before thinking about the program. This chapter shows how to develop a *domain model* and a *system sequence diagram*. It also covers the UML needed for those two diagrams.

### 4.1 Further Reading

Analysis is covered in chapters nine and ten in [LAR].

### 4.2 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.

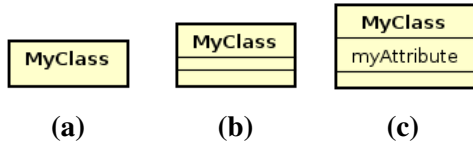
#### Class Diagram

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

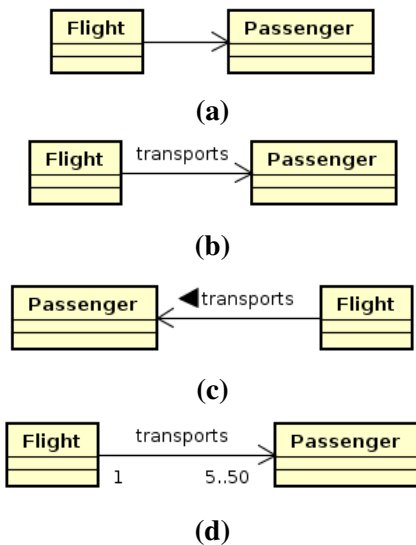
interval, or it might be everything that will ever exist. This must be defined by the diagram author.



**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.

figure 4.1a, specifies only the name of the class, `MyClass`. The second, figure 4.1b, also specifies only the name. Here, however, there are two empty compartments below the name. The upper of these is for specifying attributes. As in object oriented programming, an attribute defines a value that can be attached to an instance 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

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-

A class in UML means the same thing as a class in an object oriented program, a concept or idea not associated with any specific instance of that concept. A class name is always a noun in singular. A class `Person` specifies what a person contains. It does not say anything about specific instances, like the persons `me` or `you`. Figure 4.1 shows three possible ways to draw a class in a class diagram. The first exam-

Classes can have *associations* with other classes. An association between two classes means that instances (objects) of those classes are linked. If the classes 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

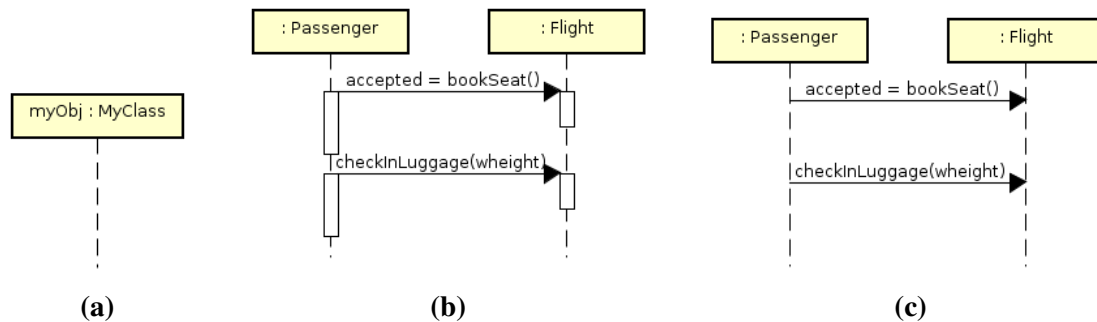
class sequence shall be read, it does not tell anything about the association's direction. It is up to the diagram author to decide if such black triangles shall be used or not. They are most commonly used if class-association-class shall be read from right to left, or bottom up.

UML has different arrows, with different meaning. The arrows 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 the 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, `*`, 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.



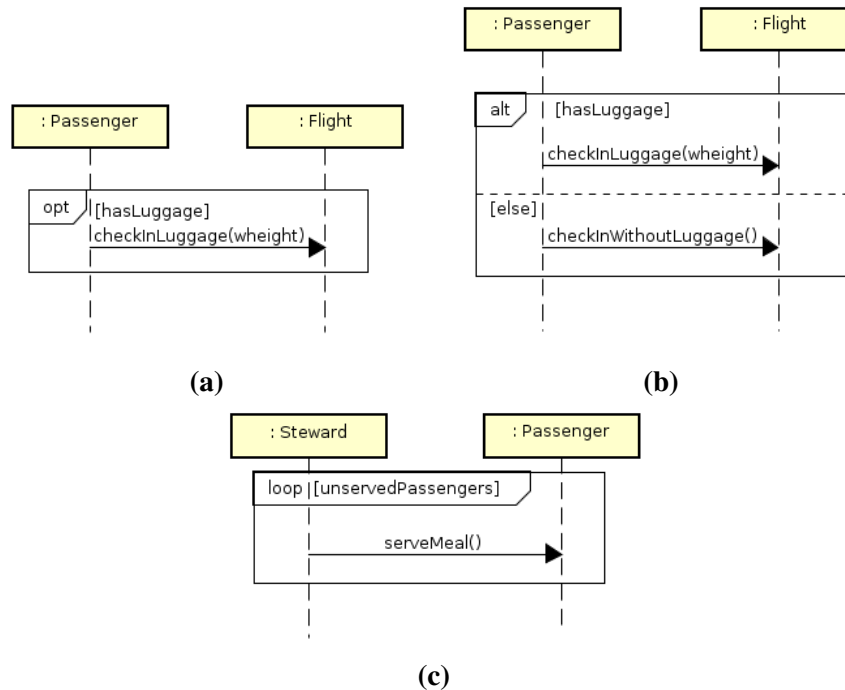
**Figure 4.3** Instances and messages in sequence diagram

- (a) The instance **myObj** of **MyClass**
- (b) Messages with activation bar
- (c) Messages without activation bar

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 `wheight`. 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.

Remember that different arrows have different meaning. 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, `LongDistanceFlight`; object names, attribute names, method names and variable names are written in camel case, `economyClassPassenger`, `luggageCount`, `checkInLuggage`, `unservedPassengers`. !

## Notes

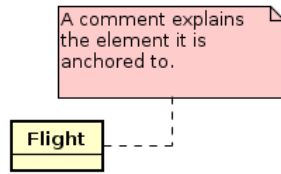


Figure 4.5 A UML comment

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 often called *notes* in UML. A note is anchored to the element it explains with a dashed line, as in figure 4.5.

## 4.3 Domain Model

The *domain model*, DM, is a model of the reality that shall be represented in the program under development. A UML class diagram is used to construct the domain model. The elements in the DM are not classes in an object oriented program, but instead things that exist in reality. 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. Although the DM does not depict the program, it will still prove to be very useful when constructing the program. Since it is a model of the reality being modeled in software, it is highly likely that the program will, when ready, have many things in common with the domain model.

### 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, use both methods.

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.

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.

### 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!

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** paid 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.

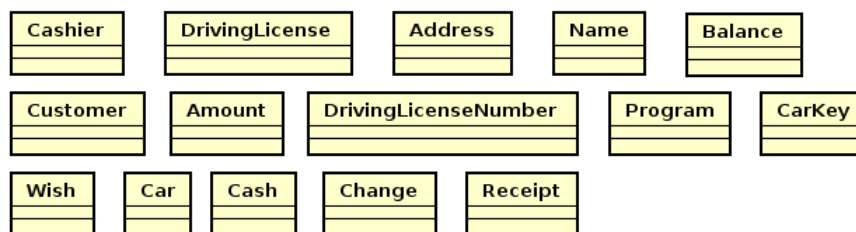


Figure 4.7 The first draft of the domain model, after noun identification

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.

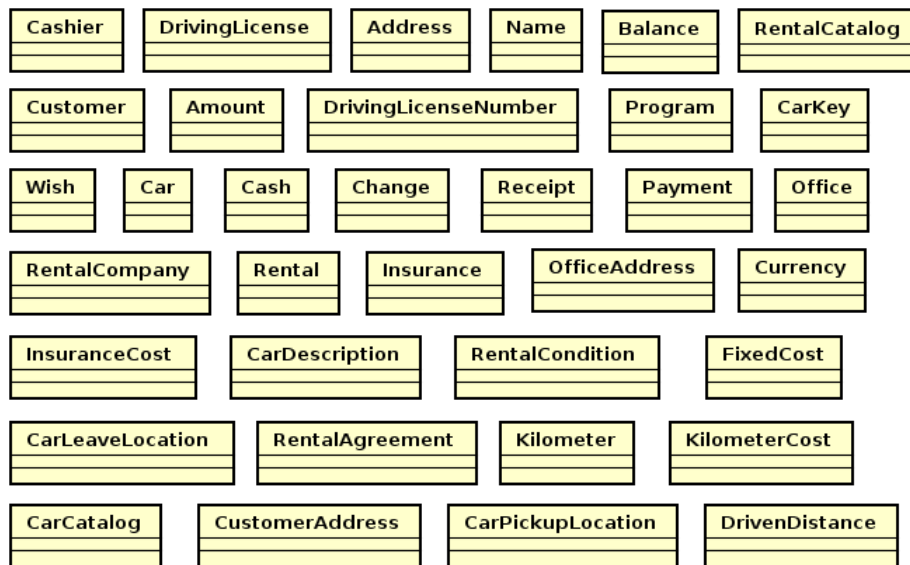
Category	Class Candidates
Transactions	Rental, Payment, Insurance
Products, Services	Car, CarKey, Rental
Roles, People, Organizations	RentalCompany, Customer, Cashier
Places	OfficeAddress, CustomerAddress, CarPickupLocation, CarLeaveLocation
Records	RentalAgreement, Receipt
Events	Rental
Physical objects	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.

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.



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

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?

- 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 inexperienced 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 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.

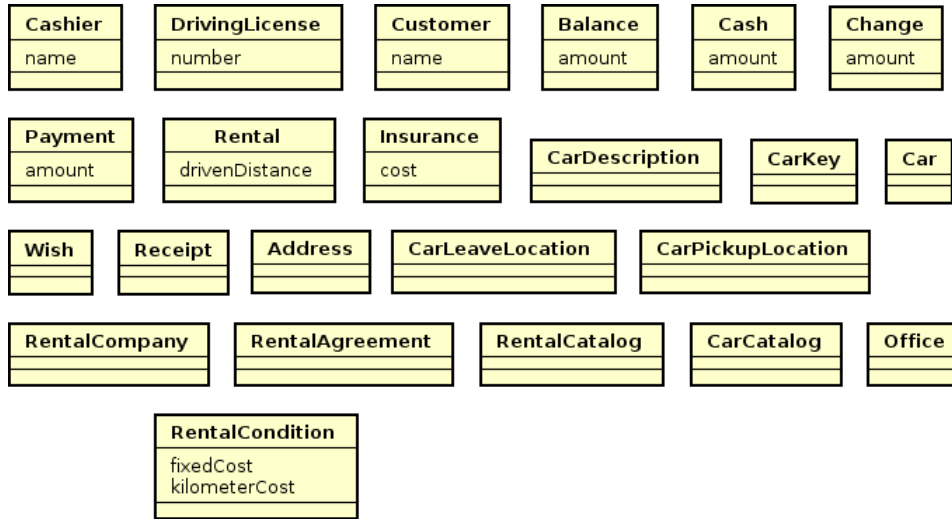


Figure 4.9 The domain model with attributes

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. Try always to name the association, without name it hardly clarifies at all. Try to avoid the names *has* and *hasA* since it 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*. 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 sentences start and end, the reader would probably try to read just *Luggage at Counter*, which does not make sense. The sequence must always be exactly class-association-class, and the association name must start with a verb.

Multiplicity is often just confusing, add multiplicity only if it clarifies the DM. Do not specify direction, trying to understand the direction of an association in the DM often leads to long and meaningless discussions. Also, 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.

Start with the most central associations. Since it is all about renting a car, that 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.

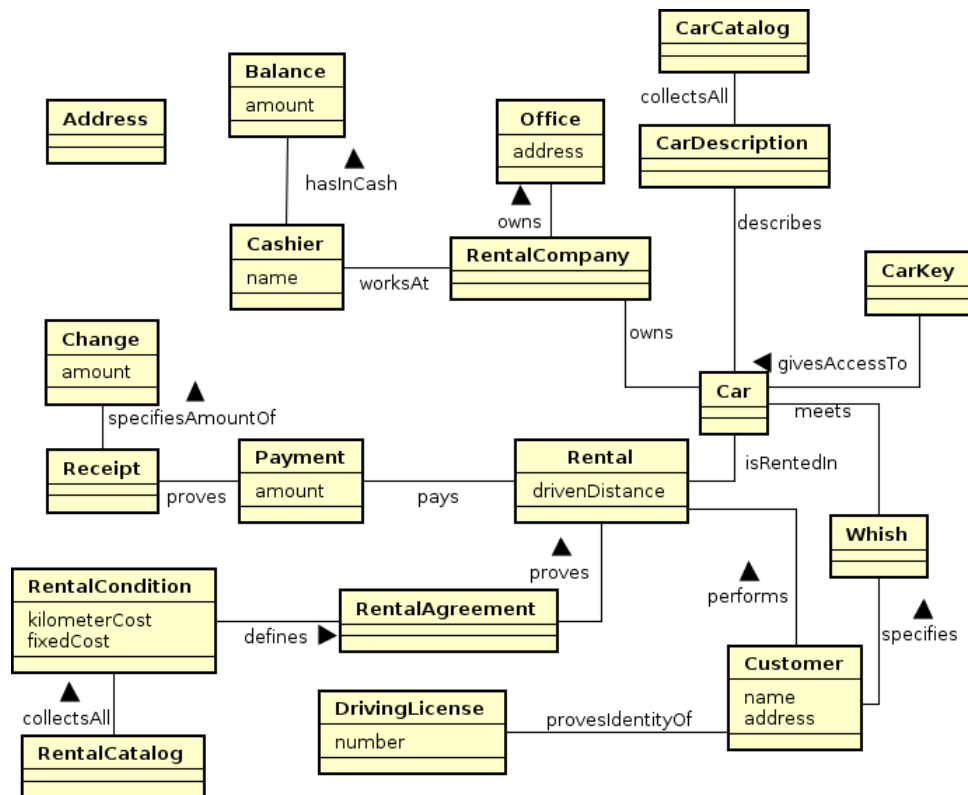


Figure 4.10 The domain model with associations

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 were 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.

## 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 the `RentalCatalog` class 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

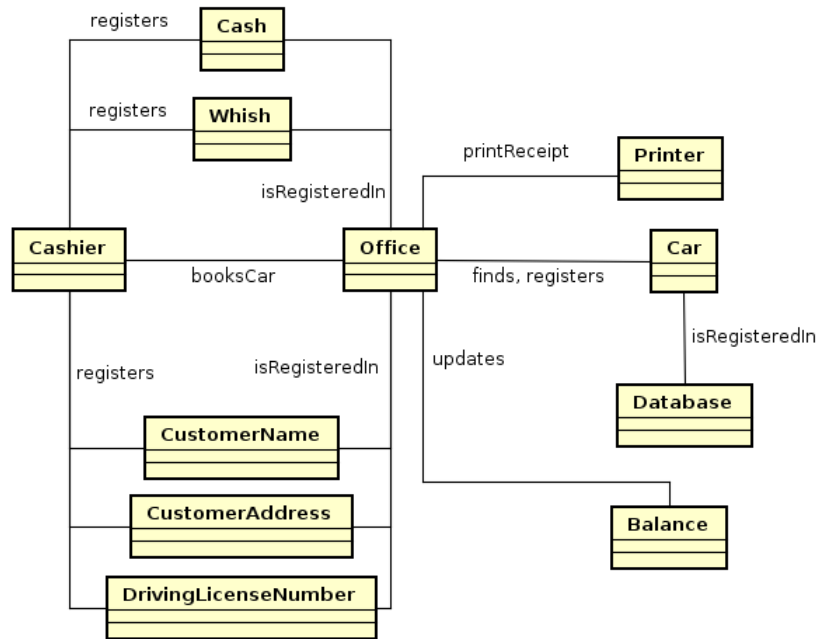
Since creating a domain model is a matter of discussion and, at least to some extent, a matter of opinion, it might be difficult to assess the quality of the resulting DM. There are many ways to create a good model, but also many ways to create a bad model. This section explains two typical mistakes,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

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, the actor, which is the cashier in the case study, becomes the user of the program. Figure 4.11 shows an example of a “programmatic domain model” where the class `Office` represents the program.

**NO!**

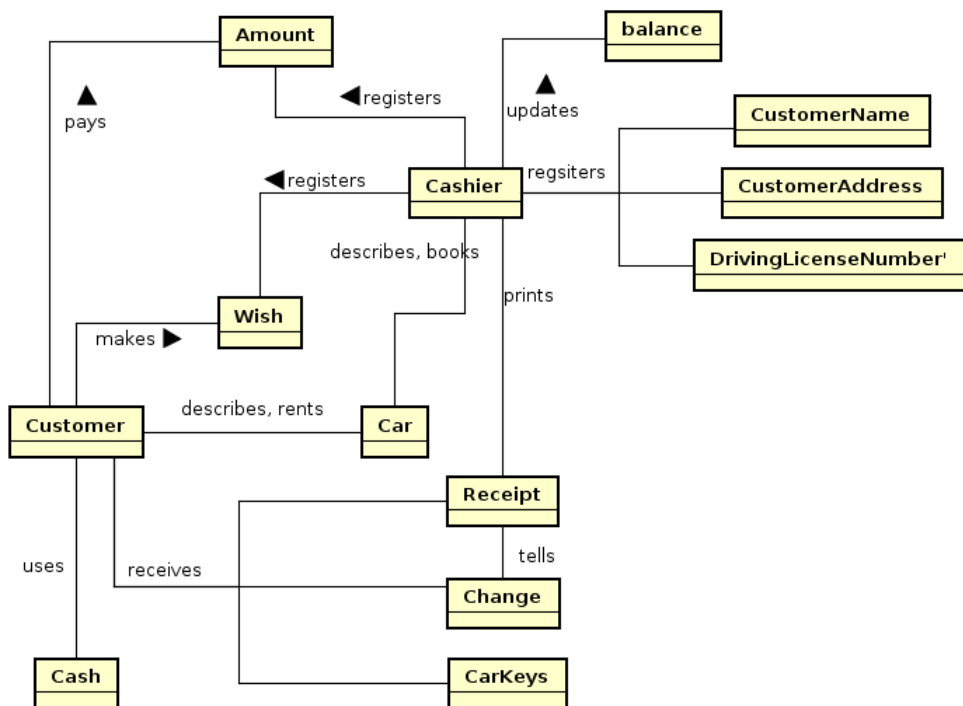
The second, and less obvious, mistake, is to create a DM that correctly models the reality, but does not convey any information besides what is already in the requirements specification. In such a “naïve domain model”, the actors, customer and cashier in the case study, become central classes with many 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, modelling flow, instead of giving a static picture of what exists. This might not be completely wrong, but adds little value to what already exists, in text. Figure 4.12 is an example of a naïve DM, compared to the DM of figure 4.10, it does not say much.

**NO!**



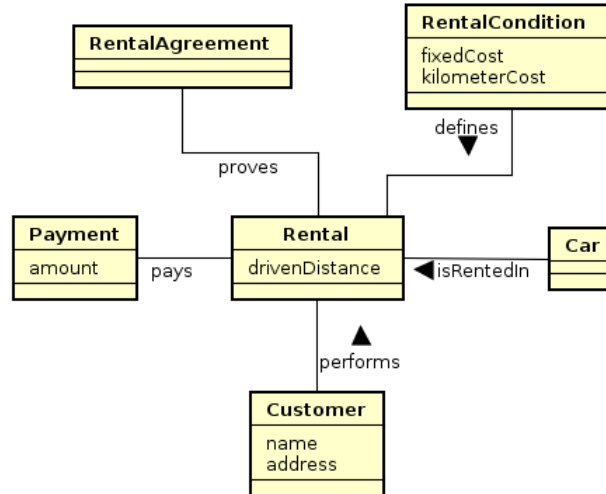
NO!

**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.



NO!

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



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

The third, less serious, mistake, is that there is 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 central class, with many associations, is often difficult to understand, since it seems to have many different roles. 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*.

## 4.4 System Sequence Diagram

The *system sequence diagram*, SSD, is a sequence diagram that shows 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 it might have to be revised, but it is not allowed to let the SSD deviate from the specification.

Since the SSD shall show the interaction between the system and its actors, therefore 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 4.14, for the sake of convenience.

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 paid 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.

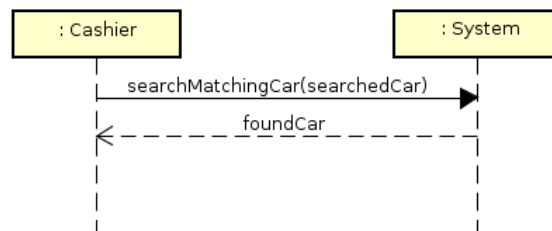
**Figure 4.14** The requirements specification for the RentCar case study.

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 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, whether primitive or classes, are not of interest in the SSD. Now, exactly which the wishes are is not necessary to decide. 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 4.15. 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.



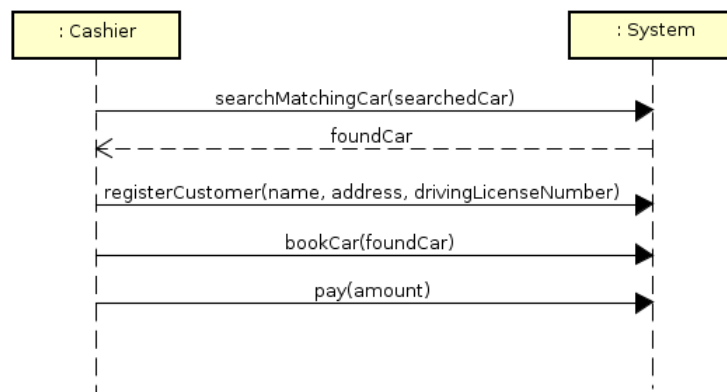
**Figure 4.15** The system sequence diagram, after the first system operation has been created.

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.

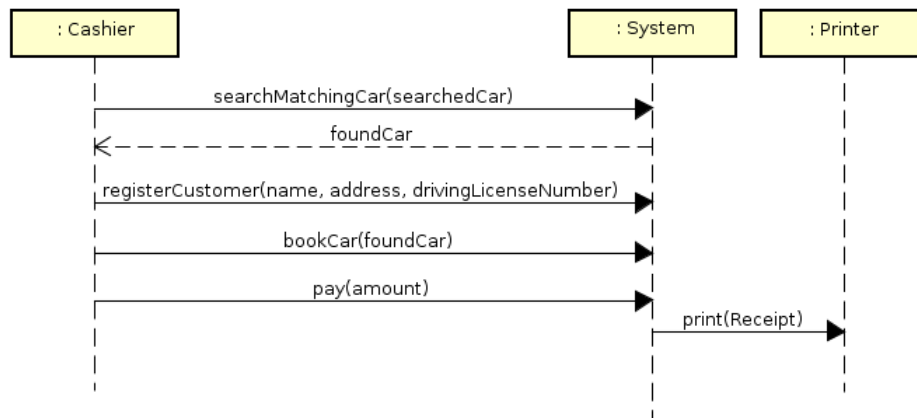
Bullet ten is no system operation. It describes work done internally, inside the system, and does not involve any interaction with an actor. Bullet eleven also is no system operation, since it takes place only between customer and cashier. Bullet twelve is a system operation, it can be called simply `pay`, and take the parameter `amount`. The SSD now looks as in figure 4.16.



**Figure 4.16** The system sequence diagram, with more system operations added.

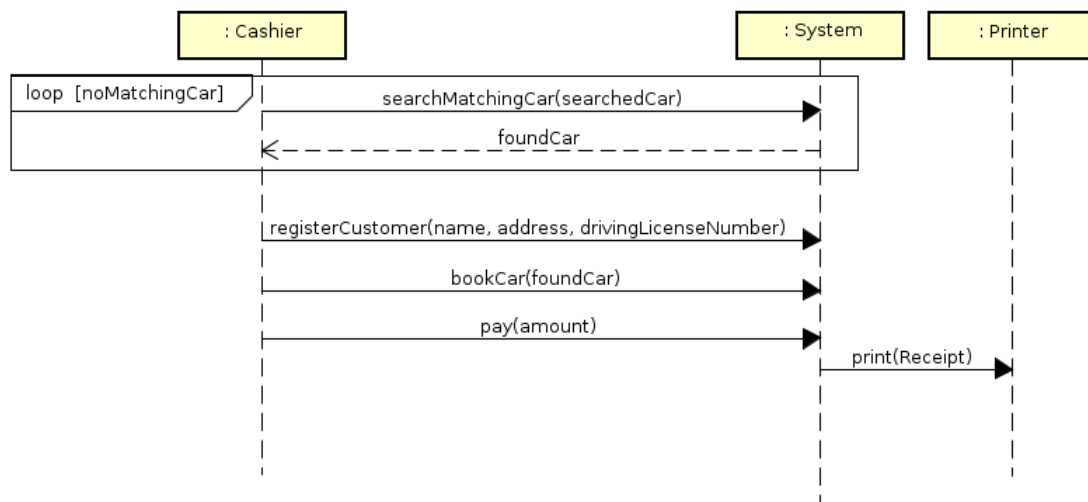
In bullet 13, a receipt is printed as a result of the `pay` operation. Does this mean `receipt` is a return value of `pay`? The answer depends on if the printer is considered to be part of the system under development (SUT), or not. If the printer *is* part of the system, the printing is an internal matter and shall not be included in the SSD. The receipt then becomes a return value. On the other hand, if the printer *is not* part of the SUT, it becomes an external system, called from the SUT. This means there is an interaction between the SUT and an external entity, which shall be included in the receipt. The latter alternative is chosen, mainly to illustrate how such an interaction looks in the SSD, see figure 4.17.

Continuing, bullets 14-16 are either internal or external, and do not bring any interaction between the SUT 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



**Figure 4.17** The system sequence diagram, with call to external system.

changed in coming iterations of the development. The loop can be modeled as in figure 4.18.



**Figure 4.18** The system sequence diagram, with a loop reflecting the alternative flow.

There are 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 4.18, 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 in the SSD.

## 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.
- To draw activation bars is not wrong, but it is discouraged since it tends to confuse, rather than clarify.

NO!

# Chapter 5

## Design

The purpose of design is to 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. 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 originally creating the program, shall be able to understand and maintain it, without rewriting anything or destroying the program structure.

The result of the design is a plan in the form of UML diagrams, illustrating the details of the program. Before those can be created, this chapter covers the necessary UML. 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 1.2 and 1.3 before reading this chapter.**



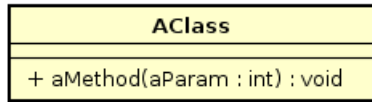
### 5.1 Further Reading

The topics of this chapter are covered mainly in chapters 13, 15, 17 and 18 in [LAR].

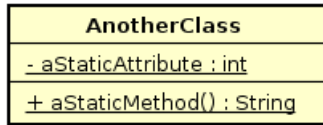
### 5.2 UML

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 were introduced in chapter 4.2, are covered.

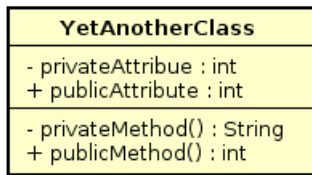
## Class Diagram



(a)



(b)



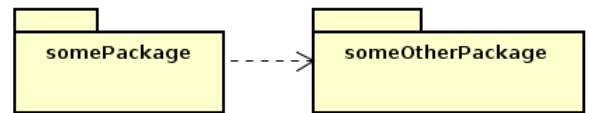
(c)

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

To create a design class diagram, some features are needed that have not been used previously in this text, namely methods, visibility and types. Methods are declared in the lowest compartment of the class symbol. A method parameter's type is written after the parameter, separated from the parameter by a colon. The methods return type is written the same way, but after the entire method, 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 an object's level of access to that member. 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 say anything about the extent or type of this dependency.

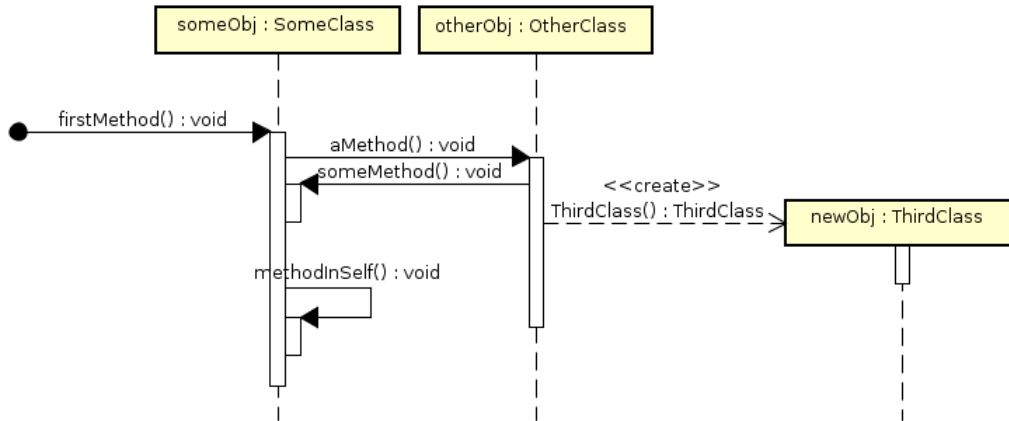


**Figure 5.2** A package diagram

## 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, types are depicted as in a class diagram, following the parameter or method, separated by a colon.



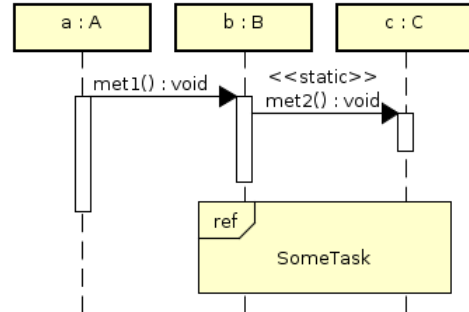
**Figure 5.3** Found message, types, activation bars, message to caller, constructor

Third, note the use of activation bars, which show the duration of a method. The bar begins on the first line of a method and ends when returning from the method. There can be any number of activation bars overlapped simultaneously on the same object, since execution might 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`.

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.



**Figure 5.4** Static method and interaction use



## 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 tend to become very wide.

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 call 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 are more than one messages 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.

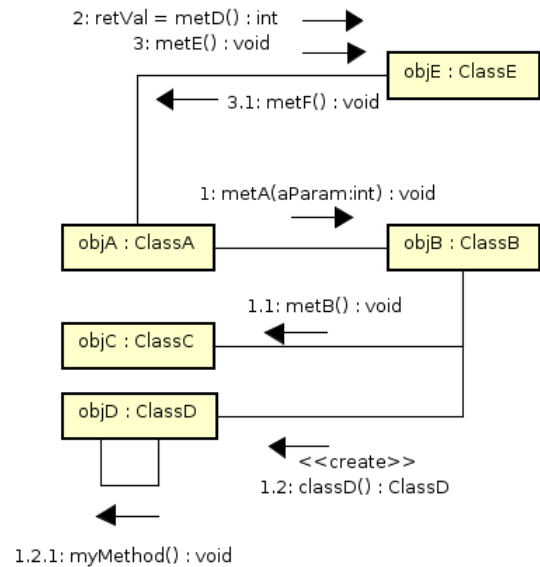


Figure 5.5 A communication diagram

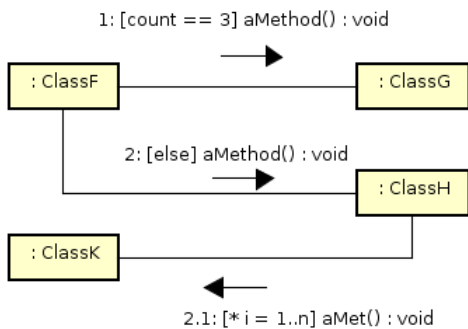


Figure 5.6 Conditional call and iteration in communication diagram

Figure 5.6 illustrates conditional calls in message 1 and 2, and iteration in message 2.1. A condition is called *guard* and is specified in square brackets. UML does not specify guard syntax, any text or program statement is allowed. The asterisk in message 2.1 indicates iteration. It 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.

## 5.3 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 first and 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 would have to understand all the internals of a clock to tell the time, it would be very bad encapsulation. Image by FreeImages.com/Colin Adamson

To understand encapsulation in software, the concept *visibility* must be clear. The visibility of a declaration states 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 use a declaration with public visibility, no matter what is declared or where it is declared. Private visibility, specified in Java

with the modifier `private`, means the declaration is visible only to code in the class in which the declaration is placed.

Encapsulation relies 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. It is possible to tell exactly which code can access a certain part of the implementation, it is not required that no other code *at all* can access it. Also, there is no “neutral” code, all code is either public interface or im-

```

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** A first example of public interface, in blue italic print, and implementation.

plementation. 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 then 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 without any risk to change other parts of the implementation, for example 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.

```

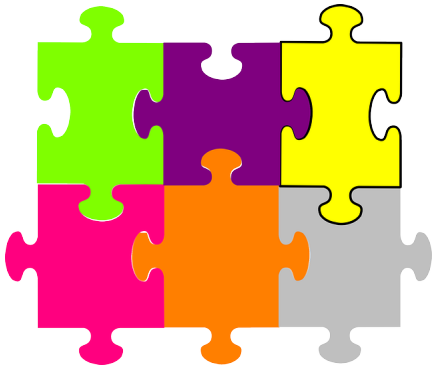
1  /**
2   * Doubles the operand and returns the result.
3   */
4  public int multiplyWithTwo(int operand) {
5      return operand * 2;
6  }
7
8  /**
9   * Doubles the operand and returns the result.
10  */
11 public int multiplyWithTwo(int operand) {
12     return operand << 1;
13 }

```

**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.

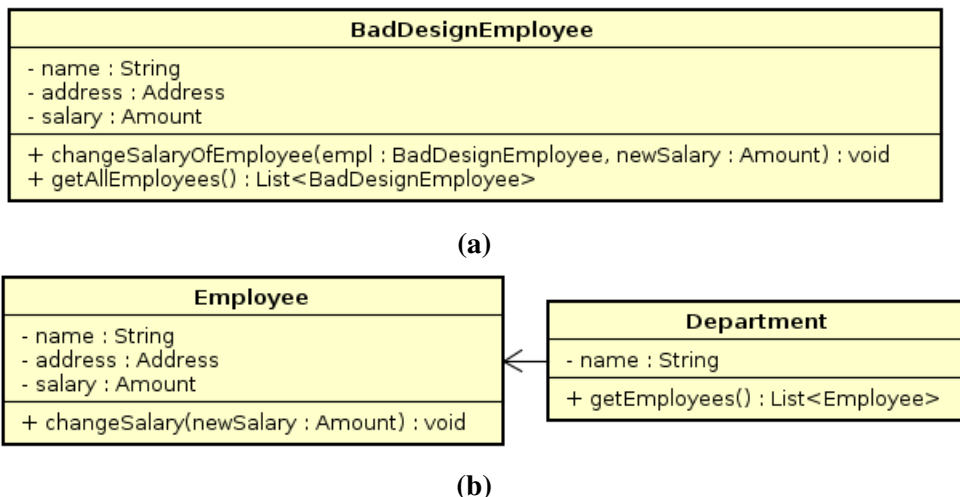
## High Cohesion



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

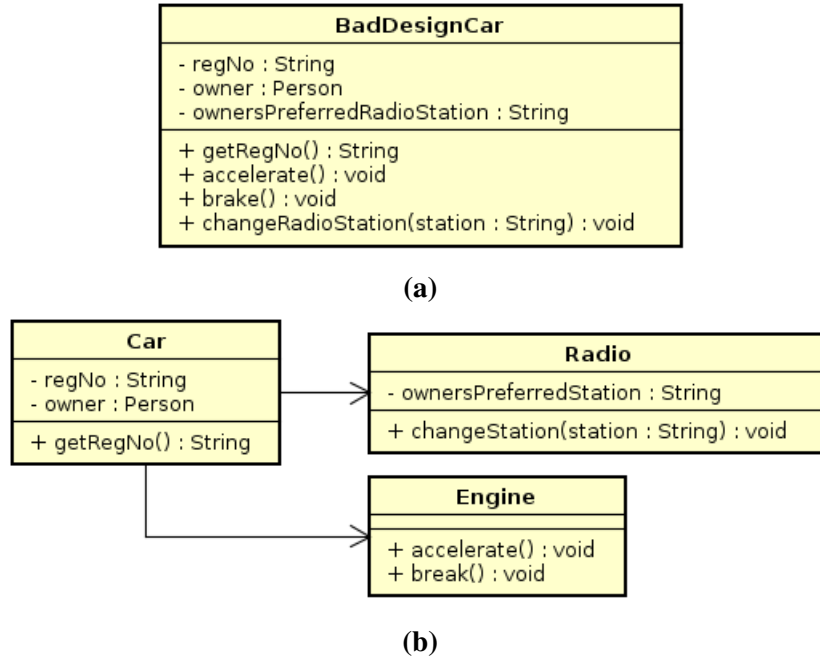
*Cohesion* is a measurement of how well defined a class' knowledge and its tasks are, and how well they fit together. The goal is that a class shall represent one single abstraction, which is clearly identified by the class name. Furthermore, the class shall have knowledge about that abstraction, not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



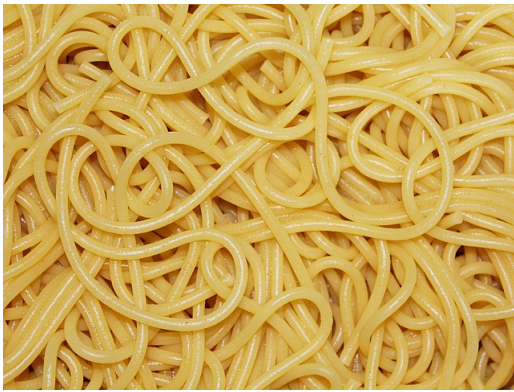
**Figure 5.10** Two different designs of the same program:  
(a) with low cohesion (b) with high cohesion

It is absolutely mandatory to always strive for high cohesion. If not, the program will be difficult to understand, and also difficult to change, since code that is not really related will be mixed together in the same class. The programmer can not relax even if 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.

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



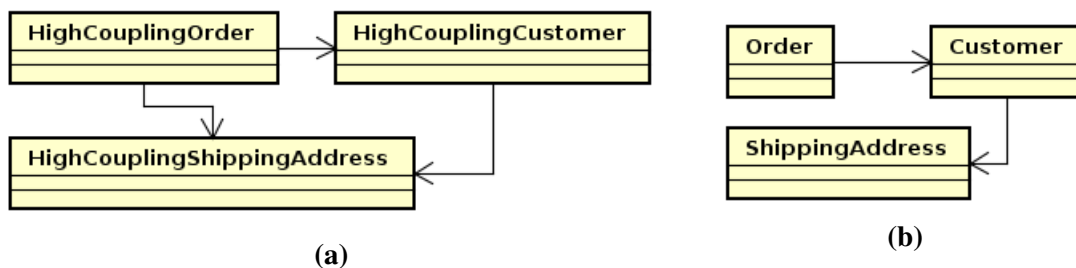
**Figure 5.11** If the UML diagram looks like a bowl of spaghetti, there is too high coupling. Image by Katrin Baustmann [Public domain], via <https://pixabay.com>

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 (dependee) depends on another class (dependee), there is a risk that the dependee 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 quite 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 size 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.

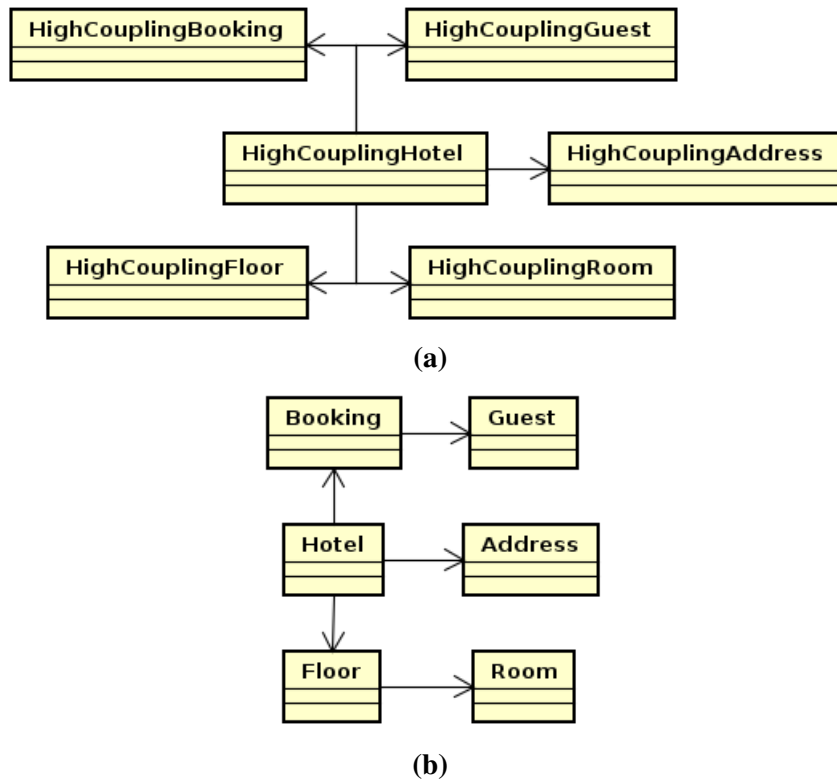
Figure 5.12 shows two different designs of the same program, one with high coupling and one with low coupling. In the version with unnecessarily high coupling, figure 5.12a, `HighCouplingOrder` has a reference to `HighCouplingShippingAddress`. This is not required since `Order` can get `ShippingAddress` 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, tends 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 `Room` in figure 5.13b. This, improved, design does not have a spider class with references to all other classes. Also, 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 since the better design does not include a spider class with high coupling.

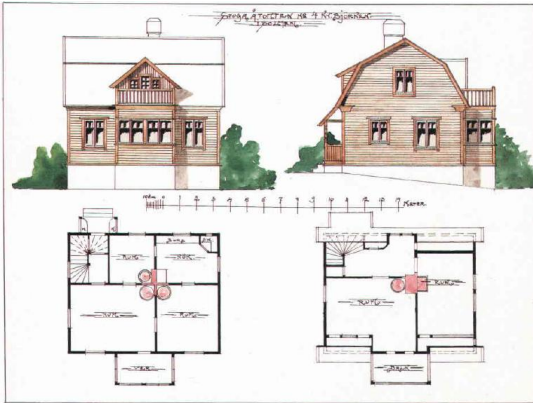


**Figure 5.13** Two different designs of the same program:  
(a) with high coupling (b) with low coupling

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.



## 5.4 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

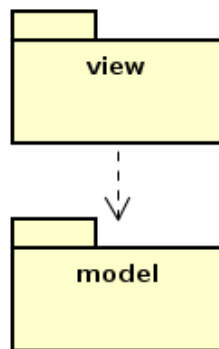
PackPriv
~ packagePrivateAttribute : int
~ packagePrivateMethod() : void

**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

The architectural pattern *MVC (Model-View-Controller)* 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 to mix user interface code and business logic code 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 (the actual functionality of the program) and its current state (the values of all variables). 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

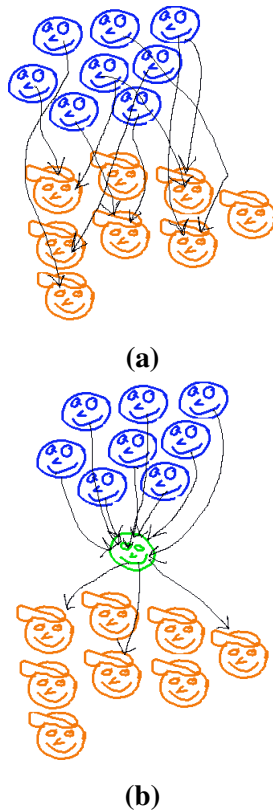


**Figure 5.16** The `view` and `model` packages.

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 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 in-

ternet 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

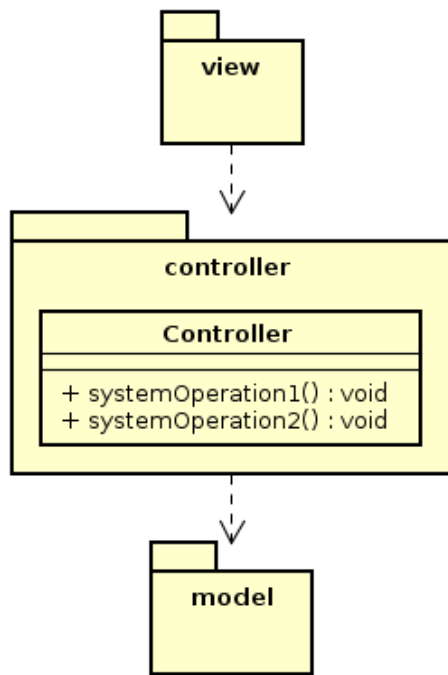


**Figure 5.17** People involved in constructing a new school. The orange persons symbolize the construction workers (carpenters, electricians, plumbers, etc). The blue persons symbolize people that will use the school (teachers, students, administrative staff, etc), and therefore give directives to the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

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.

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 the work actually performed in the model. The solution is to introduce a class (or some classes) corresponding to the steering

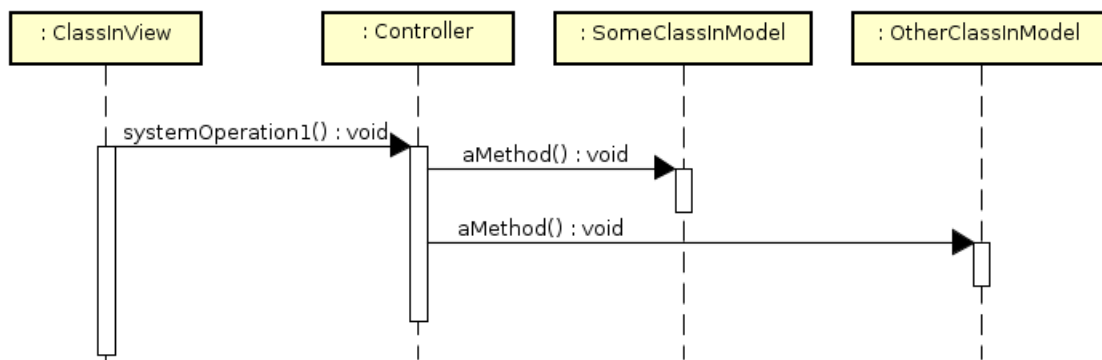
committee. Such a class is called a *controller* and is placed in the third layer of the MVC pattern, the `controller` layer. Figure 5.18 shows all three MVC layers and the `Controller` class. The controller shall contain the system operations, that is, the operations in the system sequence diagram made 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.



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

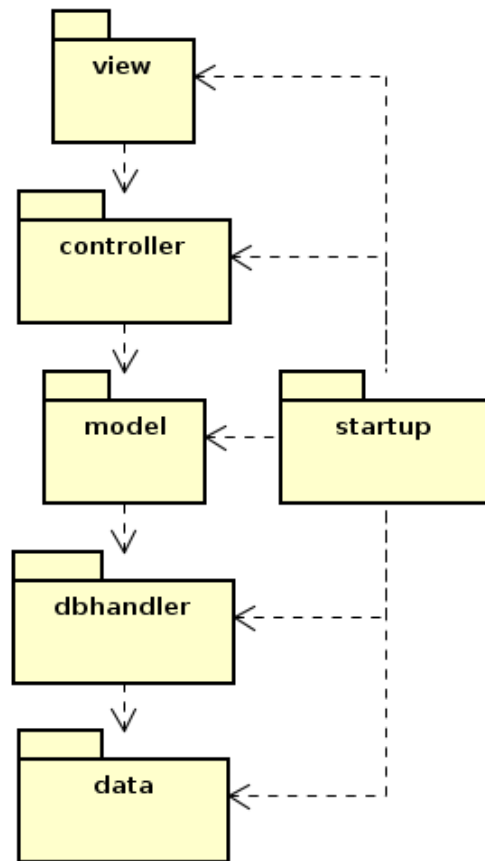
Before leaving the MVC pattern, it is worth considering interaction between the three subsystems a bit more. Regarding view and controller, a remaining question is who handles flow control between views. Suppose for example the the user interface shows a list with summary information about different items. If the user clicks an item, a new view shall be displayed, with detailed information about the clicked 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 object could be placed in the `controller` layer, but note that it is not a `Controller` class.

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 are completely independent, reducing coupling as much as possible. If so, the only way to send data from model to view, for showing 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 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 performing business logic on objects 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 of the



**Figure 5.20** Often used layers.

layers that have been 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 are depicted in figure 5.20, they 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, that is from higher (closer to the user) layers to lower (further from the user) layers. Those dependencies are unavoidable, since execution is initiated by the user, and to call lower layers there must be dependencies. 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 it is to change business rules, and yet less common than to change those rules is to remove entities represented in the database.

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 different developers or teams of 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 through 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     //Call to model
6 }
7
8 //In the model layer
9 public void registerUser(String name, String streetAddress,
```

```

10         String zipCode, String city, String country,
11         String phone, String email) {
12     //Call to dbhandler
13 }
14
15 //In the dbhandler layer
16 public void registerUser(String name, String streetAddress,
17         String zipCode, String city, String country,
18         String phone, String email) {
19     //Call to database
20 }

```

**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 bigger 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     //Call to model
34 }
35
36 //In the model layer
37 public void registerUser(UserDTO user) {
38     //Call to dbhandler
39 }
40
41 //In the dbhandler layer
42 public void registerUser(UserDTO user) {
43     //Call to database
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.

## 5.5 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 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. An educated guess that is valid for many programs is to use the layers depicted in figure 5.20. The class `Controller` can also be created already now. Illustrate the packages and `Controller` in a class diagram.
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 deciding which classes and methods there shall be, use interaction diagrams. An interaction diagram shows the flow through the program, how methods call each other. Whether to use sequence or communication diagrams is a matter of taste.

3. **Strive for encapsulation with a small, well-defined public interface, high cohesion and low coupling.** 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 if that improves the overall design.

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 with all classes.** When done designing a system operation, summarize the design in the class diagram created in bullet 1, which will contain all classes in the program. Such a diagram gives a good overview.
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. Also, postponing programming allows focusing on design without getting lost in code. 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.

## 5.6 Designing the RentCar Case Study

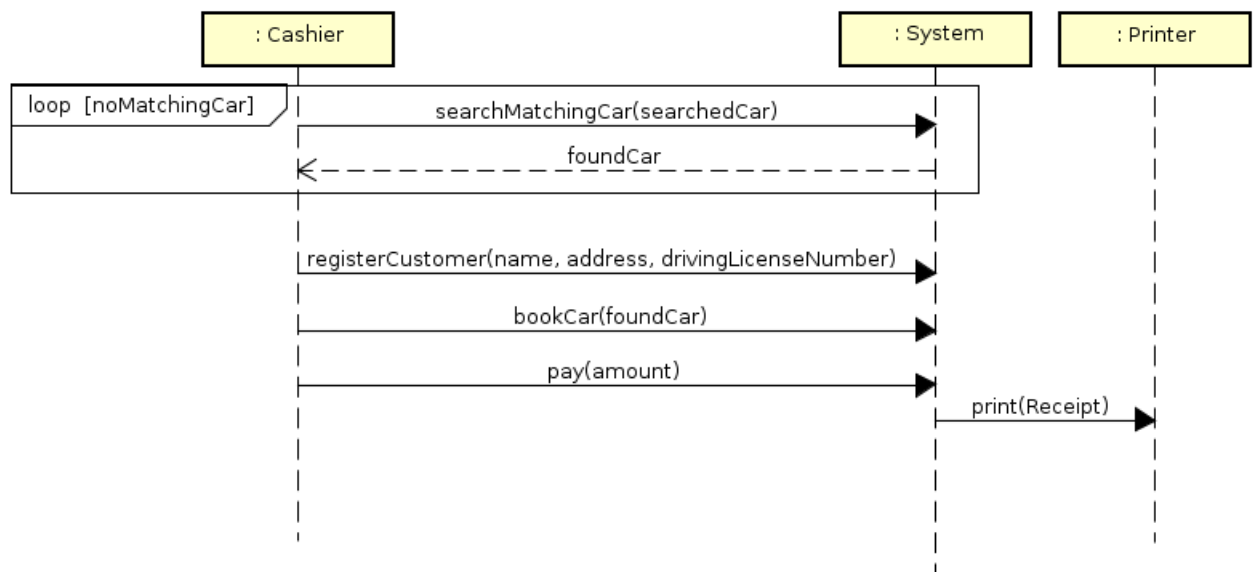
As an example, the `RentCar` case study is designed in this section. For convenience, the specification, SSD and domain model are repeated below, in figures 5.21, 5.22 and 5.23. When creating the design, be sure to have an understanding of how it may be implemented in code. If that is not clear, now is the time to repeat Java programming, for example by reading chapter 1, especially sections 1.2 and 1.3.

### Step 1, Use the Patterns MVC and Layer

Following the method described in section 5.5, 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

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 paid 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.

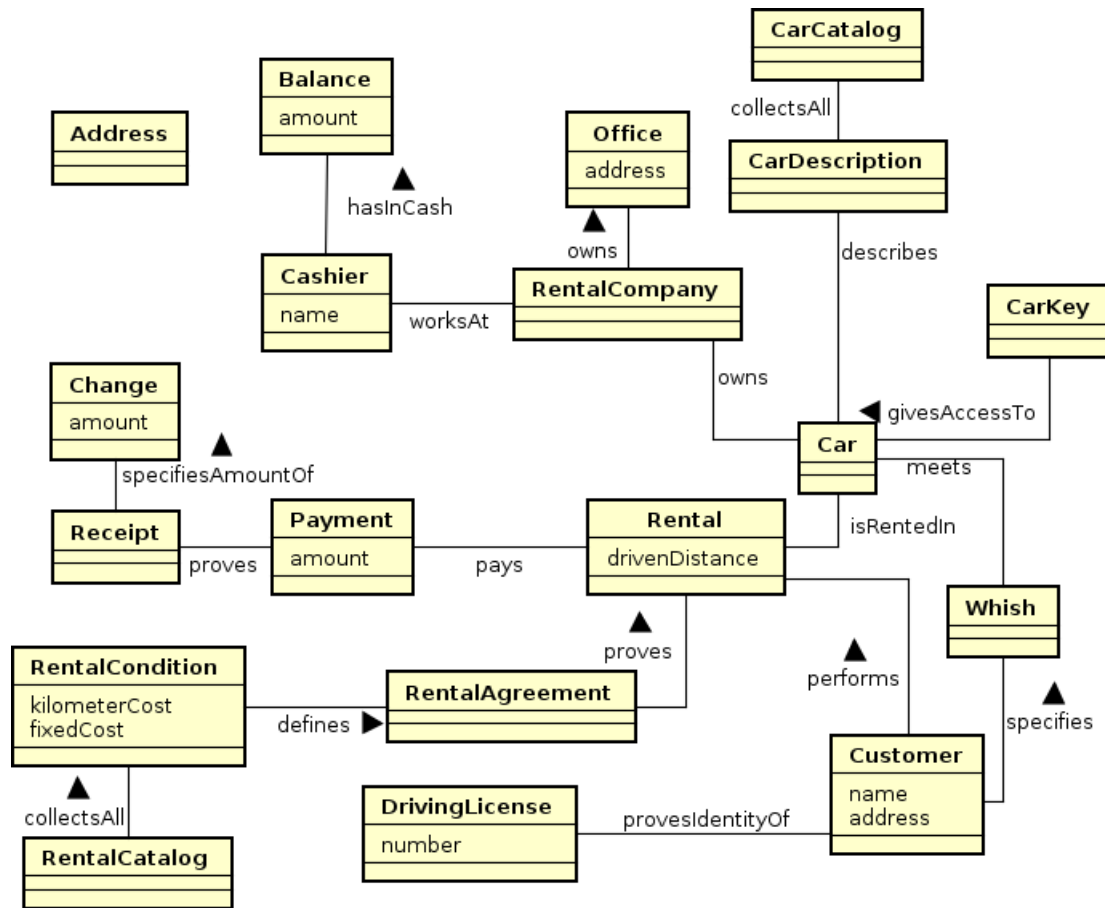


Figure 5.23 The RentCar domain model.

5.20. Therefore, after having introduced the `Controller` class, the design looks as in figure 5.24.

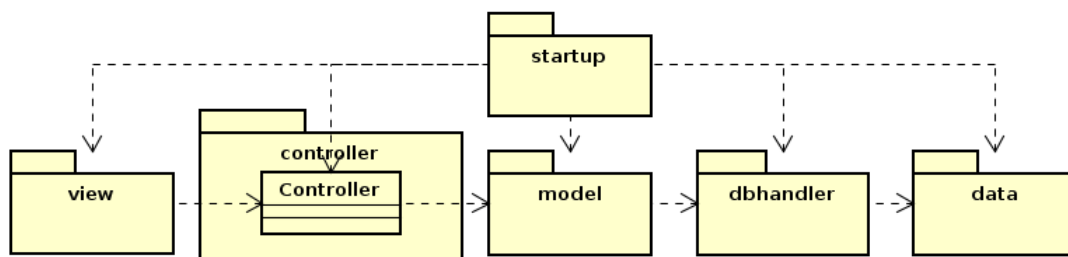


Figure 5.24 The first version of the RentCar design class diagram.

## Step 2, Design One System Operation at a Time

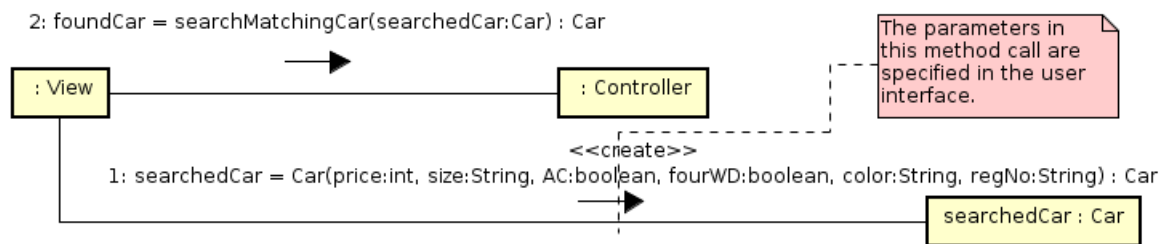
The view will not be designed here, instead there will be a single class, `View`, in the `view` package. This class is a placeholder for the real view, which certainly would consist of more than one class. This way, there is no need to bother about view technologies like window management, console IO or HTTP. Also, as far as design is concerned, 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.

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 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

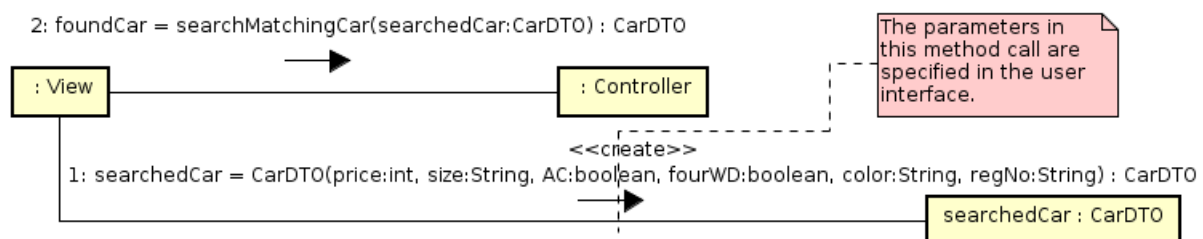
The question, *which is the type of the parameter `searchedCar` and the return value `foundCar`*, is the first of a large number of design decisions, let's consider it carefully. The answer shall be guided by the concepts encapsulation, cohesion and coupling. The purpose of `searchedCar` is to represent 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, represents 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.

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



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

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, there is no real need to introduce both a DTO and an entity, since they would be identical. Instead, the `Car` class in the model could be considered to be a DTO. 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 it is a DTO, this makes the communication diagram look as in 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 the `CarRegistry` class. Here, however, there is no database. Thus, instead of calling the database, `CarRegistry` will just look in an array of available cars. The final design of `searchMatchingCar` is in figure 5.27. Note that this diagram has no notion at all of the loop around the system operation, that is drawn in the system sequence diagram. This is because the loop condition, `noMatchingCar` is not visible in the program, but is completely decided by the cashier (after talking with the customer). Exactly the same flow, depicted in figure 5.27, is executed again and again, until the customer is satisfied.

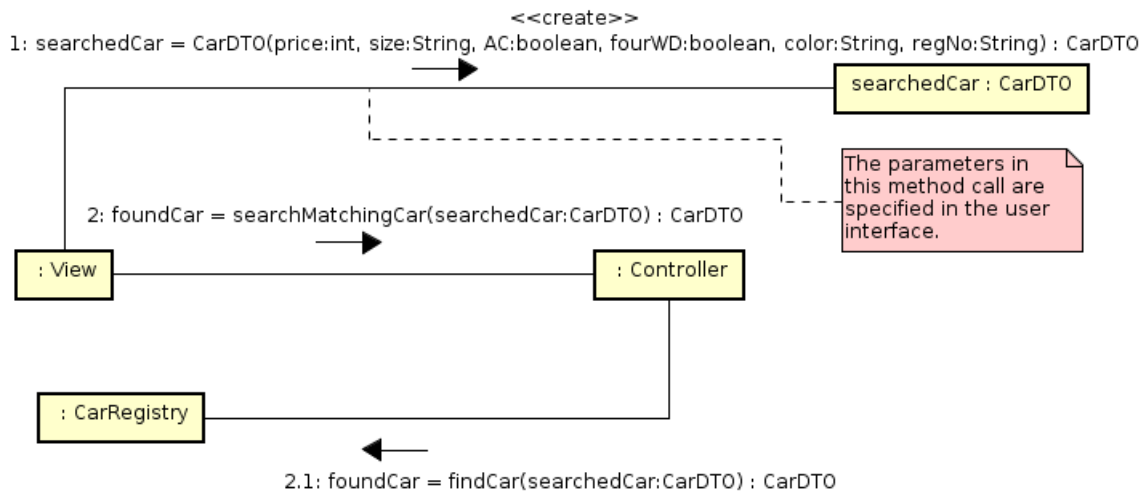


Figure 5.27 Call to `dbhandler` layer is added.

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

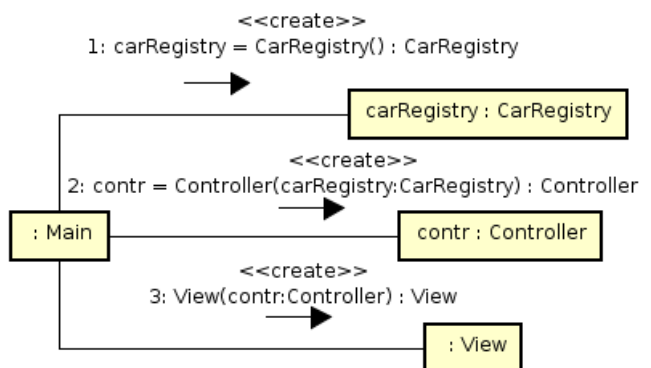
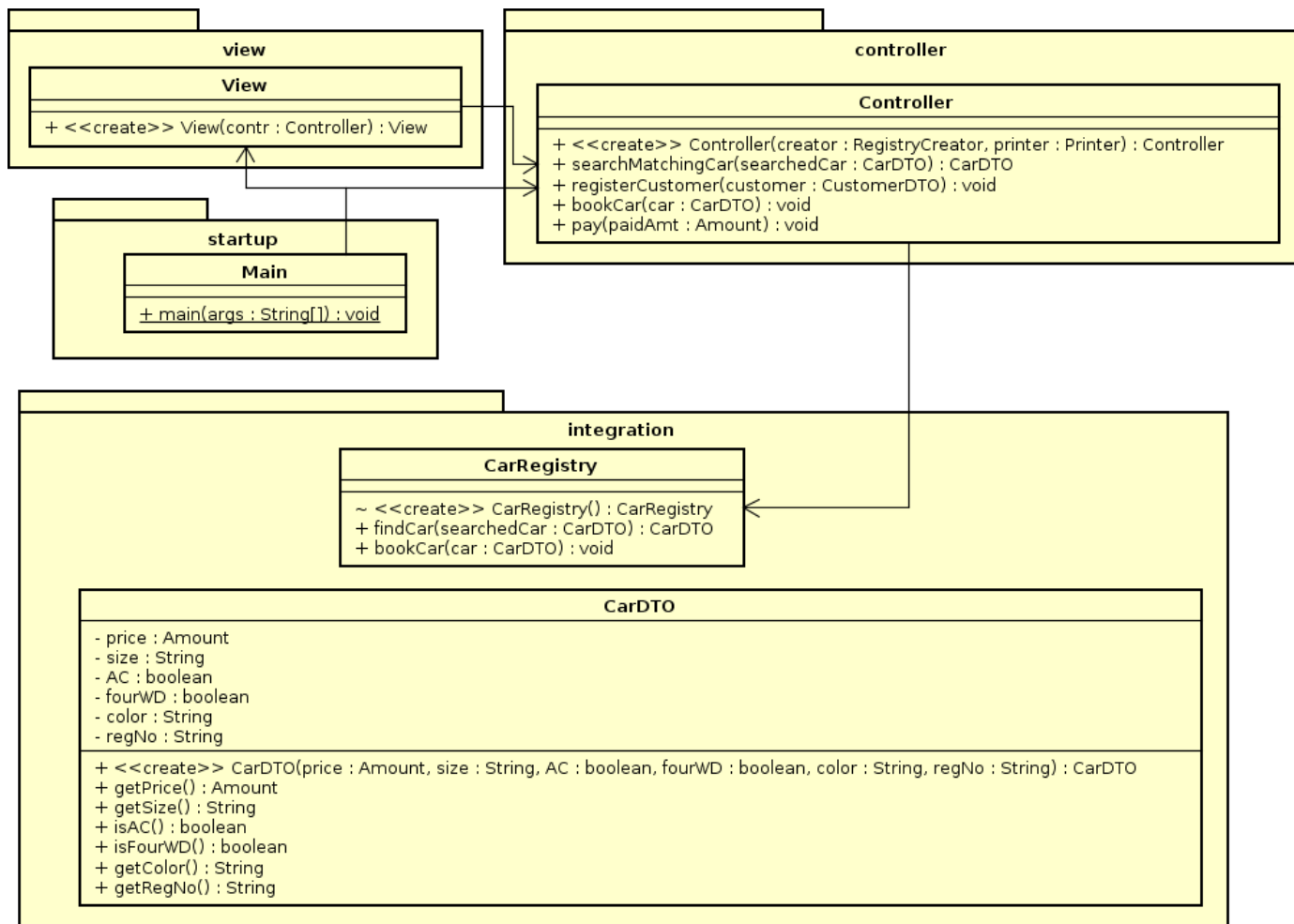


Figure 5.28 The start sequence in the `main` method.

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`. Let's choose the former alternative, `main` creates all three objects, see figure 5.28. The other 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`.

#### 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.

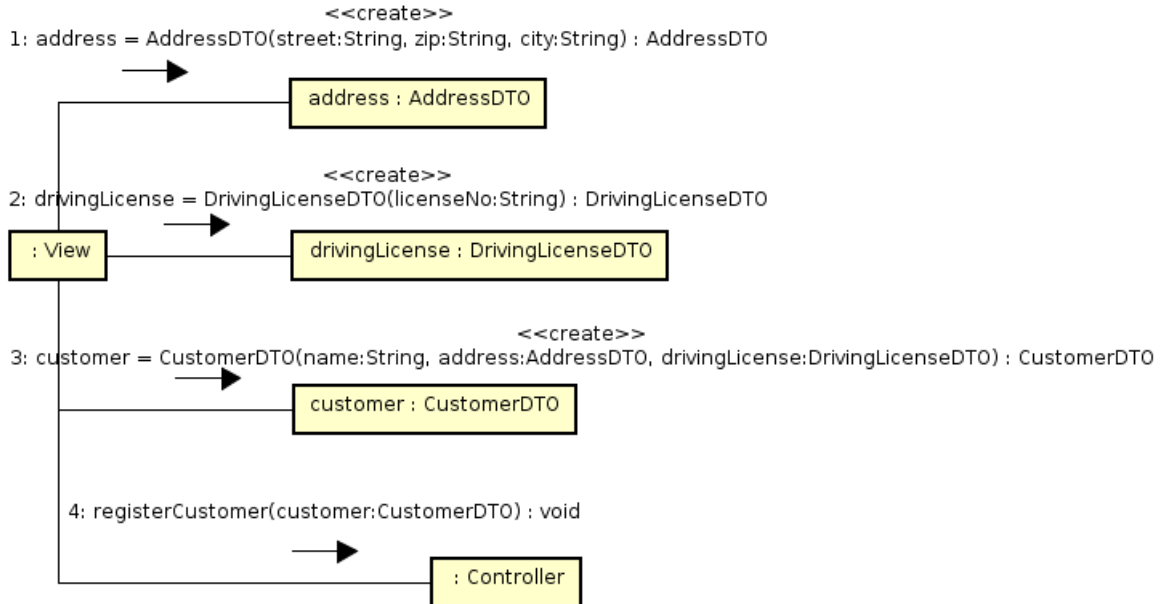
That concludes the design of the `searchMatchingCar` system operation. The question naturally arises, whether all this designing is really necessary just to fetch an element from an array? The answer is *yes*, most definitely *yes*. A professional programmer should, and normally does, make this kind 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.

## 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 or is there no difference between the two? 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.



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

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.

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.

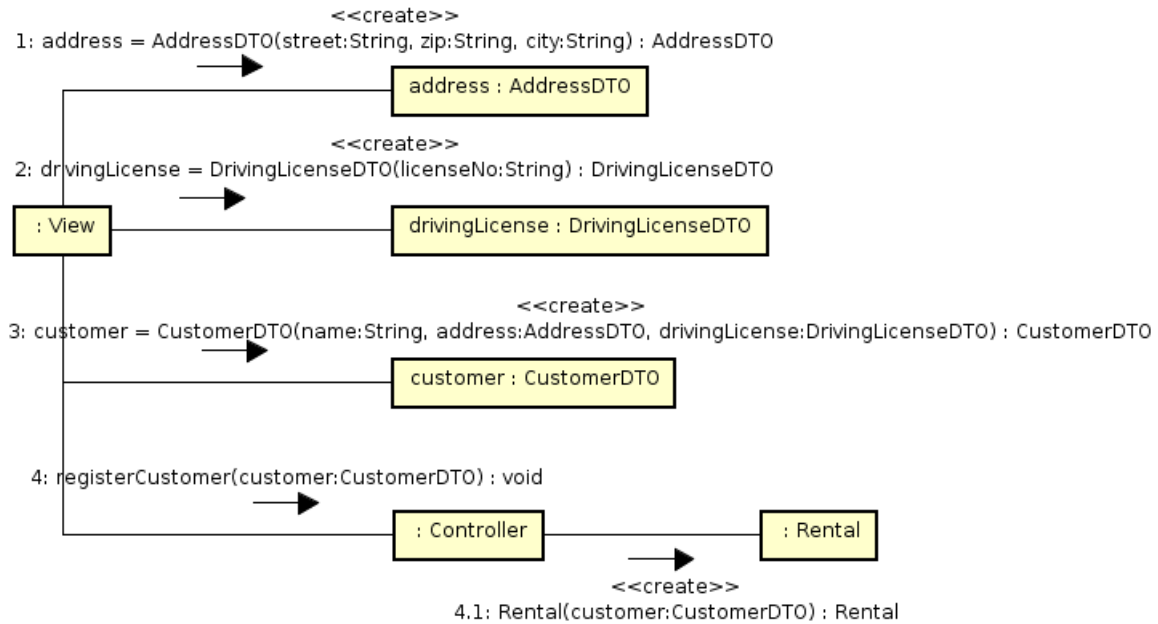


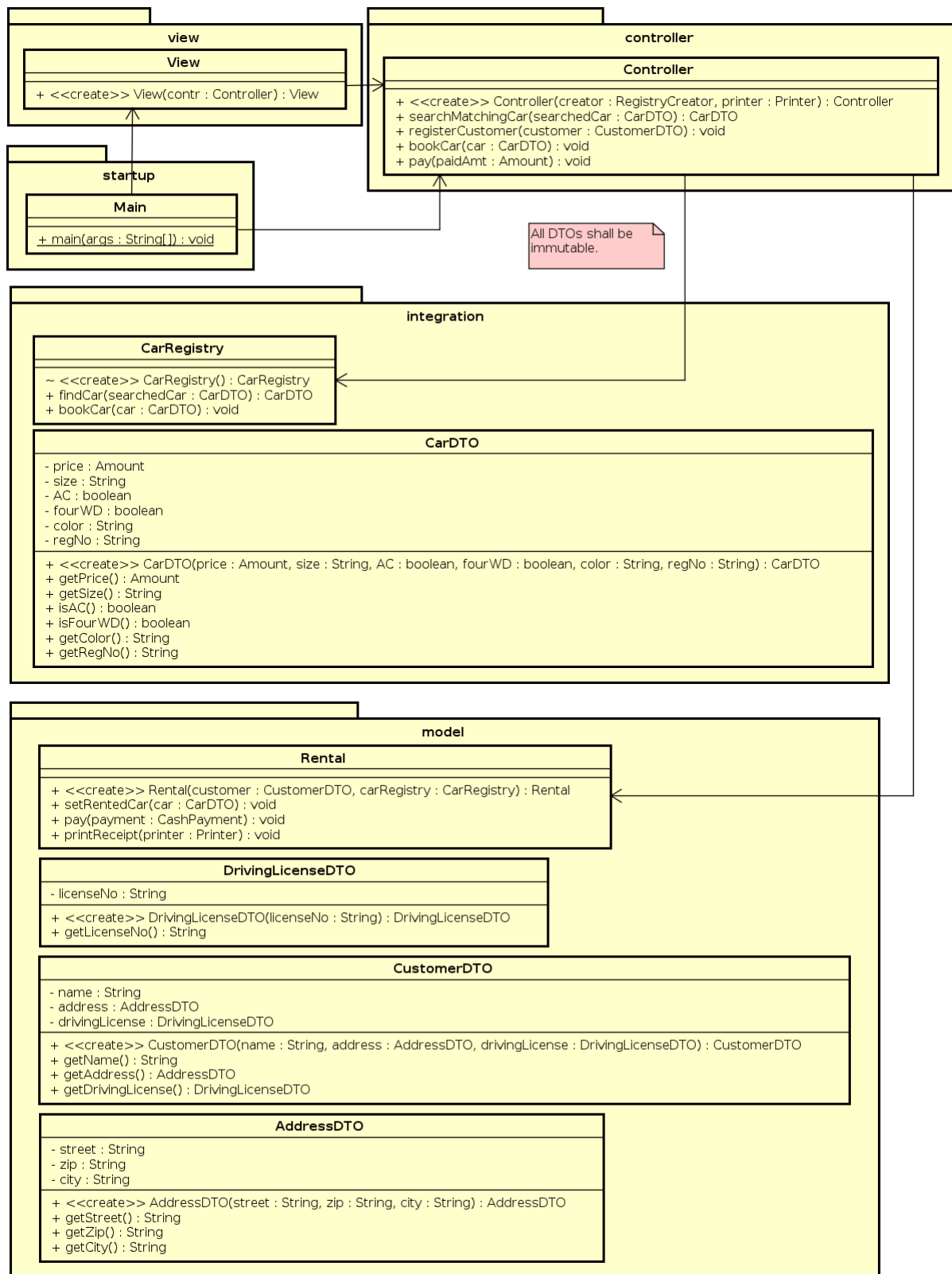
Figure 5.31 The `Rental` class symbolizes the entire rental transaction.

## 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.

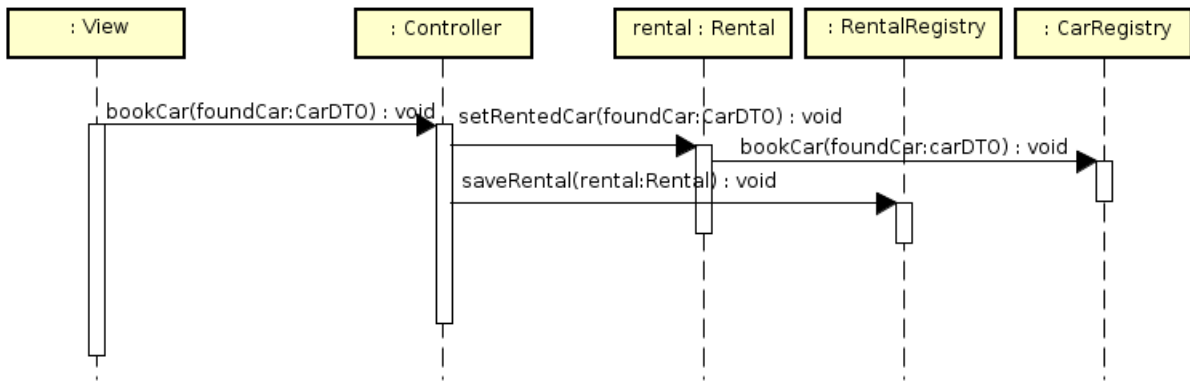
The name of the parameter that specifies the car in the SSD is `foundCar`, which is the same name as the variable that was returned from the `searchMatchingCar` system operation. This indicates that these two are the same object. This information will be kept in the design, by using the same name, `foundCar`,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

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



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

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 too 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.



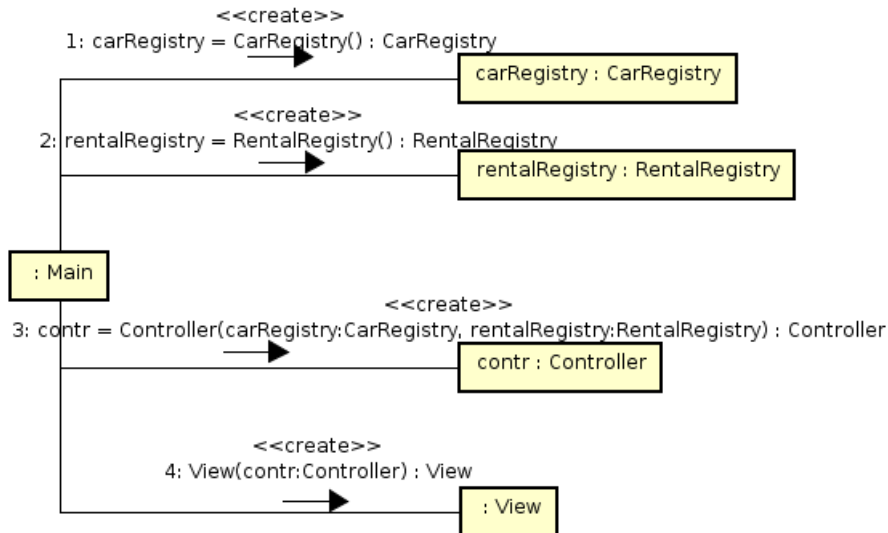
**Figure 5.33** The `bookCar` design interaction diagram.

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.

It is a quite interesting decision to let `Rental`, instead of `Controller` call `bookCar`. 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, 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, and thereby hides their existence to `main`. This solution will be discussed and improved further in chapter 9. For now, we conclude that 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.

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.

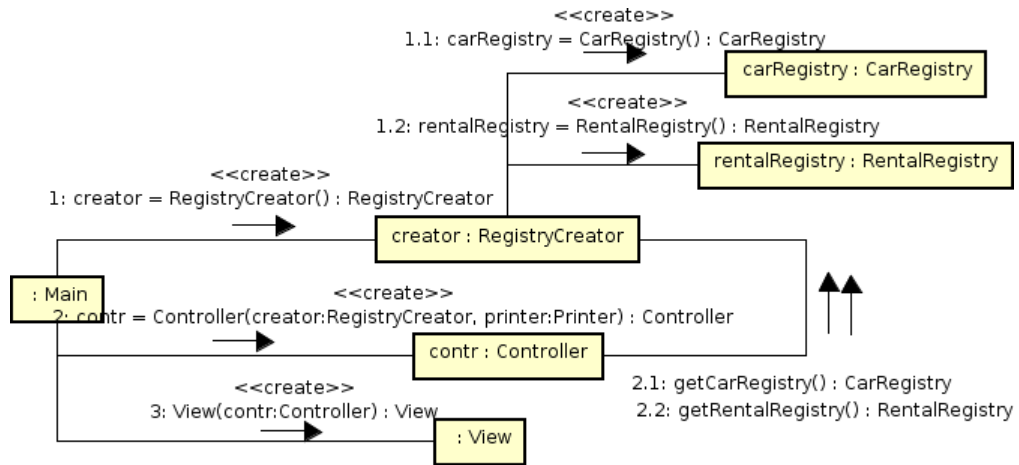


Figure 5.35 The start sequence when RegistryCreator is added.

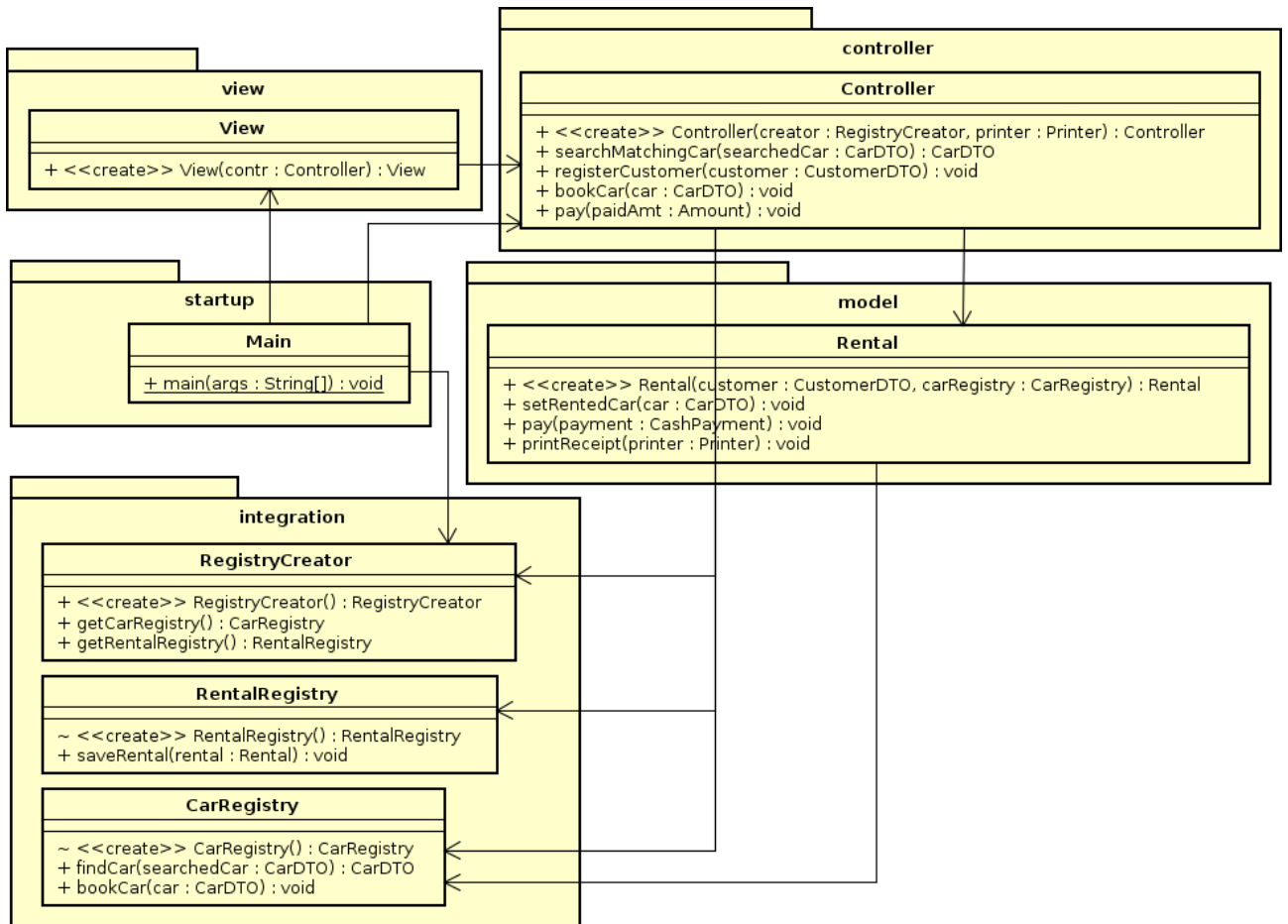


Figure 5.36 RentCar design class diagram after designing bookCar.

## 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: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.

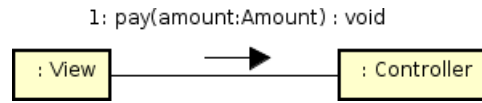


Figure 5.37 The `pay` system operation.

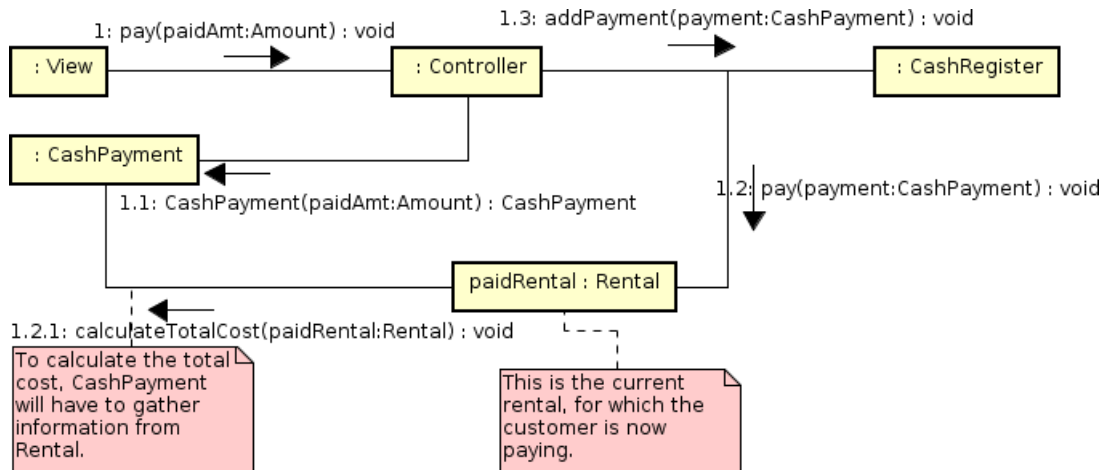


Figure 5.38 Payment and `CashRegister` handling the `pay` system operation.

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.

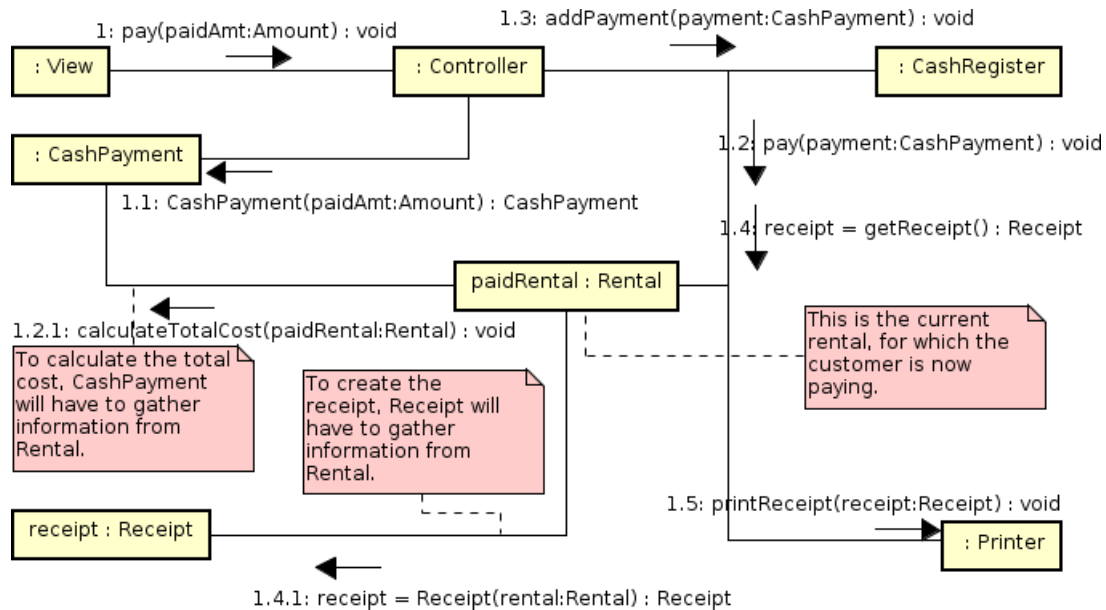


Figure 5.39 Receipt printout is added.

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 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 `RegistryCreator` (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.

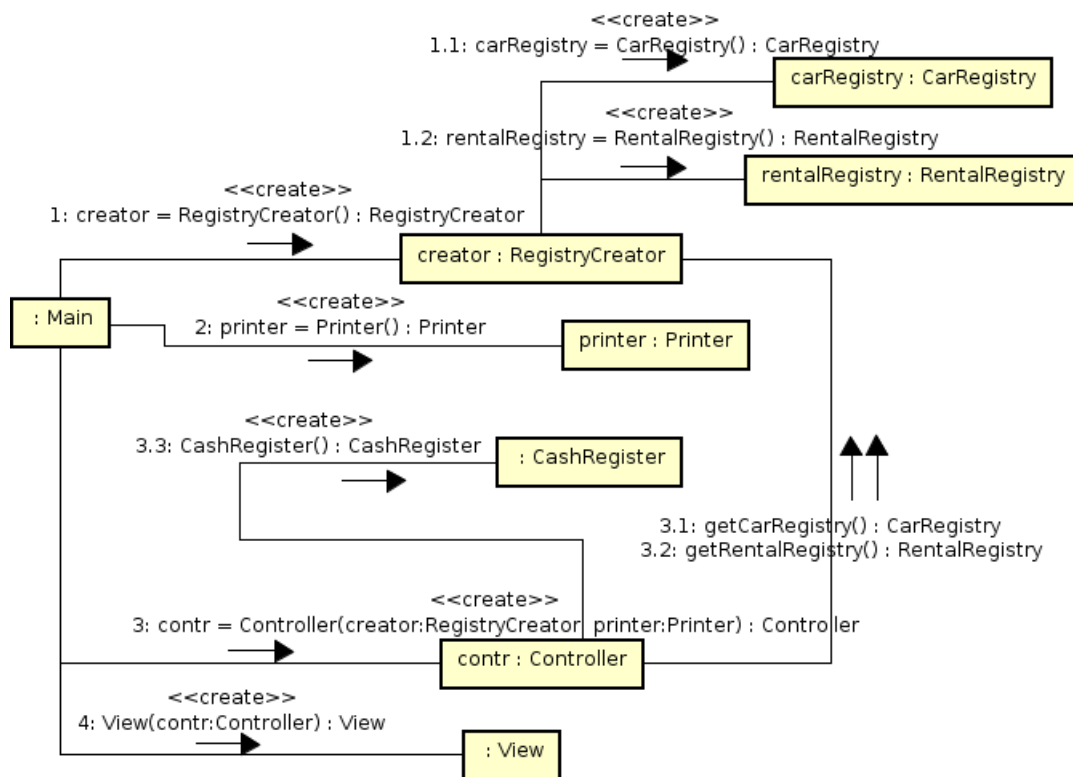


Figure 5.40 The complete startup design.

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.

The last thing to do is to draw the 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.

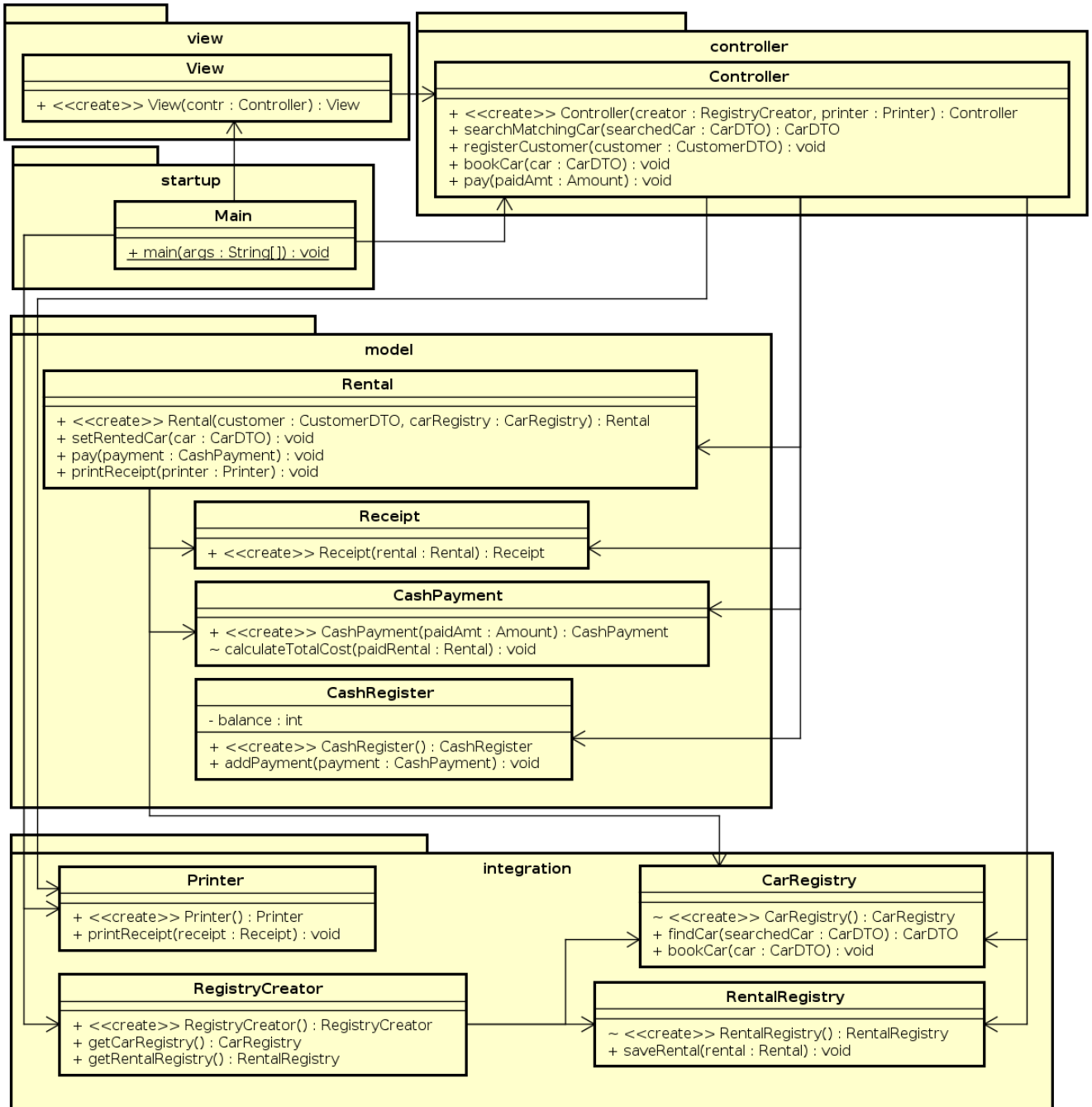
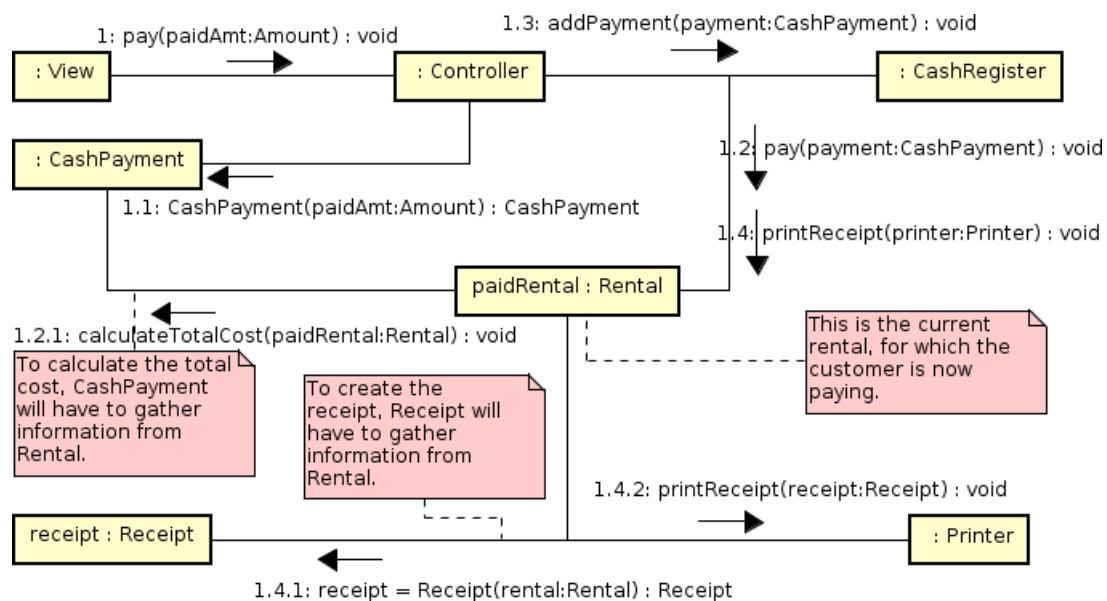


Figure 5.41 The class diagram after `pay` is designed.

## 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 is not the desired result. Also, there is quite high coupling. For example `Controller` is associated with all classes in `model` and most classes in `integration`. This situation would be improved if `model` classes called each other instead of passing through `controller`. One thing that can be done is to change the `Rental` method `Receipt getReceipt()` to `void printReceipt(Printer printer)`. This moves the printer call from `Controller` to `Rental`, removes the association from `Controller` to `Receipt` and makes the `Printer` constructor package private. This gives the pay design in figure 5.42 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.

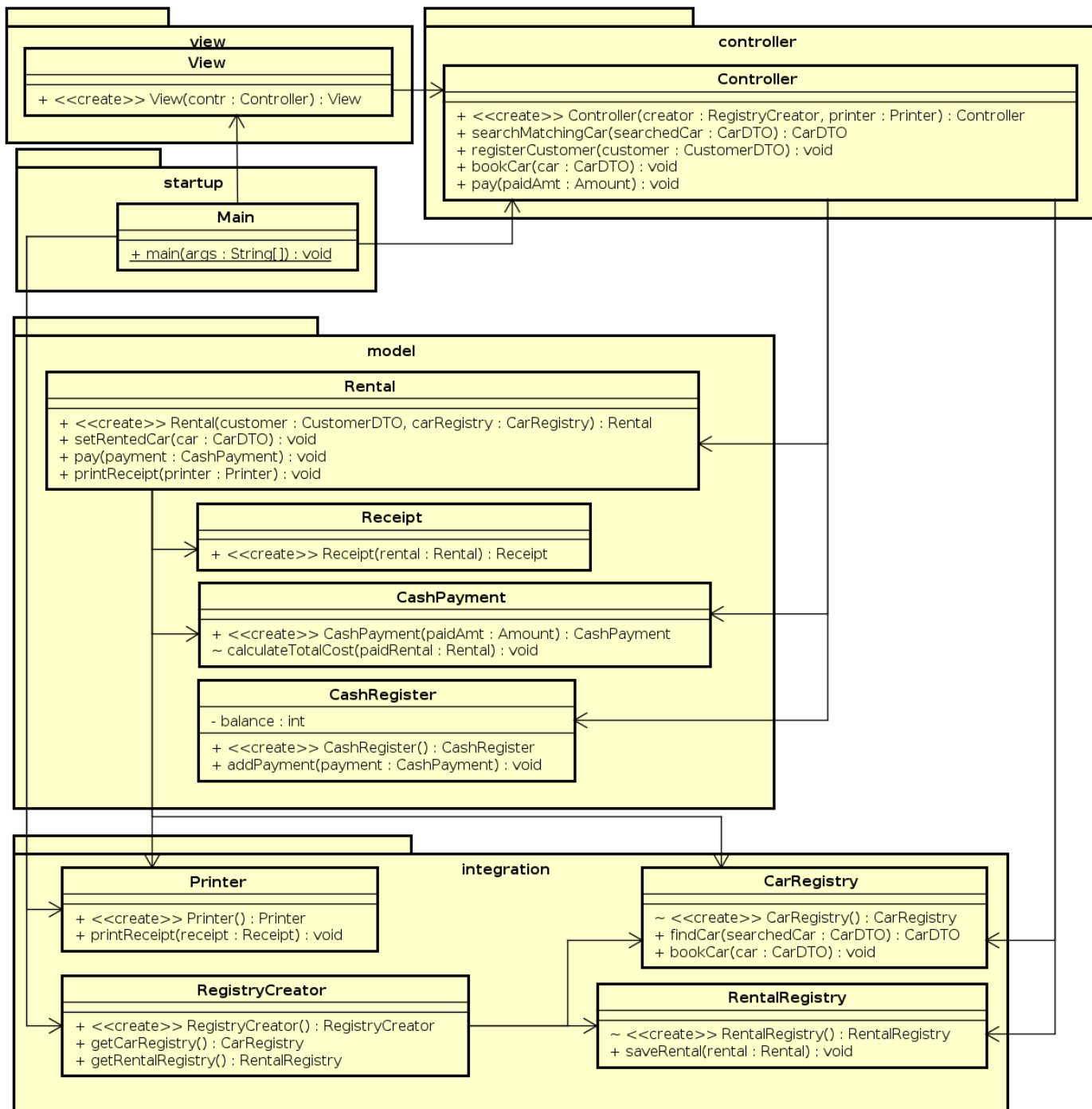


Figure 5.43 The final design class diagram.

## 5.7 Common Mistakes

Below follows a list of common design mistakes.

- 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.
- 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 attributes in a class. Not using objects means that the prime tool (objects) to handle encapsulation, cohesion and coupling is thrown away.
- There are unwarranted static methods or fields. It is not forbidden to use static members, but there must be a very good reason why there are such. Static members do not belong to any object, and are therefore, just as is the case for primitive members, using them means that the prime tool (objects) to handle encapsulation, cohesion and coupling is thrown away.
- There are 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.
- Too few layers 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.
- There can also be 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 in any method. If so, the design is not complete and a new method must be introduced, performing the missing task.
- The MVC pattern might be used the wrong way. Under no circumstance must there be any form of input or output outside the view.
- Also the layer pattern might be used the wrong way. All layers must have high cohesion and there should be calls only from higher (closer to the user) layers to lower layers.

**NO!**

- Data appears out of nothing. Always consider if it is possible to implement the design in code. That is not possible if a certain variable is passed in a method call, but the variable does not exist in the calling method.
- The class diagram is too big and is 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 members and/or 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, this can be used when programming.

**NO!**

# Chapter 6

## Programming

This chapter describes how to implement the design in code, and how to test the code. Coding is never a straightforward translation of the design diagrams. Most likely there are coding details that were not considered during design. It is also quite likely to discover actual design mistakes. This means there is no sharp line between the 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.

As soon as a certain functionality is implemented in code, it should be tested. This chapter covers fundamentals of unit testing. This includes guidelines about what, when and how to test, and an introduction to technologies that facilitate testing.

### 6.1 Further Reading

### 6.2 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, e.g., `se.kth`. This is to avoid name conflicts with packages created by other organizations. Following that, there are normally components that uniquely identifies the product within the organization, e.g., department and/or project names, `iv1201.carRental`. Finally, there are the components that identify a particular package within the product. This part could start with layer name, e.g., `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.iv1201.carRental.model.payment`

Sometimes, a class does not clearly belong to a specific package, but is needed in many different packages. This behavior of a class gives important information, often that the class does not really fit in the layer structure of the program. This might be because of a design mistake, but another common reason is 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 that 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.iv1201.carRental.util`.

## 6.3 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	<code>se.kth.iv1201.rentCar.model</code> or <code>se.kth.iv1201.rentcar.model</code>
class and interface	Full description, first letter of each word uppercase	<code>CashRegister</code>
method	Full description, first letter lowercase, first letter of non-initial words uppercase.	<code>calculateTotalCost</code>
variable, field, parameter	Full description, first letter lowercase, first letter of non-initial words uppercase.	<code>paidRental</code>
constant	This applies to <code>final static</code> fields. Only uppercase letters, words separated by underscores.	<code>MILES_PER_KM</code>

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



## 6.4 Comments

There should be one javadoc comment for each declaration that belongs 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 and Eclipse) provide a graphical user interface to the `javadoc` command.

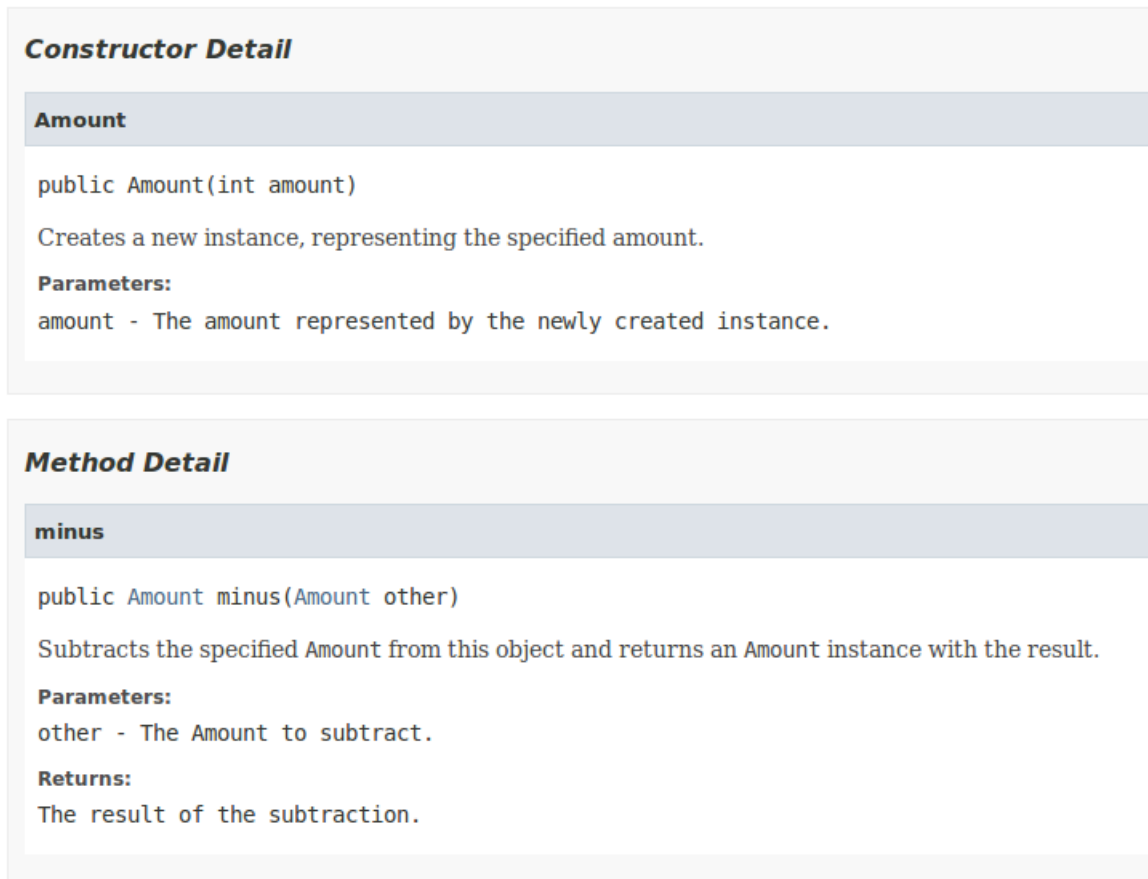
The javadoc comment shall describe what the commented unit does, but not how it is done. *How* belongs to the implementation, not to the public interface. Method comments shall explain not only what the method does, but also its parameters and return value. These are commented using the `@param` and `@return` javadoc tags. The `<code>` tag shall be used for Java keywords, names and code samples. This is illustrated in listing 6.1. The `@param` tag is used on lines 11 and 23, the `@return` tag on line 24 and the `<code>` tag on lines 19, 20 and 23. 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 Amount from
20      * this object and returns an Amount
21      * instance with the result.
22      *
23      * @param other The Amount 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.



**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.

## 6.5 Code Smell and Refactoring

The concept *code smell* describes the state of a particular piece of code. It originates from [FOW], which, despite its age, is still very relevant. 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 normally introduced by copying previously written code. You should hear a loud warning bell ring if you ever type `ctrl-c ctrl-v` when 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 /**

```

```

4  * This class does not contain duplicated code. The
5  * previously duplicated code has been extracted to
6  * the method <code>printNames</code>
7  */
8  public class ClassWithoutDuplicatedCode {
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         printNames();
18         //some code.
19     }
20
21     /**
22      * To perform its task, this method has to print the
23      * contents of the <code>names</code> array.
24     */
25     public void otherMethodThatShowsNames() {
26         //some code.
27         printNames();
28         //some code.
29     }
30
31     /**
32      * To perform its task, this method has to print the
33      * contents of the <code>names</code> array.
34     */
35     public void thirdMethodThatShowsNames() {
36         //some code.
37         printNames();
38         //some code.
39     }
40
41     private void printNames() {
42         for (String name : names) {
43             System.out.println("name: " + name);
44         }
45     }
46 }

```

**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 sequence[1] to access
6  * the first element in the sequence array.
7  */
8 public class ClassWithUnobviousDuplicatedCode {
9     private int[] sequence;
10
11     /**
12      * @return true 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  * firstElement
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 5.3. 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.util.Date;
4
5 /**
6  * This class represents a meeting in a calendar.
7  */
8 public class MeetingLowerCohesion {
9     private Date startTime;
10    private Date 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.util.Date;
18
19  /**
20   * Represents a period in time, with specific start
21   * and end time.
22   */
23  class TimePeriod {
24      private Date startTime;
25      private Date 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 very likely that a long parameter list changes, and changed parameters means changed public interface. The need to change the list often arises because it is long since it consists of primitive data, not objects. This means there is no encapsulation in the list, whenever the need of data changes in the method, it is reflected in its parameter list.

This bad smell can often be removed with the refactorings *Preserve Whole Object* or *Introduce Parameter Object*, which both replace primitive data with objects. The former 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). The latter 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   * dbHandler 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  * deposit 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  * deposit 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 currency The currency of the deposited amount.
13     * @param amount The amount to deposit.
14     */
15     public void deposit(Amount amount) {
16     }
17 }
18
19 package se.kth.ict.oodbook.prog.smell;
20

```

```

21  /**
22   * Represents an amount
23   */
24  class Amount {
25      private String currency;
26      private int amount;
27
28      //Methods
29  }

```

**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  * withdraw does not pass the
6  * fee 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 fee parameter
26  * is not passed to withdraw, 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. Primitive data is not forbidden, but should be used with care. Excessive primitive data means that everything related to object oriented development is just thrown in the wastebin. 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** This `Person` class uses objects for the fields.

It might be that a class has not only fields, but field(s) and one or more methods that are closer related than 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 the field(s) and methods of the original class that belong closely together. This refactoring is called *Replace Data Value With Object*. The code before the applying the refactoring is listed in listing 6.18. It shows the class `Person`, which has the `pnr` field (line 10) that holds 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 shall always 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. If arrays are used correctly, all elements in the same array means the same thing. It is not correct if different elements in an array means different things, as is the case with the `stats` array in listing 6.20. In this code, 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 a temporarily used variable `tmp` or `temp` (unless it represents a temperature). For example do not swap two values as in listing 6.24, instead use variable names like 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 variable is erroneously named `tmp`, just because it is used temporarily.

```

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.

## Unnamed Values

This is not mentioned as a particular smell in [FOW], but should still be avoided at all costs. All values in a program shall have an explaining name. Never introduce a value in a statement without naming it first, even if that statement is the only place the value is used. Without a name it can be very hard to understand the purpose of the value. Naming the value is a better practice than writing a comment to explain it. A name is part of the program, the compiler helps to ensure that 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 is a list with some examples of naming values.

- 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 defining 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 code this 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, the declaration of the constant.
- It is not required to use a variable or constant to name the value, sometimes a method suits better. 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 an unnamed value, like the 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`

## 6.6 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 the course web [CW]. Here follows a description of the first parts of the code, parts where particular afterthought was needed, and where the design 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. To implement a design in code is the only way to get a full understanding of its strengths and weaknesses. The best way to implement the design is still to code one system operation at a time, in the order they are executed. That makes it possible to test run each part of the program as soon as it is written. However, there is of course no reason to reiterate all the considerations made during design. Therefore, the final version of each system operation is implemented.

### 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 in figures below, but three things require extra attention. First, nothing has been decided on the implementation of the car registry. Where are the cars stored? The solution is to use a list with some hard coded cars, see lines 11, 14 and 46-53 in figure 6.31. This is enough for testing purposes. Second, neither requirements specification, nor design, are very specific on the search algorithm when looking for a matching car. To which extent must the search criteria be met to consider a car to match them? Can search criteria be ignored or must they all be specified? This should of course be discussed with the customer. The chosen implementation, see lines 49-69 in figure 6.30, 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 27-44 in figure 6.28. 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 71-81 in figure 6.30.

```

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 6.28** 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 6.29** 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 6.30** 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

```

```

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 that
22      *                    are set to null or
23      *                    0 are ignored.
24      * @return true if a car with the same
25      *         features as searchedCar was found,
26      *         false if no such car was found.
27      */
28     public CarDTO findCar(CarDTO searchedCar) {
29         for (CarDTO car : cars) {
30             if (car.matches(searchedCar)) {
31                 return car;
32             }
33         }
34         return null;
35     }
36
37     private void addCars() {
38         cars.add(new CarDTO(1000, "medium", true, true, "red",
39                             "abc123"));
40         cars.add(new CarDTO(2000, "large", false, true, "blue",
41                             "abc124"));
42         cars.add(new CarDTO(500, "medium", false, false, "red",
43                             "abc125"));
44     }
45 }

```

**Listing 6.31** 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 6.32** 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 6.33** 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 DTOs in the `Rental` constructor. For now, there is no reason to do anything at all, except to save them as fields in the constructed `Rental` object. Since all DTOs are immutable, there is no risk that any other object changes the content of a DTO. Remember that being *immutable* means an object can not change state, for example because the class

itself and all its fields are final. If DTOs had not been final, it would have been suicide to just keep them in `Rental`. In that case, the object that sent the DTO to `Rental` could have kept a reference to the same DTO object, and later updated it.

There is also another issue with keeping the DTOs. Is it really sure they are just DTOs, 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 obvious that all objects named DTO are, at least for the moment, DTOs. Therefore, it is perfectly safe to leave them like that now, but we must be aware that this might have to be changed in the future.

`Rental` is listed in listing 6.34, to illustrate the reasoning above. The rest of the `registerCustomer` implementation can be found in the accompanying NetBeans project.

```

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 6.34** 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 `bookCar` method in `CarRegistry`. The cars in the registry are currently stored in a list of `CarDTOs`. 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 91-111 in listing 6.35.

```

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   * Books the specified car. After calling this method, the
41   * car can not be booked by any other customer.
42   *
43   * @param car The car that will be booked.
44   */
45  public void bookCar(CarDTO car) {
46      CarData carToBook = findCarByRegNo(car);
47      carToBook.booked = true;
48  }
49
50  private void addCars() {
51      cars.add(new CarData("abc123", 1000, "medium", true,
52                          true, "red"));
53      cars.add(new CarData("abc124", 2000, "large", false,
54                          true, "blue"));
55      cars.add(new CarData("abc125", 500, "medium", false,
56                          false, "red"));
57  }
58
59  private boolean matches(CarData found, CarDTO searched) {
60      if (searched.getPrice() != 0 &&
61          searched.getPrice() != found.price) {
62          return false;
63      }
64      if (searched.getSize() != null &&
65          !searched.getSize().equals(found.size)) {
66          return false;
67      }
68      if (searched.getColor() != null &&
69          !searched.getColor().equals(
70              found.color)) {
71          return false;
72      }
73      if (searched.isAC() != found.AC) {
74          return false;
75      }
76      if (searched.isFourWD() != found.fourWD) {
77          return false;
78      }
79      return true;
80  }
81

```

```

82     private CarData findCarByRegNo(CarDTO searchedCar) {
83         for (CarData car : cars) {
84             if (car.regNo.equals(searchedCar.getRegNo())) {
85                 return car;
86             }
87         }
88         return null;
89     }
90
91     private static class CarData {
92         private String regNo;
93         private int price;
94         private String size;
95         private boolean AC;
96         private boolean fourWD;
97         private String color;
98         private boolean booked;
99
100        public CarData(String regNo, int price, String size,
101                        boolean AC, boolean fourWD,
102                        String color) {
103            this.regNo = regNo;
104            this.price = price;
105            this.size = size;
106            this.AC = AC;
107            this.fourWD = fourWD;
108            this.color = color;
109            this.booked = false;
110        }
111    }
112 }

```

**Listing 6.35** 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 59-80 in listing 6.35. 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.

Since a booked car can not be rented by other customers, the `findCar` method must not return a booked car, even if it matches the search criteria. This results in the `if`-statement

on line 30 in listing 6.35. To clarify this new behavior, the method name is changed 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 `regNo` first. This is a clear sign that this 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 `bookCar` in `CarRegistry`.

There is one unhandled issue left in the current `CarRegistry` implementation, what happens if the car that shall be booked, in `bookCar`, is already booked? This situation is not handled yet, but should be addressed before the program is completed. That concludes the implementation of `bookCar`, the rest of the code can be downloaded in the accompanying NetBeans project.

## 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 6.36. The printer will call this method to get 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 it 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 6.37. This might seem a strange solution. Why shall `Rental` call `CashPayment`, which then only calls back to `Rental` to get the cost. Why not just let `Rental` pass the cost to `CashPayment`? The reason is that it is assumed that, as the program grows, `CashPayment` will have to gather more information, like driven distance and possible discounts. It is `CashPayment` who has this knowledge about what data is needed, from where to get it, and how to use it to calculate the total cost.

```

1 package se.kth.ict.oodbook.rentcar.model;
2
3 import java.util.Date;
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         Date rentalTime = new Date();
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().getTotalCost().
40                     toString());
41         builder.append("Change: ");
42         appendLine(builder, rental.getPayment().getChange().
43                     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
59 }

```

**Listing 6.36** 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 6.37** The class `CashPayment`, after implementing the pay system operation.

## 6.7 Common Mistakes When Implementing the Design

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.

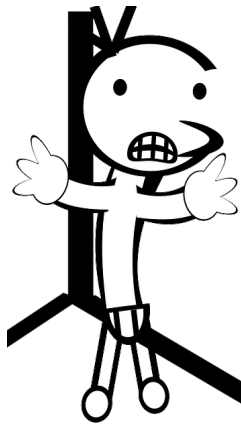
**NO!**

- Many of the common design mistakes can be introduced when programming, even if they were avoided during design. For example, there is the 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 5.7 for more details on this.
- Section 6.5, 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*.

NO!

# 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 become 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 4 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 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 expectedResult = true;
7     boolean result = instance.equals(other);
8     assertEquals(
9         "Amount instances with same states are not equal.",
10        expectedResult, result);
11 }

```

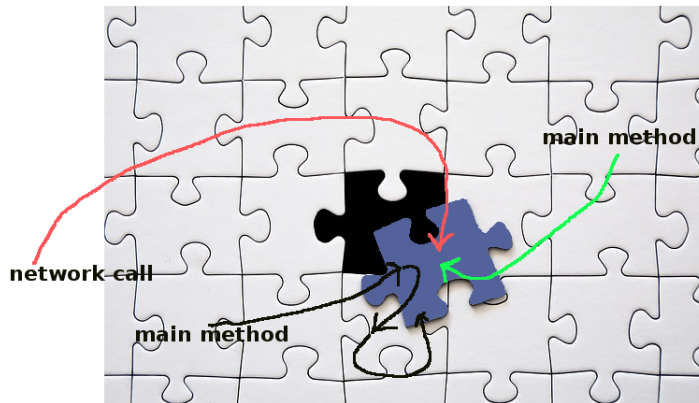
**Listing 7.2** A unit test for the code in listing 7.1

## 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. The framework is responsible for calling application code at the right time. This fact, that the application is relieved of flow control responsibility, 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 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 and green lines. Execution may also enter the framework from the outside, for example via a network call, as is the case for the red line. 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.

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.



**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.

## 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 specify the content 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
<b>@Test</b> <b>public void aTest()</b>	<b>aTest</b> contains tests and will be executed when tests are run.
<b>@Test(expected=Exception.class)</b> <b>public void aTest()</b>	Test fails if <b>aTest</b> does not throw an exception of class <b>Exception</b>
<b>@Ignore("Not implemented")</b> <b>@Test</b> <b>public void aTest()</b>	<b>aTest</b> will not be executed.
<b>@Before</b> <b>public void prepareTest()</b>	<b>prepareTest</b> is executed before each test method.
<b>@After</b> <b>public void cleanup()</b>	<b>cleanup</b> is executed after each test method.
<b>@BeforeClass</b> <b>public void prepareTests()</b>	<b>prepareTests</b> is executed once before the first test in this class.
<b>@AfterClass</b> <b>public void cleanup()</b>	<b>cleanup</b> 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.

With this knowledge about frameworks and JUnit, we can understand the first example in listing 7.2 in more detail. The complete test class is listed in listing 7.3.

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

Table 7.2 Some of the most common JUnit assert methods

```

1 package se.kth.ict.oodbook.tests.firstexample;
2
3 import org.junit.After;
4 import org.junit.Before;
5 import org.junit.Test;
6 import static org.junit.Assert.*;
7
8 public class AmountTest {
9     private Amount amtNoArgConstr;
10    private Amount amtWithAmtThree;
11
12    @Before
13    public void setUp() {
14        amtNoArgConstr = new Amount();
15        amtWithAmtThree = new Amount(3);
16    }
17
18    @After
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("Amount instance equal to null.",
30             expResult, result);
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("Amount instance equal to " +
39             "java.lang.Object instance.",
40             expResult, result);
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("Amount instances with different " +
50             "states are equal.",
51             expResult, result);
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("Amount instances with different " +
61             "states are equal.",
62             expResult, result);
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("Amount instances with same states " +

```

```

72         "are not equal.",
73         expResult, result);
74     }
75
76 }

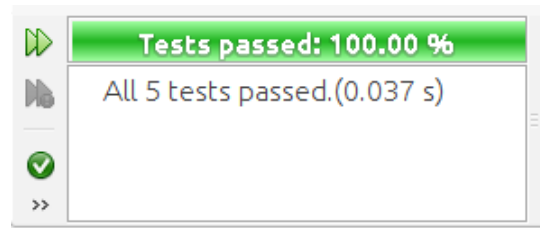
```

**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 `@Before`. This means it is executed before each test method. That way, each test is performed on the two new `Amount` objects created in `setUp`. In a similar way, the `tearDown` method, which is annotated `@After` 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 so much time planning tests that the burden of writing them becomes so big that they are not written, or that they 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 `@Ignore` 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 everything in the program that we know can go wrong.

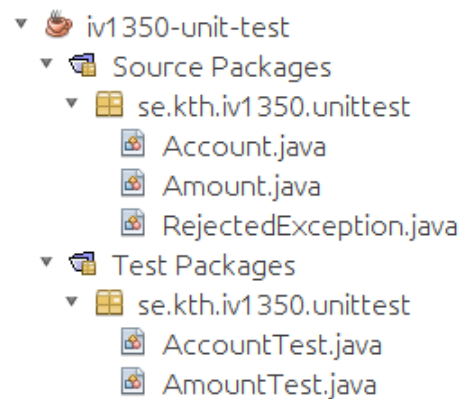
**Do not over-design** There is no need to design or document test code as thoroughly as the product that is tested. Allow a certain amount of hacking when writing tests. In test code, we can play around a bit and write some of those funny and interesting hacks that never really seem to fit in production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

**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 confidence that the code is really working, and that it will be easy to verify that it is still working if we have to make changes.

**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.

**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 is tests and which is the 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.

It is common to write one test class per tested class, and give it the same name as the tested class, but appending `Test` 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.



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

**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 the test class. Also, if methods with all other accessibilities work, also private methods work. It is normally not needed to test setters or constructors that only save values, nor getters that only return a value. Unless bugs appear, we can take for granted that such methods work as intended. 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.

**Never worsen SUT design!** Try to never, under any condition, worsen the design of the SUT just to enable testing. This is a slightly controversial statement. It is often suggested, for example, to break encapsulation by adding get methods to enable retrieving the state of an object. The only purpose of breaking encapsulation would be to verify that the state is correct after a method in the object is tested. This is, however, practically never necessary. Using hard work, a lot of fantasy, and pragmatically testing more than one method together, we can almost always write tests without worsening SUT design. A method must have some effect somewhere, otherwise it is 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. The create, update, read, etc. More on testing in difficult situations follows below.

### 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.

**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 obvious how the test shall provide the input. 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



**Figure 7.6** Testing can be very complicated and frustrating, but with hard work, a lot of fantasy, and pragmatically testing more than one method together, we can almost always write tests without worsening SUT design. Image by unknown creator [Public domain], via <https://pixabay.com>

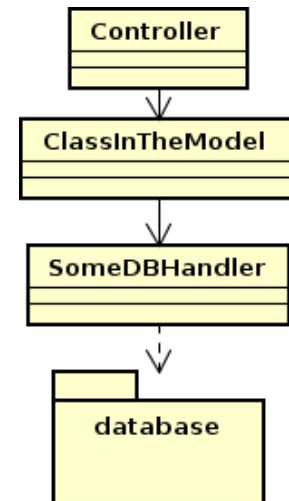


SUT. It is quite common to write a large amount of test code in order to create files, databases, etc required for testing. Whatever test structure is created, must be deleted after the test is executed. Remember that tests must be completely independent and repeatable. A test must leave no traces of its execution.

**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. Fact is that the SUT must update something somewhere, or it would be useless. Maybe the problematic method can not be tested alone, but there is often some combination of method calls that will show if the test passed. This reasoning is expanded above, in the paragraph labeled *Never worsen SUT design* in section 7.2.

**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!

Finally, it can not be stated often enough, *whatever the problem is, do not worsen SUT design just to enable testing.*



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

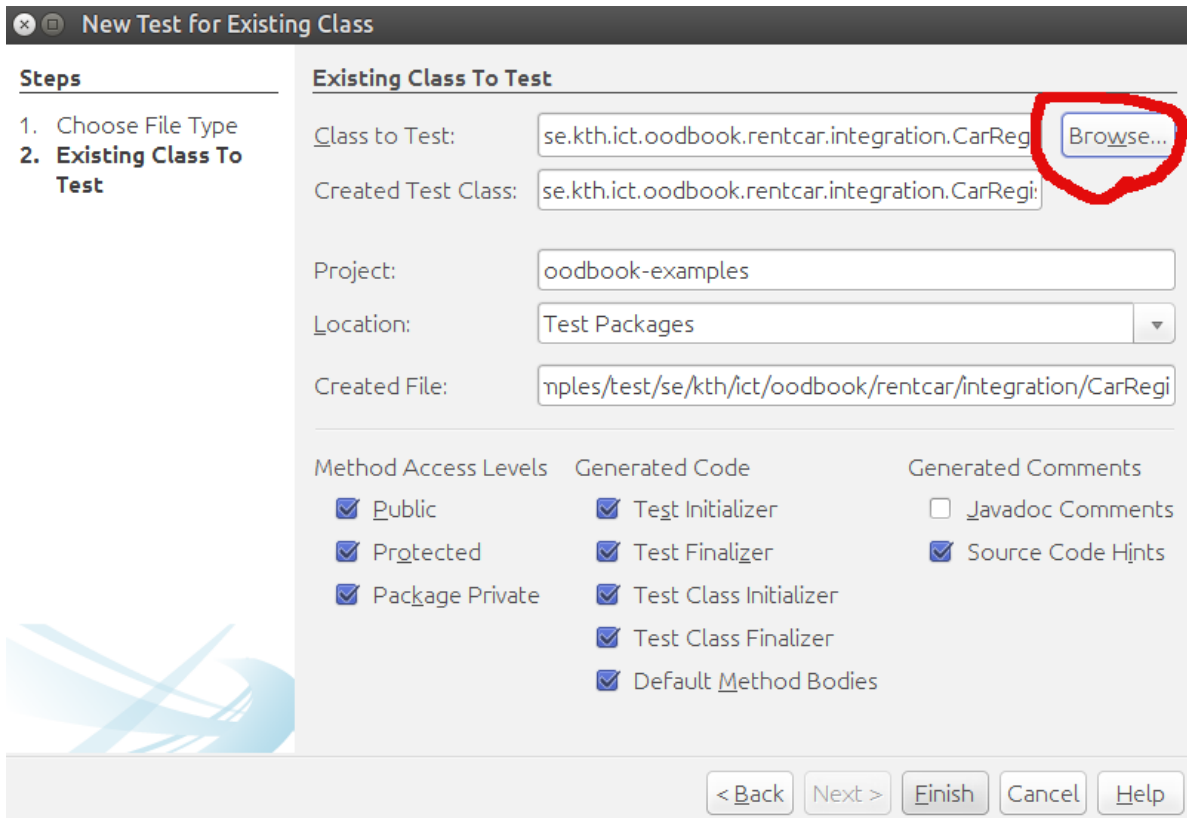
## 7.4 Unit Testing Case Study

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 the course web [CW]. 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 used when developing this unit test case study. This section illustrates how NetBeans facilitates creating the tests. Similar functionality is available in all 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. Last, 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.After;
4 import org.junit.AfterClass;
5 import org.junit.Before;
6 import org.junit.BeforeClass;
7 import org.junit.Test;
8 import static org.junit.Assert.*;

```

```

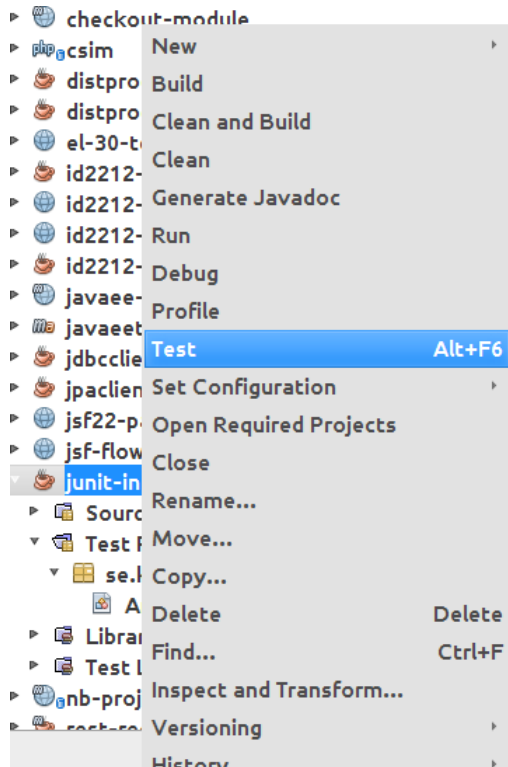
9
10 public class CarRegistryTest {
11     @BeforeClass
12     public static void setUpClass() {
13     }
14
15     @AfterClass
16     public static void tearDownClass() {
17     }
18
19     @Before
20     public void setUp() {
21     }
22
23     @After
24     public void tearDown() {
25     }
26
27     @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         // TODO review the generated test code and remove the
36         // default call to fail.
37         fail("The test case is a prototype.");
38     }
39
40     @Test
41     public void testBookCar() {
42         System.out.println("bookCar");
43         CarDTO car = null;
44         CarRegistry instance = new CarRegistry();
45         instance.bookCar(car);
46         // TODO review the generated test code and remove the
47         // default call to fail.
48         fail("The test case is a prototype.");
49     }
50 }

```

**Listing 7.4** Skeleton code for a test class, 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-

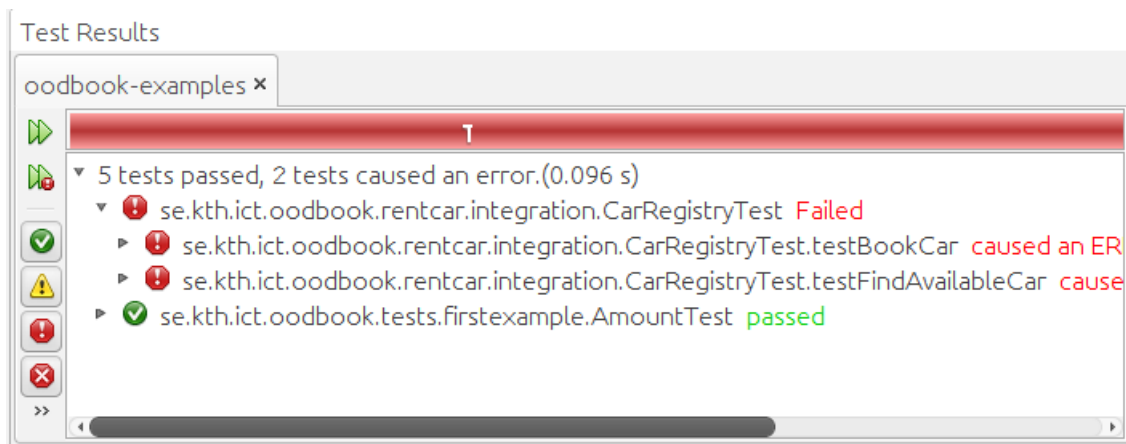
25), but they are empty. If some code is needed to prepare a test or to clean up after a test, it shall be added here. Methods that remain empty can be removed. One test method is generated for each public, protected or package private method in the SUT (lines 27-49). These methods contain a printout (lines 29 and 42), which should be removed since tests are not supposed to produce any output if they pass.



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

After this, the test methods create an instance of the SUT (lines 31 and 44) and of other objects that are necessary to perform the test (lines 30 and 43). 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 (lines 33 and 45). Then an assertion is called to evaluate the test result (line 34). The `testBookCar` method contains no assertion, since `bookCar` is void. In this case, NetBeans does not know how to evaluate the outcome. Again, even if the assertion is generated, there is no guarantee it is correct. Finally, there is a TODO comment and a call to `fail` (lines 35-37 and 46-48), which should both be removed when the test is completed.

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 `testBookCar`, are implemented. As a result, they both fail.



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

## 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. We first wrote the entire program and then started to write tests. Normally, a test is written either before or immediately after the method that shall be tested. The only reason for this workflow is to mix theory and practice, by using programming best practices as soon as they were introduced. !

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. The `Amount` class has only public methods and constructors, so they shall all be tested. The constructors, however, contain no logic, they only set a value. Therefore, they will not be tested. 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 7.5.

```

1  /**
2   * Two Amounts are equal if they represent the
3   * same amount.
4   *
5   * @param other The Amount to compare with this
6   *               amount.
7   * @return true if the specified amount is equal
8   *         to this amount, false 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 7.5** The `equals` method of the `Amount` class

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 7.6.

```

1 package se.kth.ict.oodbook.rentcar.model;
2
3 import org.junit.After;
4 import org.junit.Before;
5 import org.junit.Test;
6 import static org.junit.Assert.*;
7
8 public class AmountTest {
9     private Amount amtNoArgConstr;
10    private Amount amtWithAmtThree;
11
12    @Before
13    public void setUp() {
14        amtNoArgConstr = new Amount();
15        amtWithAmtThree = new Amount(3);
16    }
17
18    @After
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("Amount instance equal to null.",
30                    expResult, result);
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("Amount instance equal to " +
39                    "java.lang.Object instance.",

```

```

40         expResult, result);
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("Amount instances with different states" +
50             " are equal.", expResult, result);
51     }
52
53     @Test
54     public void testNotEqual() {
55         int amountOfOther = 2;
56         Amount other = new Amount(amountOfOther);
57         boolean expResult = false;
58         boolean result = amtWithAmtThree.equals(other);
59         assertEquals("Amount instances with different states" +
60             " are equal.", expResult, result);
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("Amount instances with same states are" +
70             " not equal.", expResult, result);
71     }
72 }

```

**Listing 7.6** 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 test also with an `Amount` object representing zero. That way, also the default constructor will be executed in a test. Theoretically, we could test with `Amounts` representing positive, negative, `Integer.MAX_VALUE` and `Integer.MIN_VALUE` amounts, but there is really no reason to suspect that the method would behave differently for such values. Listing 7.7 shows the test for an `Amount` with the value zero.

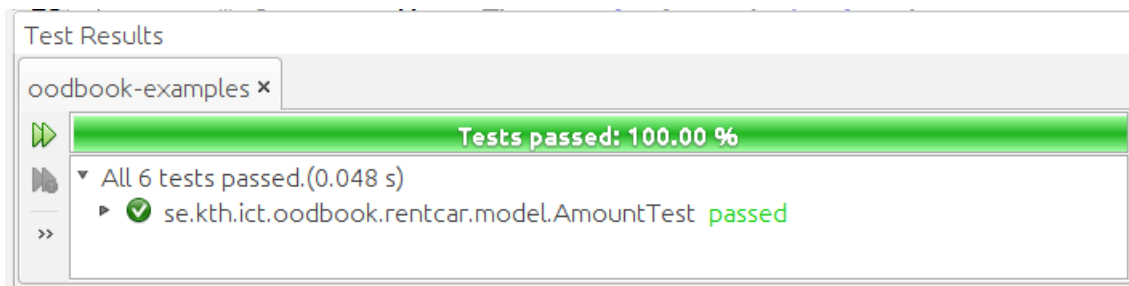
```

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

```

**Listing 7.7** 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 7.11. Our first green bar!!



**Figure 7.11** 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 7.8.

```

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 7.8** 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 later 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 7.9. 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 expectedResult = new Amount(amountOfOperand1 -
8                                          amountOfOperand2);
9      Amount result = operand1.minus(operand2);
10     assertEquals("Wrong subtraction result",
11                 expectedResult, 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 expectedResult = new Amount(amountOfOperand1 -
21                                         amountOfOperand2);
22     Amount result = operand1.minus(operand2);
23     assertEquals("Wrong subtraction result",
24                 expectedResult, result);
25 }
26 @Test

```

```

27 public void testMinusZeroResultNegOperand() {
28     int amountOfOperand1 = -3;
29     int amountOfOperand2 = -3;
30     Amount operand1 = new Amount(amountOfOperand1);
31     Amount operand2 = new Amount(amountOfOperand2);
32     Amount expResult = new Amount(amountOfOperand1 -
33                                   amountOfOperand2);
34     Amount result = operand1.minus(operand2);
35     assertEquals("Wrong subtraction result",
36                 expResult, result);
37 }

```

**Listing 7.9** 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 7.10. Also this method is tested with positive, negative, and zero amounts, see listing 7.11. 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 7.10** 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("Wrong string returned by toString",
8                  expResult, result);
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("Wrong string returned by toString",
18                 expResult, result);
19 }

```

```

20
21 @Test
22 public void toStringZeroAmt() {
23     int representedAmt = 0;
24     Amount amount = new Amount(representedAmt);
25     String expectedResult = Integer.toString(representedAmt);
26     String result = amount.toString();
27     assertEquals("Wrong string returned by toString",
28         expectedResult, result);
29 }

```

**Listing 7.11** 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. The next class to test is `CarRegistry`, which is a bit more challenging since it has a `void` method, namely `bookCar`. 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, `bookCar` can be tested together with `findAvailableCar`, as illustrated in listing 7.13. Listing 7.12 shows `bookCar` and `findAvailableCar`. There is an interesting problem here, the calls to `assertEquals`, for example on line nine in listing 7.13, 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   * 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      CarData carToBook = findCarByRegNo(car);
31      carToBook.booked = true;
32  }

```

**Listing 7.12** The bookCar and findAvailableCar methods of the CarRegistry class

```

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

```

**Listing 7.13** The tests for the bookCar 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. When developing continues, it will certainly be possible to read the balance of the cash register and to see which rentals have been made. Thereby, it will also be 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.

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 7.14), require quite a lot of work to set up. This is quite normal, and should be expected to happen. Listing 7.15 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     Date rentalTime = new Date();
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 7.14** 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     Date rentalTime = new Date();
19     String expResult = "Car Rental\n\nRental time: "
20                         + rentalTime.toString()
21                         + "\n\nRented car: " + regNo
22                         + "\nCost: " + price
23                         + "\nChange: " + paidAmt.minus(price)
24                         + "\n\n";
25     String result = instance.createReceiptString();
26     assertEquals("Wrong receipt content.", expResult, result);
27 }

```

**Listing 7.15** The test for the `createReceiptString` method of the `Receipt` class

## Testing User Interface

`printReceipt` in `Printer` (listing 7.16) 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 line seven in listing 7.17. 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 System.out instead of a printer.
4   *
5   * @param receipt
6   */
7  public void printReceipt(Receipt receipt) {
8      System.out.println(receipt.createReceiptString());
9  }
```

**Listing 7.16** The `createReceiptString` method of the `Receipt` class, and its private helper methods.

```

1  public class PrinterTest {
2      ByteArrayOutputStream outContent;
3
4      @Before
5      public void setUpStreams() {
6          outContent = new ByteArrayOutputStream();
7          System.setOut(new PrintStream(outContent));
8      }
9
10     @After
11     public void cleanUpStreams() {
12         outContent = null;
13         System.setOut(null);
14     }
15
16     @Test
17     public void testCreateReceiptString() {
```

```

18     Amount price = new Amount(1000);
19     String regNo = "abc123";
20     String size = "medium";
21     boolean AC = true;
22     boolean fourWD = true;
23     String color = "red";
24     CarDTO rentedCar = new CarDTO(regNo, price, size, AC,
25                                   fourWD, color);
26     Amount paidAmt = new Amount(5000);
27     CashPayment payment = new CashPayment(paidAmt);
28     Rental paidRental = new Rental(null,
29                                   new RegistryCreator().
30                                   getCarRegistry());
31     paidRental.setRentedCar(rentedCar);
32     paidRental.pay(payment);
33     Receipt receipt = new Receipt(paidRental);
34     Printer instance = new Printer();
35     instance.printReceipt(receipt);
36     Date rentalTime = new Date();
37     String expResult = "Car Rental\n\nRental time: "
38                       + rentalTime.toString()
39                       + "\n\nRented car: " + regNo
40                       + "\nCost: " + price
41                       + "\nChange: "
42                       + paidAmt.minus(price) + "\n\n\n";
43     String result = outContent.toString();
44     assertEquals("Wrong printout.", expResult, result);
45 }
46 }

```

**Listing 7.17** 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 7.19, which tests that the method `bookCar` (listing 7.18) 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   *
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 7.18** The `bookcar` method of the Controller class.

```

1  public class ControllerTest {
2      private Controller instance;
3      private RegistryCreator regCreator;
4      ByteArrayOutputStream outContent;
5      PrintStream originalSysOut;
6
7      @Before
8      public void setUp() {
9          originalSysOut = System.out;
10         outContent = new ByteArrayOutputStream();
11         System.setOut(new PrintStream(outContent));
12         Printer printer = new Printer();
13         regCreator = new RegistryCreator();
14         instance = new Controller(regCreator, printer);
15     }

```

```

16
17     @After
18     public void tearDown() {
19         outContent = null;
20         System.setOut(originalSysOut);
21         instance = null;
22         regCreator = null;
23     }
24
25     @Test
26     public void testRentalWithBookedCarIsStored() {
27         String customerName = "custName";
28         CustomerDTO rentingCustomer =
29             new CustomerDTO(customerName,
30                             new AddressDTO("street", "zip",
31                                             "city"),
32                             new DrivingLicenseDTO("1234567"));
33         String regNo = "abc123";
34         CarDTO rentedCar = new CarDTO(regNo, new Amount(1000),
35                                       "medium", true,
36                                       true, "red");
37         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("Wrong number of stored rentals.",
45                     expectedNoOfStoredRentals,
46                     noOfStoredRentals);
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("Saved rental does not contain rented car",
54                   result.contains(regNo));
55     }
56 }

```

**Listing 7.19** 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.

That concludes the rent car testing case study. A total of 56 test methods were created, which should be acceptable for such a small program, completely lacking exception handling. All 56 tests pass, which gives the joyful sight presented in figure 7.12. Quite amazingly, the SUT consists of 1351 lines of code in total, and the tests of 1322 (no cheating). A difference of only two percent!

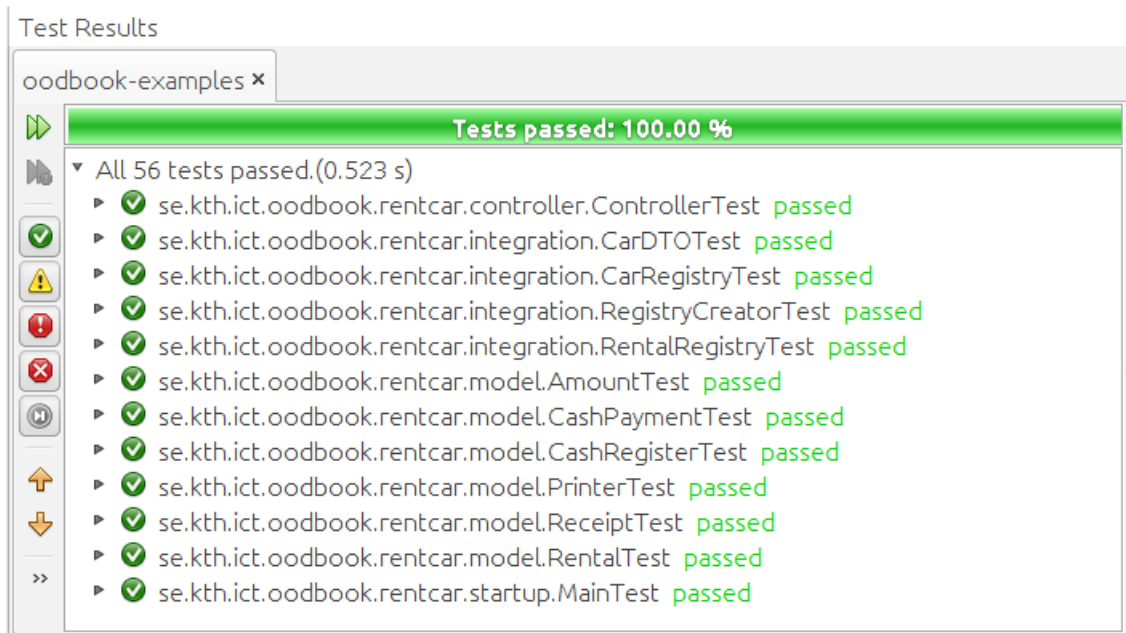


Figure 7.12 All 56 tests of the case study pass.

## 7.5 Common Mistakes When Writing Unit Tests

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.

**NO!**

**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. 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 are actually required.

**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.

**NO!**

## **Chapter 8**

# **Exception Handling**

## **Chapter 9**

# **Polymorphism and Design Patterns**

# **Chapter 10**

## **Inheritance**

# **Chapter 11**

## **Inner Classes**



# **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.

<i>English</i>	<i>Swedish</i>
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, refaktorerering, refaktorisering
sequence diagram .....	sekvensdiagram
system operation .....	systemoperation
system sequence diagram ...	systemsekvensdiagram
visibility .....	åtkomst

# Appendix B

## UML Cheat Sheet

### Class Diagram

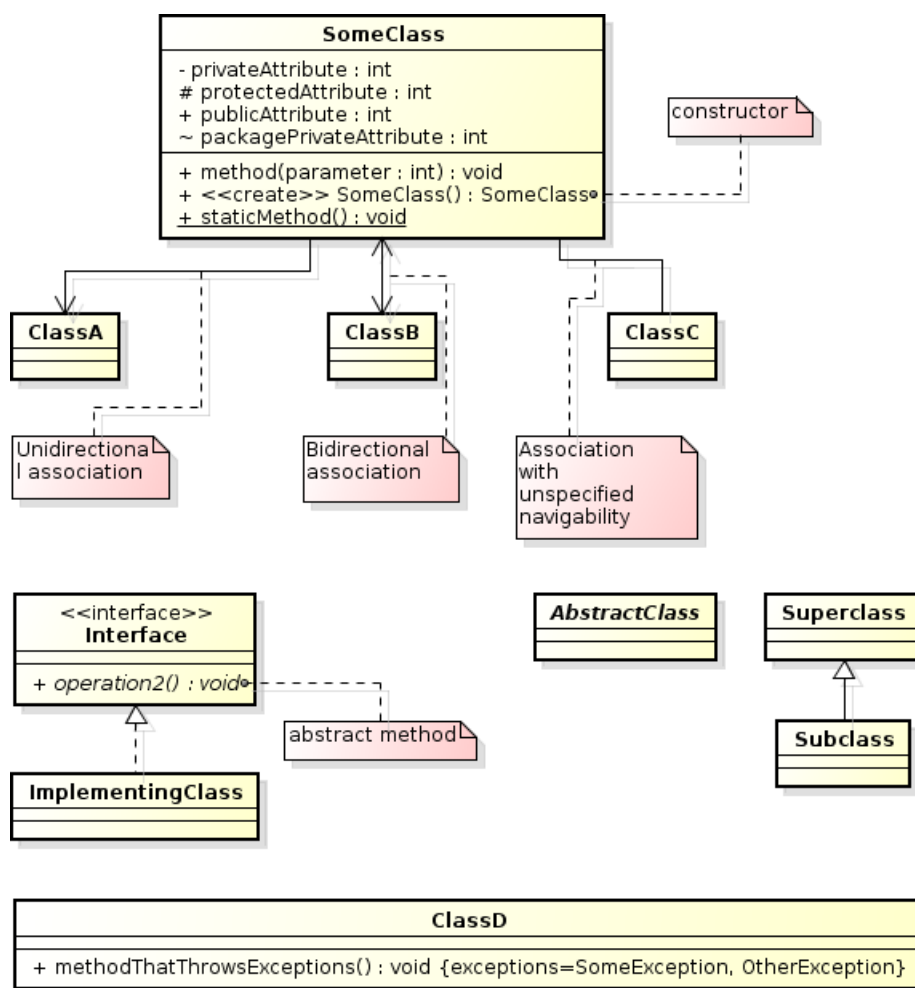


Figure B.1 Class diagram

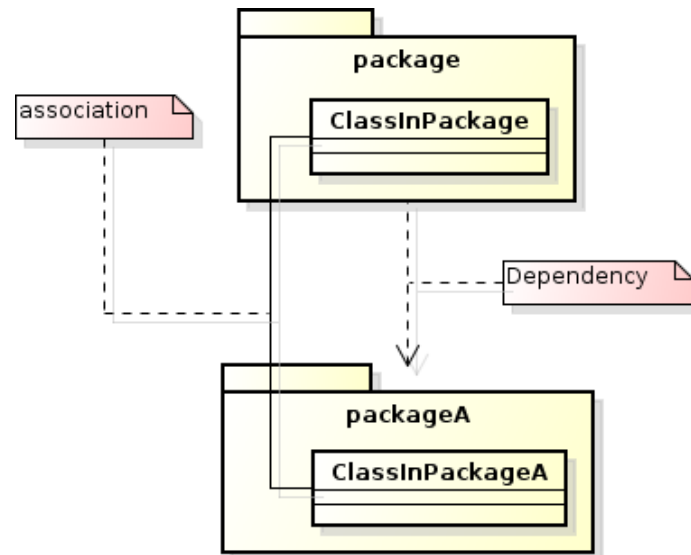


Figure B.2 Packages in class diagram

## Sequence Diagram

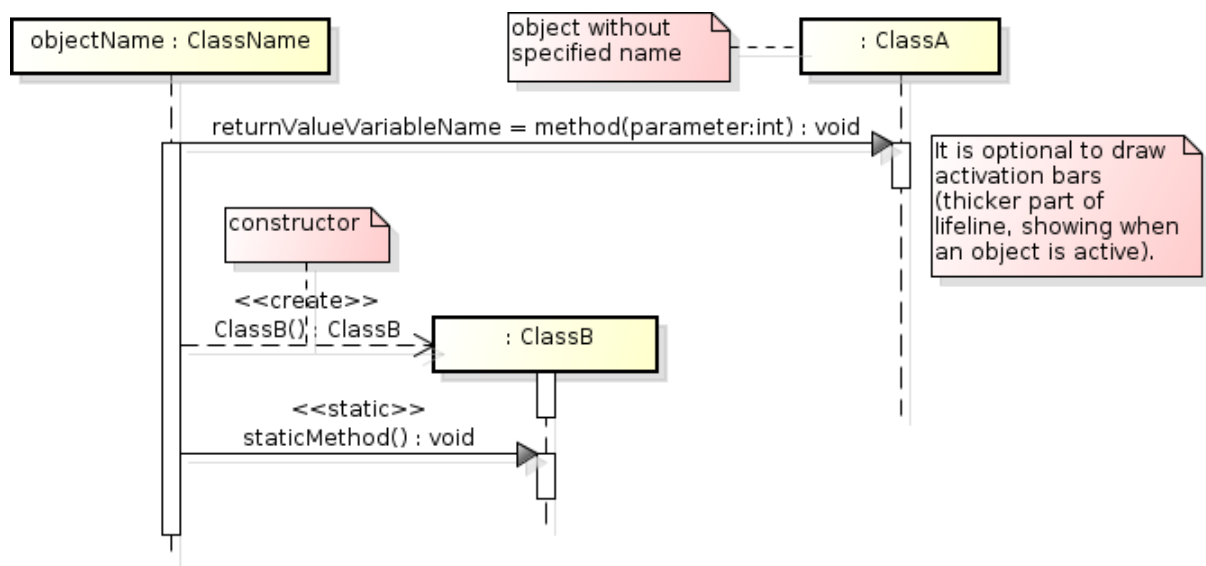
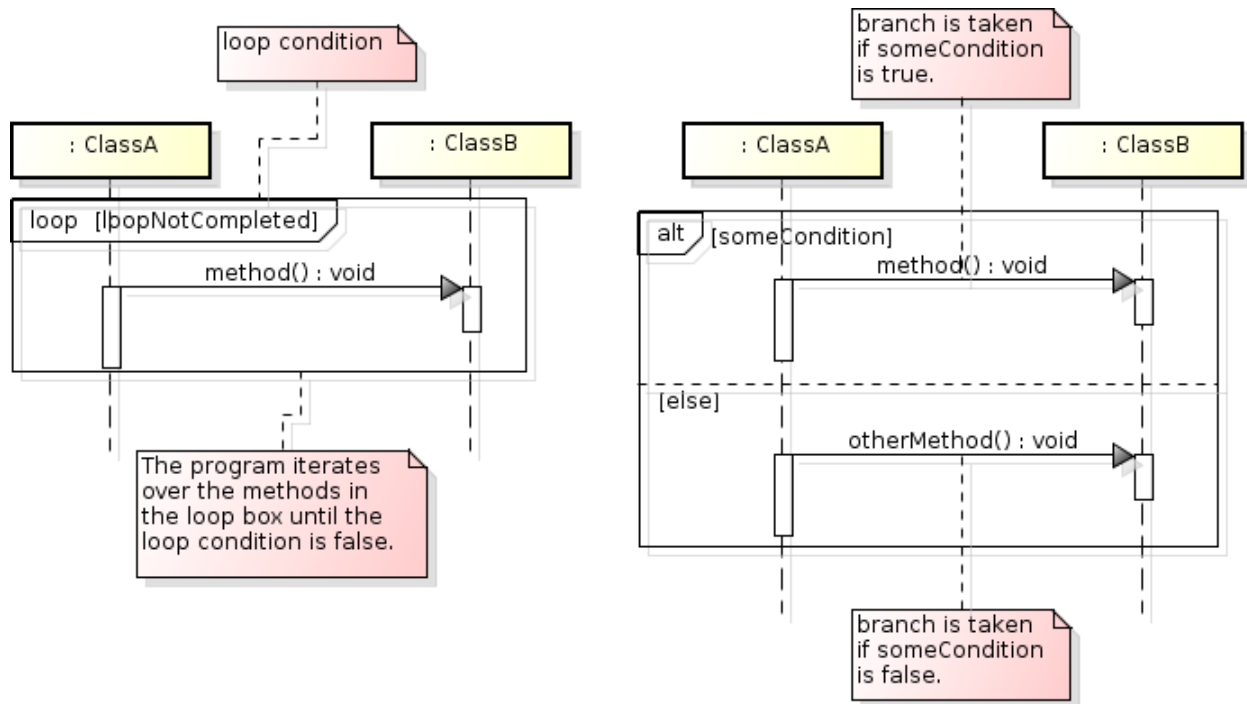
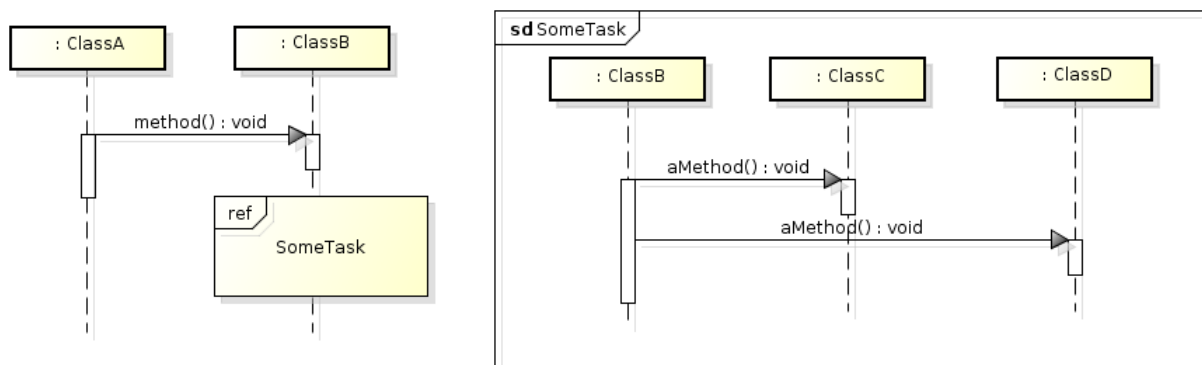


Figure B.3 Sequence diagram



**Figure B.4** Flow control in sequence diagram



**Figure B.5** Reference to other sequence diagram

## Communication Diagram

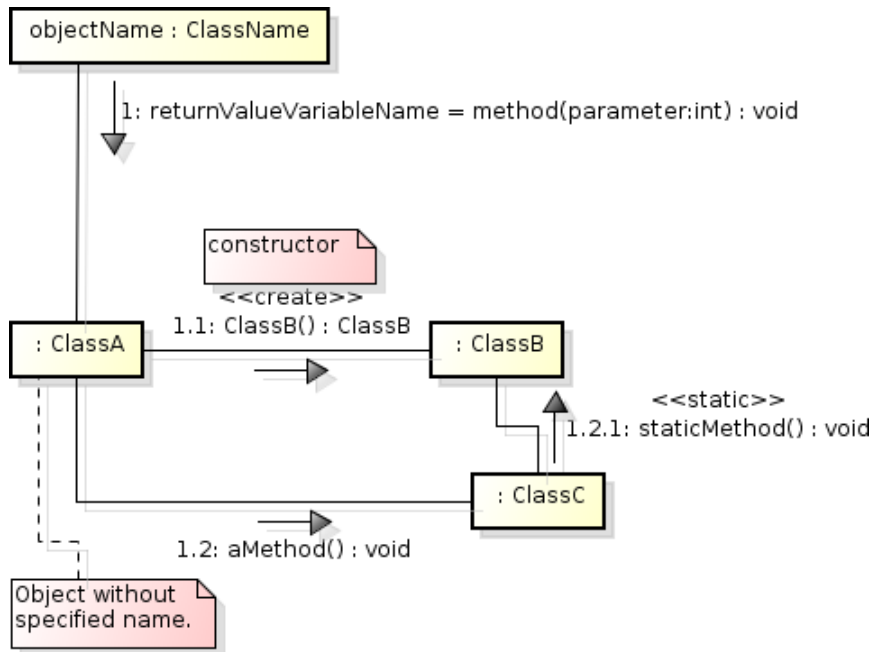


Figure B.6 Communication diagram

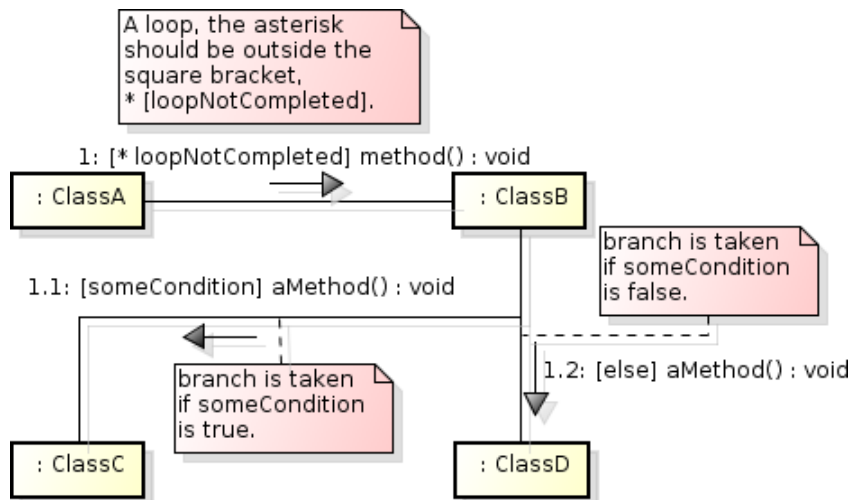


Figure B.7 Flow control in communication diagram

# 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 somewhere 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 /**

```

```

6  * Represents an employee.
7  */
8  public class BadDesignEmployee {
9      private String name;
10     private Address address;
11     private Amount salary;
12
13     /**
14      * Changes the salary of <code>employee</code> to
15      * <code>newSalary</code>.
16      *
17      * @param employee The <code>Employee</code> whose salary will be
18      *                  changed.
19      * @param newSalary The new salary of <code>employee</code>.
20      */
21     public void changeSalary(BadDesignEmployee employee,
22                             Amount newSalary) {
23     }
24
25     /**
26      * Returns a list with all employees working in the same
27      * department as this employee.
28      */
29     public List<BadDesignEmployee> getAllEmployees() {
30     }
31 }

```

Listing C.21 Java code implementing the UML diagram in figure 5.9a

```

1  package se.kth.ict.oodbook.design.cohesion;
2
3  /**
4   * Represents an employee.
5   */
6  public class Employee {
7      private String name;
8      private Address address;
9      private Amount salary;
10
11     /**
12      * Changes the salary to <code>newSalary</code>.
13      *
14      * @param newSalary The new salary.
15      */
16     public void changeSalary(Amount newSalary) {
17         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 packagePrivateAttribute;
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.

```

```

9      */
10     public class View {
11         // Somewhere in the code. Note that the arguments to the
12         // Car constructor are not specified in the UML diagram.
13         Car searchedCar = new Car(0, null, false, false,
14                                 null, null);
15         Car foundCar = contr.searchMatchingCar(searchedCar);
16     }
17
18 }

```

Listing C.52 Java code implementing the View class in figure 5.25

```

1 package se.kth.ict.oodbook.design.casestudy.controller;
2
3 import se.kth.ict.oodbook.design.casestudy.dbhandler.Car;
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 Car searchMatchingCar(Car searchedCar) {
11     }
12 }

```

Listing C.53 Java code implementing the Controller class in figure 5.25

```

1 package se.kth.ict.oodbook.design.casestudy.dbhandler;
2
3 /**
4  * Contains information about one particular car.
5  */
6 public class Car {
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     *

```

## Appendix C Implementations of UML Diagrams

```
18      * @param price The price paid to rent the car.
19      * @param size The size of the car, e.g., medium
20                hatchback.
21      * @param AC      true if the car has air
22                condition.
23      * @param fourWD true 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 {

```

## Appendix C Implementations of UML Diagrams

```
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.
```

## Appendix C Implementations of UML Diagrams

```
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

```

## Appendix C Implementations of UML Diagrams

```
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.
```



## Appendix C Implementations of UML Diagrams

```
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     /**

```

## Appendix C Implementations of UML Diagrams

```
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();

```

## Appendix C Implementations of UML Diagrams

```
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.

```

```

10  */
11  public class Main {
12      public static void main(String[] args) {
13          RegistryCreator creator = new RegistryCreator();
14          Controller contr = new Controller(creator);
15          new View(contr);
16      }
17  }

```

**Listing C.101** Java code implementing the Main class in figure 5.35

```

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.102** Java code implementing the CarRegistry class in figure 5.35

```

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.103** Java code implementing the RentalRegistry class in figure 5.35

```

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      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.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     /**
12     * Creates a new instance.
13     *
14     * @param regCreator Used to get all classes that handle
15     *                   database calls.
16     */
17     public Controller(RegistryCreator regCreator) {
18     }
19 }

```

**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 paid 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 {

```

## Appendix C Implementations of UML Diagrams

```
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     /**

```

## Appendix C Implementations of UML Diagrams

```
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

```

## Appendix C Implementations of UML Diagrams

```
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  /**

```

## Appendix C Implementations of UML Diagrams

```
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

```
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.167** Java code implementing the CarRegistry class in figure 5.43

```

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.168** Java code implementing the RentalRegistry class in figure 5.43

# Bibliography

[LAR] Larman, Craig: *Applying UML and Patterns*, third edition, Prentice-Hall 2004, ISBN:0-13-148906-2

[CW] Course web for Object Oriented Design, IV1350 <http://www.kth.se/social/course/IV1350/page/lecture-notes-60/>

[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>

[FOW] Fowler, Martin: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley 1999, ISBN: 978-0201485677

[JU] Home page for the JUnit unit testing framework <http://junit.org/junit4/>

[NB] Home page for the NetBeans IDE <http://netbeans.org/>

# Index

- @Override, 10
- activation bar, 22, 40
- actor, 33
- alternative flow, 18
- analyses, 15, 20
- anchor, 24
- annotation, 9, 124
- architectural pattern, 50
- architecture, 50
- array, 5
- assert, 124
- association, 21, 29
  - direction of, 21
  - multiplicity, 22, 29
  - name, 21
- attribute, 21, 28
- basic flow, 17
- business logic, 51
- category list, 24
- class, 2, 21, 24
- class candidate, 24
- class diagram, 20
- code convention, 81
- code smell, 83
- coding, 15, 58
  - mistake, 119
- cohesion, 46
- collection, 5
- combined fragment, 23
- comment, 24
- communication diagram, 42
- constant, 102
- constructor, 3, 41, 42
- controller, 52
- coupling, 48
- design, 15, 39
  - concept, 43
  - method, 57
  - mistake, 78
- dictionary, 155
- domain model, 24
  - naïve, 31
  - programmatic, 31
- DTO, 55
- duplicated code, 84
- encapsulation, 43
- exception, 6
  - checked, 7
  - runtime, 7
- found message, 40
- framework, 123
- guard, 37
- high cohesion, 46
- immutable, 67
- implementation, 43
- inheritance, 10
- integration, 15
- interaction diagram, 42, 58, 61
- interaction operand, 23
- interaction use, 41
- interaction operator, 23
- interface, 9
- iteration, 14

- javadoc, 7
  - @param, 7
  - @return, 7
- JUnit, 122, 124
- large class, 90
- layer, 54
- lifeline, 22
- list, 5
- long method, 88
- long parameter list, 91
- low coupling, 48
- meaningless name, 101
- member, 40
- message, 22
- methodologies, 13
- model, 51
- MVC, 51
- naming convention, 23
- naming identifier, 101
- NetBeans, 131
- new, 4
- note, 24
- noun identification, 24
- object, 2, 22
- operation, 21
- overload, 11
- package, 50
- package diagram, 40
- package private, 50
- pattern, 50, 51, 54, 55
- primitive variables, excessive use, 96
- programming, *see* coding
- public interface, 43
- refactoring, 83
- reference, 4
- requirements analyses, 14
- sequence diagram, 22, 40
- spider-in-the-web, 33, 48, 69, 78
- state, 51
  - static, 3, 40, 41
  - stereotype, 41
  - subclass, 10
  - super, 11
  - superclass, 10
  - SUT, 122
  - system operation, 33, 52, 57, 61
  - system sequence diagram, 33
  - system under test, 122
- test, 14, 15, 121
- this, 3
- type, 11
- UML, 16, 20, 39
- unit test, 122
- unnamed values, 102
- utility class, 80
- view, 51
- visibility, 40, 43, 50