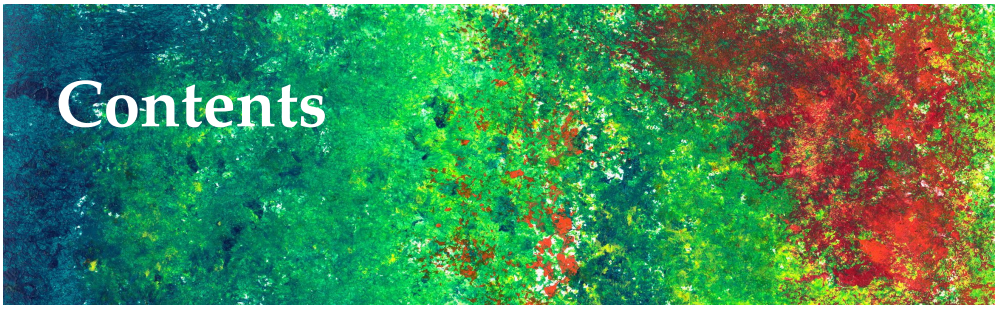
Flexible, Reliable Software:
Using Patterns and Agile Development
PreRelease - 2nd Edition
Part III
© 2021 – Henrik Bærbak Christensen

Henrik Bærbak Christensen
Version: 04-2021
Status: Release Candidate.

August 19, 2025

Excerpts from
Flexible, Reliable Software: Using Patterns and Agile Development
© Taylor and Francis, 2010.

Reprinted and revised by permission 2020



Contents

III The First Design Pattern **1**

7 Deriving Strategy Pattern **5**

7.1 New Requirements 5
7.2 One Problem – Many Designs 6
7.3 Source Tree Copy Proposal 7
7.4 Parametric Proposal 9
7.5 Polymorphic Proposal 13
7.6 Compositional Proposal 17
7.7 The Compositional Process 22
7.8 The Strategy Pattern 22
7.9 Summary of Key Concepts 23
7.10 Selected Solutions 24
7.11 Review Questions 24
7.12 Further Exercises 24

8 Refactoring and Integration Testing **27**

8.1 Developing the Compositional Proposal 27
8.2 Summary of Key Concepts 41
8.3 Selected Solutions 42
8.4 Review Questions 42
8.5 Further Exercises 42

9 Design Patterns – Part I **45**
9.1 The History of Design Patterns . . . 45
9.2 The Purpose of Patterns . . . 46
9.3 Patterns as a Communication Device . . . 47
9.4 The Pattern Template . . . 48
9.5 Summary of Key Concepts . . . 49
9.6 Review Questions . . . 50
9.7 Further Exercises . . . 50

10 Coupling and Cohesion **51**
10.1 Maintainable Code . . . 51
10.2 Coupling . . . 52
10.3 Cohesion . . . 53
10.4 Law of Demeter . . . 55
10.5 Summary of Key Concepts . . . 57
10.6 Selected Solutions . . . 57
10.7 Review Questions . . . 57
10.8 Further Exercises . . . 58

Bibliography **59**

Index **61**

Index of Sidebars/Key Points **63**

Copyrighted Material. Do Not Distribute.

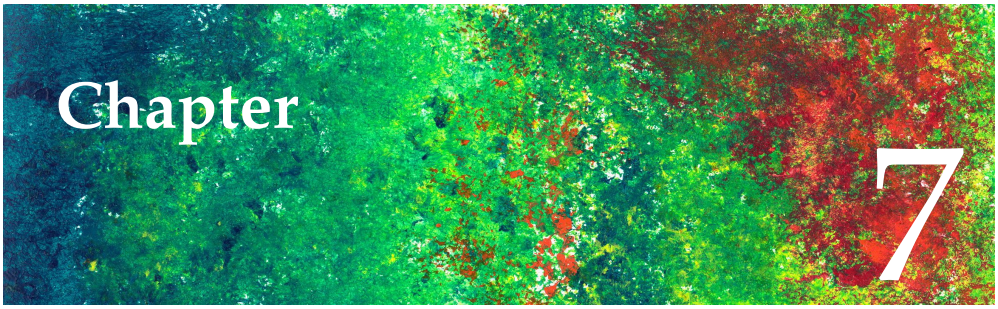
Iteration III

The First Design Pattern

This part of the book, *The First Design Pattern*, is a gentle introduction to design patterns. The path, however, is over how to handle variability in your production code when two different customers want *nearly* the same system but have minor divergences in requirements. I will also demonstrate how to introduce a design pattern by refactoring an existing design, which brings us around yet another area of testing.

Chapter	Learning Objective
Chapter 7	*Deriving Strategy Pattern.* You will learn four different techniques to handle variable behavior in the production code. Each of these are thoroughly analyzed. One of them, the compositional design approach, is an instance of the STRATEGY pattern.
Chapter 8	*Refactoring and Integration Testing.* We need to introduce the STRATEGY pattern in our production code and this chapter is about how to use TDD to do this refactoring. The refactored design allows me to introduce the concepts of integration testing and system testing.
Chapter 9	*Design Patterns – Part I.* This chapter is a historical account of patterns. It also discusses the writing template and key aspects that all patterns must describe.
Chapter 10	*Coupling and Cohesion.* Here the focus is two important metrics that correlate to design patterns and how maintainable your software is.

Copyrighted Material. Do Not Distribute.



Deriving Strategy Pattern

Learning Objectives

Our pay station system has successfully been deployed in Alphatown. However, a new customer, Betatown, wants to buy our system but has another requirement on how rates are calculated. The learning objective of this chapter is twofold. First you will learn several different solutions to this problem and see their respective benefits and liabilities. Second, you will learn that one of the solutions, the *compositional* one, is actually the STRATEGY pattern. You will also see a special process, that I denote the ③-①-② process, in action for the first time.

During my analysis a lot of new concepts and principles will be presented. Do not worry too much if you do not get all the fine points as I will return to these principles and concepts many times during the rest of the book.

7.1 New Requirements

The pay station software has been a great success. The Alphatown municipality is satisfied. Rumors start to spread about this fantastic piece of software and before long we are facing the nightmare of all developers: new requirements! We are contacted by the municipality of the neighbor town Betatown.

As is often the case they want *"exactly the same, but with one minor change."* They want another rate policy. Betatown has had problems with cars staying too long at the parking lots so they require a *progressive rate* according to the following scheme:

1. First hour: $1.50 (5 cents gives 2 minutes parking)
2. Second hour: $2.00 (5 cents gives 1.5 minutes)
3. Third and following hours: $3.00 per hour (5 cents gives 1 minute)

Note that this specification divides the time spent on the parking lot into three intervals in minutes, $[0; 60]$, $]60; 120]$, and $]120; \infty[$, in which the cost per minute change between 2.50, 3.33, and 5 cents. This way there is an economic benefit from using a parking lot as little as possible.

But—I am facing a challenge. I have to maintain and support software running in two different towns but most of the software's behavior is exactly the same. I am also faced with exciting business possibilities: why should it stop with only two towns? If I come up with a flexible solution that will allow me to provide whatever rate policy a town wants quickly and reliably, my product will clearly have a competitive edge!

> ☞ Spend some time considering how you would approach this problem before reading on. Do not try to come up with a single solution, but instead present as many different solutions as possible and for each consider their benefits and liabilities.

7.2 One Problem – Many Designs

Let me start by stating the problem more rigorously.

> **Problem (formulation 1):** We need to develop and maintain two variants of the software system in a way that introduces the least cost and defects.

This statement directly reflects my wish for reliable and flexible software. Looking closer at the code I discover that the part of the source code that must be changed is almost surgically small:

Fragment: chapter/tdd/iteration-9/PayStationImpl.java

```
1  private int timeBought;
2
3  public void addPayment( int coinValue )
4          throws IllegalCoinException {
5    switch ( coinValue ) {
6    case 5: break;
7    case 10: break;
8    case 25: break;
9    default:
10     throw new IllegalCoinException("Invalid coin: "+coinValue);
11   }
12   insertedSoFar += coinValue;
13   timeBought = insertedSoFar / 5 * 2;
14 }
15 public int readDisplay() {
16   return timeBought;
```

The only change in the Alphatown software to make it Betatown software is the single statement, line 13, that has to be replaced by another set of statements. Such a point in the source code that I need in two (or more) different variants is termed:

Copyrighted Material. Do Not Distribute.

> Definition: **Variability point**
> A variability point is a well-defined section of the production code whose behavior it should be possible to vary.

So, another way to formulate the problem is:

> **Problem (formulation 2):** How do I introduce having two different behaviors of the rate calculation variability point such that both the cost and the risk of introducing defects are low?

This question has many different possible answers. Here I will concentrate on four classes of solutions that are feasible in modern object-oriented languages and all widely used in practice.

1. *Source tree copy proposal:* I simply make a copy of all the source code. In one copy I maintain the Alphatown variant and in the other the Betatown variant.
2. *Parametric proposal:* I introduce a parameter that defines the town the software is running in: either Alphatown or Betatown. I only have one copy of the production code but at appropriate places (in my concrete case: in the rate calculation variability point) the code branches on this parameter to give the behavior required.
3. *Polymorphic proposal:* I can subclass the PayStationImpl and override a method that calculates the rate.
4. *Compositional proposal:* I can describe the variable behavior, rate calculation, in an interface and have classes implementing each concrete rate calculation policy. I then let the pay station call such a rate calculation object instead of performing the calculation itself.

I will treat each of these proposals in depth in the following sections.

7.3 Source Tree Copy Proposal

The probably oldest way of reusing code is based on the fact that software is expressed in *text*. Thus if a programmer faces a challenge that is very similar to something he has already solved, he can simply copy that source code text and modify it to solve the new challenge. This kind of software reuse is often called **cut'n'paste reuse**.

Thus, as I have the source code in a source code folder tree I can simply make a copy of that tree, rename the root of the tree to, say, Betatown, and overwrite the rate calculation code with the proper algorithm for the progressive rate. Figure 7.1 shows a plausible folder structure before and after a source code folder copy: the original Alphatown source code (a) is simply copied and renamed (b). As I develop code in a test-driven way, I will also copy the test cases. In my case, most of the test cases have to be rewritten because almost all of them rely on the rate policy, but in the general case, only a few test cases may have to be changed.

Copyrighted Material. Do Not Distribute.

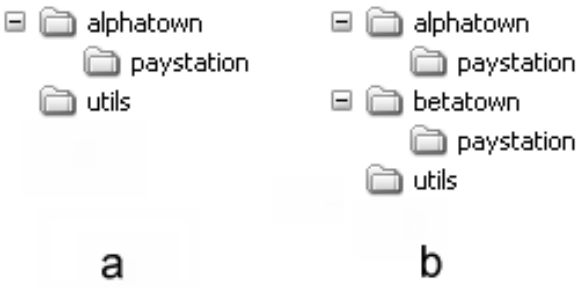


Figure 7.1: Source tree copy proposal.

This solution proposal has several obvious benefits.

- *Speed.* It is quick! It is a matter of a single copy operation, modify the test cases and let it drive the production code changes, and you are ready to ship it to the customer. This is probably *the* main reason that it is encountered so often in practice.
- *Simple.* The idea is simple and easy to explain to colleagues and new developers.
- *No implementation interference*. The two implementations do not interfere with each other. For instance, if you happen to introduce a defect in the Betatown implementation it is guaranteed that it will have no implications on the Alphatown implementation.

However, the proposal has some severe limitations. In the long run this proposal is often a real disaster because it leads to the **multiple maintenance problem**. To appreciate this problem consider a situation where I have also sold the pay station solution to some other customers, Gammatown and Deltatown and that they also only differ in the way parking time prices are calculated. Then I have four different source trees that only differ in the implementation of the behavior of single source code line number 13 in PayStationImpl. Now consider a situation in which you want to introduce a next generation pay station that can keep track of its earning: a new method amountEarned defines the responsibility of telling how much money a pay station has earned since it was last emptied.

> ☞ Spend a few minutes considering how you would introduce this new behavior in all four implementations before reading on.

There is no real easy way out of that: you have to implement the same test cases and exactly the same production code in all four source trees. This is repetitive and tedious at best, and highly prone to errors. If this is not bad enough, consider a situation where a defect is detected in this new behavior. Then the defect also has to be removed in all four code trees, one by one.

Practice shows that over time the contents of the code trees even begin to evolve into completely different directions, often because different people work on each copy and use slightly different techniques. Thus the commonality of the production code

Copyrighted Material. Do Not Distribute.

Sidebar 7.1: Source Tree Copying in SAWOS

The SAWOS system, introduced in sidebar 2.1, existed in several variants, one for each airport in Denmark (being a small country this means less than ten). I had invested great effort in avoiding the multiple maintenance problem on the production code that dealt with the core domain namely meteorological reports. Thus I came up with a design that allowed me to keep only one production code base and handle variations by adding classes. However, I found that the most easy way to handle the initialization and configuration code (including the main method) was to keep separate directories, one for each airport, and live with the copy problem in these few source files. The development environment we used and its idea of "projects" also made it natural to go this way.

However, SAWOS evolved and many programmers participated in its development. When I left the company the setup and configuration code had grown into about 20 source code files containing 425 KB code (roughly 13,000 lines of code) and these were now maintained in eight different copies. Any defect or new requirement that was realized by code in these parts had to be analyzed and potentially coded in each of the eight copies. It required extreme care to do this properly. I usually made a change work in one airport's software and when my manual testing told me it worked I would then copy the changes to the source code of the other airport. Even a simple mistake as misunderstanding what file from what airport I was looking at in the editor could introduce defects—you get quite dizzy of looking at nearly identical source code in eight copies!

replicas begins to erode and they drift apart. It becomes more and more difficult to apply the same defect removal techniques on all variants, and after a while it is more or less like maintaining a set of completely different applications: the analysis must be made anew for each copy even for the same defect! This of course makes the maintenance costs very high.

To sum up, the source tree copy proposal is quick but very dangerous. A short war story about source tree copy can be found in sidebar 7.1.

7.4 Parametric Proposal

Another solution is to use an `if` statement that selects the proper code block to execute based upon which city the code is running in. A parametric proposal represents a plausible and often used solution. Looking again over the price calculation code:

```
[...]
timeBought = insertedSoFar * 2 / 5;
```

it is obvious that we simply enclose it in an if statement where the code above gets executed in the Alphatown variant and something different in the Betatown variant.

As the requirement follows the town in question it is natural to invent a town parameter:

Copyrighted Material. Do Not Distribute.

```
public class PayStationImpl implements PayStation {
  [...]
  public enum Town { ALPHATOWN, BETATOWN }
  private Town town;

  public PayStationImpl(Town town) {
    this.town = town;
  }
  [...]
}
```

and then introduce the variability handling in the addPayment method (lines 11–15):

```
1   public void addPayment(int coinValue)
2           throws IllegalCoinException {
3     switch (coinValue) {
4     case 5:
5     case 10:
6     case 25: break;
7     default:
8       throw new IllegalCoinException("Invalid coin: "+coinValue);
9     }
10    insertedSoFar += coinValue;
11    if (town == Town.ALPHATOWN) {
12      timeBought = insertedSoFar * 2 / 5;
13    } else if (town == Town.BETATOWN) {
14      [the progressive rate policy code]
15    }
16  }
```

Exercise 7.1: Showing the solution before having written the test cases is of course not a TDD approach. Outline a plan of the iterations involved in a test-driven development process: What problem would you address first, what test cases would you look at and in which order? Next, use TDD to develop the parametric proposal.

A concrete pay station object of course has to have its town attribute set in order for it to behave correctly. Thus the setUp method must be rewritten in the JUnit test case:

```
  public void setUp() {
    ps = new PayStationImpl(PayStationImpl.Town.ALPHATOWN);
  }
```

Note that a parametric proposal can come in "many shapes", for instance you can parameterize a system in many different ways: accept a parameter as argument when starting the program, reading the parameter from a property file, accessing a key-value in the registry, reading from a database, using conditional compilation in C-type languages, etc., but they all basically follow the same principle as I have outlined here.

Copyrighted Material. Do Not Distribute.

7.4.1 Analysis

Analyzing this proposal reveals that it has several advantages. The benefits are:

- *Simple.* Conditionals are easy to understand and often one of the first language constructs introduced to new learners of programming. Thus the parametric idea is easy to describe to other developers.
- *Avoids the multiple maintenance problem.* There is only one code base to maintain for both Alphatown and Betatown. Thus a defect in common behavior can be fixed once and for all. As argued above most of the pay station's production code is identical for both towns and this will therefore make bug fixing less expensive and less prone to errors. Also new features for the pay station that both towns can benefit from is also less costly to develop as only one production code base is affected by added code and new tests.

The proposal, however, has also some liabilities most of which deal with long term maintainability. I will list them here and go through them in greater detail below.

- *Reliability concerns.* The new requirement is handled by *change by modification* that has a risk of introducing new defects.
- *Analyzability concerns.* As more and more requirements are handled by parameter switching, the production code becomes less easy to analyze.
- *Responsibility Erosion.* The pay station has, without much notice, been given an extra responsibility.

Reliability concerns. One liability of the parametric solution compared to the source tree copy proposal is that I have to *change in the existing production code* that is already released and operational in Alphatown. Thus there is a risk that I introduce defects in the software for Alphatown even though my intention is only to make the software run in Betatown as well. Defects are not introduced into production code by spontaneous creation—they are introduced because developers physically modify the source code text. As a logical consequence, the less I ever modify a class, the higher is the probability that it will remain free of defects. This is an important concept that I will call:

> Definition: **Change by modification**
>
> *Change by modification* is behavioral changes that are introduced by modifying existing production code.

My automated test cases, of course, reduce the risk of introducing defects compared to a situation where I am relying on manual, sporadic, or even no testing. Still, tests can only show the presence of defects, never that they are absent.

The main problem with the parametric proposal compared to the next two proposals, polymorphic and compositional, is that any new rate structure requirement forces me to modify the code over and over again. The problem is that *I must change pay station code every time a new rate model must be implemented.* That is, when Gammatown wants to buy our pay station but asks for a third kind of rate structure I have no options but to modify the implementation for the third time.

Copyrighted Material. Do Not Distribute.

```
[...]
if (town == Town.ALPHATOWN) {
  timeBought = insertedSoFar * 2 / 5;
} else if (town == Town.BETATOWN) {
  [BetaTown implementation]
} else if (town == Town.GAMMATOWN) {
  [GammaTown implementation]
}
```

When Deltatown wants to buy the pay station, I have to change the code the fourth time, and so on. Thus, any such new requirement is associated with the risk of introducing defects, and the necessary cost of avoiding them: testing, code reviewing, etc. The polymorphic and compositional proposals, described later, can to a large extent avoid this problem.

Analyzability concerns. Analyzability (see Chapter 3) is the capability of the software to be diagnosed for defects. This property is of course closely related to a developer's ability to understand the code that he or she reads. At this point in time, there is only a single *if-else* that is quite obvious. My concern is if the number of cases starts growing as I hinted at in the previous section. If I need to handle a large set of variants using switches, then it becomes difficult to "see the forest for all the trees." If the amount of variant switching code, all the ifs, outnumbers the code associated with the stated requirements, the rate calculations, then analyzability suffers greatly. This problem is commonly referred to as **code bloat**: code that is unnecessarily difficult to read, long, or slow. I will return to this point in later chapters when the number of variants has grown.

Responsibility Erosion. The last problem is not as much a code issue as it is an issue regarding how we design and "think" software. One major problem with the parametric proposal is that it sneaks in another responsibility into our pay station. I will discuss the concept of "responsibility" at great length in later chapters but let us just think of responsibility as "something that the pay station is required to do."

The original Alphatown pay station's responsibilities are described in the interface and designed as:

PayStation (as-designed)

- Accept payment
- Calculate parking time
- Handle transactions (buy, cancel)
- Know time bought, earning
- Print receipt

But looking over our parameterized code we discover that another responsibility has sneaked in without much attention:

PayStation (as-built)

- Accept payment
- Calculate parking time
- Handle transactions (buy, cancel)
- Know time bought, earning
- Print receipt
- **Handle variants of the product**

Copyrighted Material. Do Not Distribute.

Sidebar 7.2: Conditional Compilation in GCC

The GNU Compiler Collection (GCC 2009) is a collection of open source compilers for a number of languages, most prominent C and C++ as they were the origin of the project. GCC is able to compile the set of supported languages to a large set of operating systems and CPUs: it must cope with a large set of variations. GCC is written in C and has inherited the C tradition of handling variations by using preprocessor directives. For instance if a certain variable is only interesting for some compiler variant then you can set a variable like `HAVE_cc0` to either true or false and then the preprocessor will feed the compiler with source code depending on switches in the code, like:

```
static int
combine_instructions (rtx f, unsigned int nregs)
{
  rtx insn, next;
#ifdef HAVE_cc0
  rtx prev;
#endif
```

will either have or not have the declaration `rtx prev;` compiled into the executable.

I did a count on release 4.1.1 of GCC and there are a total of 4079 lines with an #ifdef in the source code. The `HAVE_cc0` alone is switched upon in 83 places in 22 different source files.

A study by Ernst et al. (2002) showed that about 8.4% of the source code lines on average in 26 Unix software packages written in C were preprocessor directives.

Thus, the pay station now "does something more" than it was originally designed for. Just as with code bloat this problem also has a tendency to sneak in on your design without you really noticing. But it is a bad tendency because it drives your design towards a procedural design where a large class seems to suck up all functionality in the system as it does all the decision making and processing. This phenomenon is also often met in practice—it is called **The Blob** or *The God class* (Brown et al. 1998). Such classes are difficult to understand, to modify, and to reuse. My advice is to keep track of the responsibilities each object has, keep the number small, and be very careful about introducing code that sneaks in additional responsibilities.

7.5 Polymorphic Proposal

The underlying theme of this book is the object-oriented paradigm, so it is obvious that I must consider using one of the most prominent features of OO languages: polymorphism and subclassing.

Subclassing lets me define a subclass and override methods. Thus to introduce a new way of calculating the rate I can override the addPayment method. Unfortunately, this would also force me to duplicate the coin type checking code and I have already discussed the negative consequences of the multiple maintenance problem. Therefore my proposal is to encapsulate the rate calculation by invoking a protected method,

Copyrighted Material. Do Not Distribute.

calculateTime (see Figure 7.2 and line 11 in the listing below). This method (lines 18–20) can then be overridden in a subclass PayStationProgressiveRate.

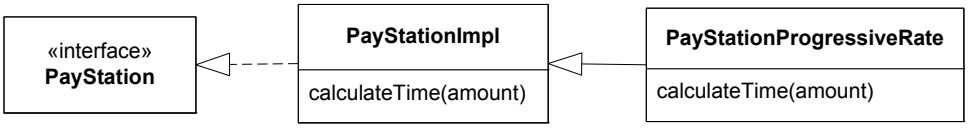


Figure 7.2: Subclassing proposal for new rate calculation.

```
1  public void addPayment(int coinValue)
2          throws IllegalCoinException {
3    switch (coinValue) {
4    case 5:
5    case 10:
6    case 25: break;
7    default:
8      throw new IllegalCoinException("Invalid coin: "+coinValue);
9    }
10   insertedSoFar += coinValue;
11   timeBought = calculateTime(insertedSoFar);
12 }
13 /** calculate the parking time equivalent to the amount of
14     cents paid so far
15     @param paidSoFar the amount of cents paid so far
16     @return the parking time this amount qualifies for
17 */
18 protected int calculateTime(int paidSoFar) {
19   return paidSoFar * 2 / 5;
20 }
```

Note that this is a refactoring. The external behavior has not changed but the internal structure has improved with respect to the upcoming addition of code to handle an additional rate structure. Once I get this Alphatown refactoring working as indicated by all test cases passing, I can implement the Betatown subclass.

Exercise 7.2: Outline a plan over iterations for a test-driven development of the polymorphic proposal. Next, develop the polymorphic proposal using TDD.

7.5.1 Analysis

Analyzing this proposal shows me that it is an improvement in many ways compared to the parametric proposal. Below I summarize benefits and liabilities for easy reference before explaining the details. To summarize the benefits, they are:

- *Avoid multiple maintenance problem.* I only have one code base which is a good property as already argued.
- *Reliability concern.* The production code has once and for all been prepared for any new rate policy to be required later.

Copyrighted Material. Do Not Distribute.

- *Code analyzability.* This proposal does not suffer from code bloat, you have to read less code to understand a variant.

But in summary the proposal has a fair number of liabilities:

- *Increased number of classes.* Every new rate policy will introduce a new subclass.
- *Inheritance relation spent on single type of variation.* I cannot use the inheritance relation to handle other types of variation.
- *Reuse across variants difficult.* It turns out to be cumbersome to reuse the algorithms defined by one variant in another.
- *Compile-time binding.* The binding between the particular rate policy and the pay station is defined at compile time.

Let us look at these benefits and liabilities in more detail.

Reliability concern. As with the parametric solution I have to modify the production code: I have to introduce both a definition and a call of the calculateTime method, and I have to move code from the addPayment method into the new calculateTime method. The question is then: is the polymorphic solution just as bad as the parametric? The answer is *no.* There is namely one big difference. The modification can be considered the *last* with regards to new requirements to rate calculation. To see why, consider that Gammatown requires a third rate policy, different from the two I have already implemented.

> ☞ Take a moment to consider how you would implement the 3rd unique rate policy requirement in the polymorphic proposal.

The advantage is that it can be implemented *without* any changes to class PayStationImpl. It only requires *adding a new subclass.* And this applies to all new rate structure requirements that we can think of: Any new requirement adds a class, it does not modify any existing ones. I will term this way of handling change:

> Definition: **Change by addition**
> *Change by addition* is behavioral changes that are introduced by adding new production code instead of modifying existing.

Change by addition is important, because I have no fear of introducing defects in existing products as they do not use any of the code I have added. Consider for a second that you have a large system that can be configured for 50 different product variants. Next consider that you must add a variant number 51. Which strategy would make you feel most confident that the existing 50 variants still behave correctly: Strategy 1: by adding a new class (which, of course, is only used in the new variant 51), or strategy 2: by modifying several existing classes? Well, strategy 1 has no risk of introducing defects in existing products while strategy 2 certainly has...

Another way of characterizing *change by addition* that you may come across is the **open/closed principle**. This principle says that software should be open for extension

Copyrighted Material. Do Not Distribute.

but closed for modification. Meyer (1988) is generally credited as having originated the term, however his focus (being in the golden days of object-oriented inheritance) was on the polymorphic approach.

Analyzability. The polymorphic proposal does not suffer from code bloat. In contrast to the parametric proposal where new rate requirements lead to ever increasing conditionals inside the same class the structure of addPayment is defined once and for all. Thus when I in the future get the 43rd unique requirement for a rate structure I can safely make a new variant without touching any of the code that is currently running in the pay stations of Alphatown.

Also, the change the new requirement has introduced is nicely localized in a single class which makes it easy to spot. As the two products are now handled by separate classes instead of by conditional statements there is no code bloat.

However, the proposal also has its fair share of liabilities:

Increased number of classes. If there is a large number of rate algorithms that I must support, it will directly lead to a large number of classes—one for each rate policy. Thus the developer has to overview a large set of classes in his development environment that could potentially be overwhelming and confusing, especially to new developers on the project. You may argue that I have traded *complexity in the code* with *complexity in the class structure*.

Inheritance relation spent on single type of variation. Java and C# only support single inheritance of implementation: a class can only have one superclass. Here I have "wasted" this scarce resource on one particular type of variation, namely variations in the rate structure. This is obvious from the somewhat odd name that I gave the subclass: PayStationProgressiveRate. It sounds a bit disturbing, does it not? I have coded the theme of variation, rate structure, directly into the class name???

Customers often demand variations across different aspects of the product. Maybe some customers want another type of information printed on the receipt, maybe some request that the parking expiration time is displayed instead of minutes parking time bought, maybe others require that each buy transaction is logged into a database, etc. Supporting many customers' different requirements may lead to a subclass called PayStationProgressiveRateDatabaseLoggingHourMinuteDisplayOptionMobilePhonePayment. Of course, this sounds like a joke, but the point is that single inheritance is not a good option for supporting several types of variations. I will return to this problem in the next section, and give a theoretical treatment of it in Chapter 16.

Reuse across variants difficult. The polymorphic solution also begins to fail when it comes to making combinations of previously coded solutions. Consider that Gammatown wants to buy our pay station and want a rate structure similar Alphatown's during weekdays but similar to Betatown's during weekends. The problem is that I have already written the two rate calculation algorithms—some of it in the original PayStationImpl and some in PayStationProgressiveRate and I of course do not want to duplicate code as it leads to the multiple maintenance problem.

Exercise 7.3: Consider how you would support this demand using the polymorphic proposal.

Copyrighted Material. Do Not Distribute.

The bottom-line is that it is actually a bit cumbersome to support this demand and any solution will require quite a bit of refactoring and modification to the existing source code. I will treat this problem in detail in Chapter 11.

Compile-time binding. The relation between a superclass and a subclass is expressed explicitly in the source code: I directly write

```
public class PayStationProgressiveRate extends PayStationImpl
```

This means that the object instance that results from a `new PayStationProgressiveRate()` can never over its lifetime change the way that it calculates the rate. *It cannot change behavior at run-time.*

> ☞ Is this also the case for the source code copy and parametric proposals?

Consider a request from our Betatown costumer that they are disappointed with the rate structure and prefer to use that of Alphatown after all. If I look at the objects that actually execute out in the pay stations then in the parametric solution I could actually make this change at run-time: all it takes is to set the town variable in the pay station object to a new value. This is not so for the polymorphic proposal: the object can never change behavior. If I insist on a run-time change, I would have to delete the object and replace it with a new instance with the PayStationImpl class. The problem may seem a bit absurd for the pay station case. Most likely the software will be shut down for maintenance in order to make such a change anyway. However, in many other types of software the ability to change behavior while running is desirable.

The bottom-line is that inheritance is a static binding that is inflexible when it comes to making behavioral changes while programs execute. If this is not important, I of course have no problem in this respect. However, often it *is* important that software can change behavior while executing.

7.6 Compositional Proposal

The fourth proposal I denote *compositional* as it composes behavior: it lets objects *collaborate* so their *combined* effort provide the required behavior—instead of letting a single abstraction struggle with doing it all by itself as was the case with both the parametric and polymorphic solution. This "combined effort" viewpoint is one of the primary aims of this book and requires a special mind set when designing software and thinking in terms of software. However, as I will argue in this section, once it is understood and applied it is extremely powerful and will help you design very flexible object-oriented software.

But—I will start the analysis in a somewhat different place. Let us look at the *responsibilities* that our original pay station has.

Copyrighted Material. Do Not Distribute.

PayStation

- Accept payment
- Calculate parking time based on payment
- Know earning, parking time bought
- Print receipts
- Handle buy and cancel transactions

Any class implementing the PayStation interface is free to provide concrete behavior as long as these responsibilities are met. We can see that all the rate requirements I have introduced so far only deals with one of the responsibilities on the list, namely the item: *Calculate parking time based on payment.* Neither Alphatown, Betatown, nor Gammatown have defined requirements that require me to change behavior associated with the four other responsibilities: *Accept payment*, *Know earning, parking time bought*, etc.

This important analysis tells me why the source-tree copy and parametric proposals have the property that I have to modify code instead of add code: as the code that handles the parking time calculation responsibility is completely handled by the very same abstraction that also serves the other responsibilities I have no other option but to modify it.

So: What do we do about that? The compositional answer is to *divide the responsibilities over a set of objects and let them collaborate*. That is: I move the responsibility *Calculate parking time based on payment* away from the pay station abstraction itself and put it into a new abstraction whose only purpose in life is to serve just this one responsibility. I will describe this abstraction and its responsibility by an interface and I end up with the UML class diagram in Figure 7.3 where the responsibilities are marked in the class box. Thus the variability point will look something like (the code will be treated in detail in the next chapter):

```
[...]
insertedSoFar += coinValue;
timeBought = someOtherObject.calculateTime(insertedSoFar);
```

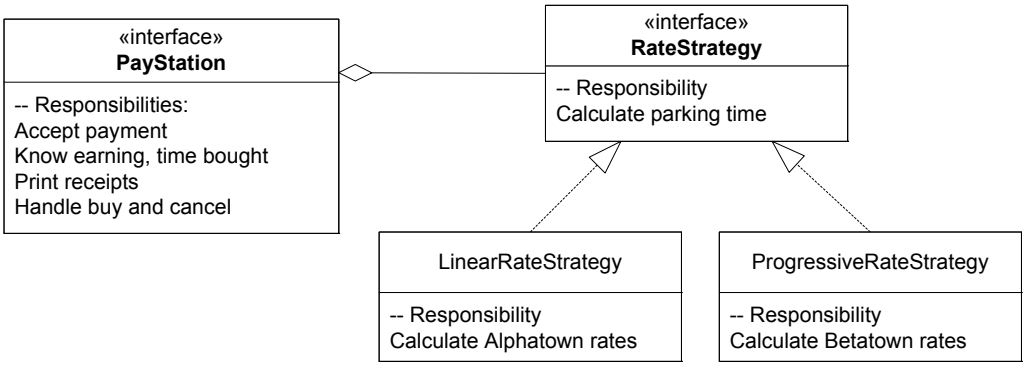


Figure 7.3: Dividing the responsibilities between two abstractions.

The Alphatown variant will use one particular implementation (LinearRateStrategy) of this interface that implements the old linear rate policy, while the Betatown variant will use another (ProgressiveRateStrategy) that implements the new rate requirements. The interaction between the pay station object and the rate strategy object

Copyrighted Material. Do Not Distribute.

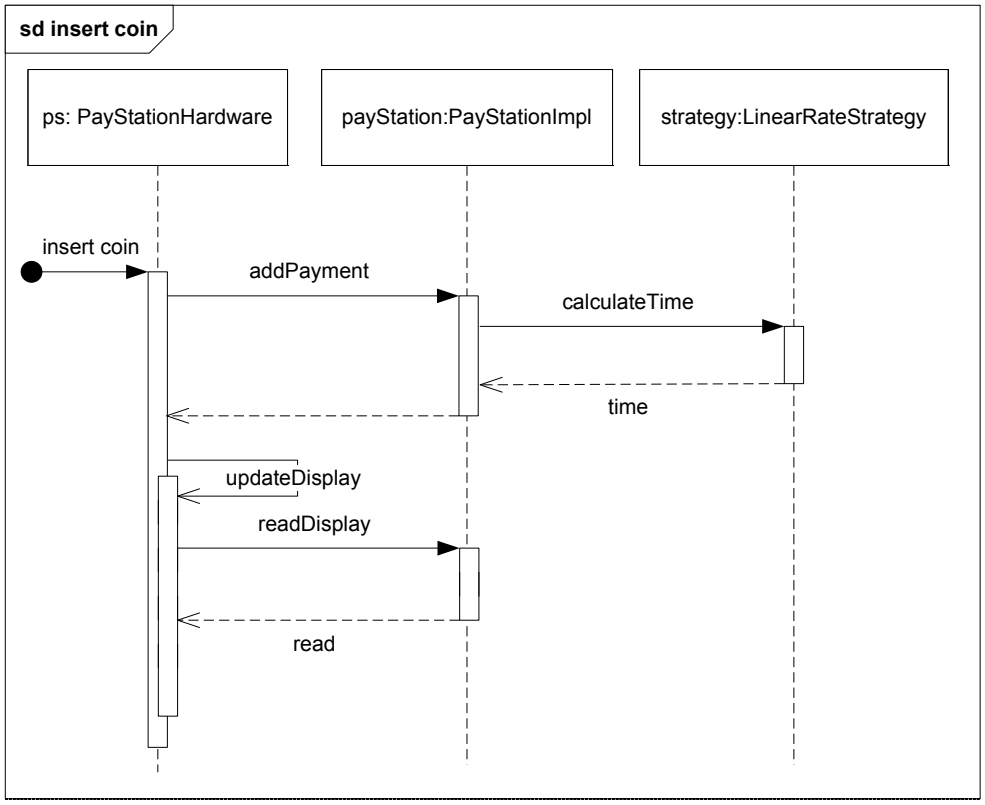


Figure 7.4: Delegating rate calculation to a delegate.

is seen in Figure 7.4. The overall behavior of the pay station when accepting coins becomes a *combined* effort between the pay station implementation and the object implementing the rate calculation. That is, instead of one object being involved in handling a request, two objects collaborate to achieve the desired behavior. This is called delegation.

Definition: **Delegation**

In delegation, two objects collaborate to satisfy a request or fulfill a responsibility. The behavior of the receiving object is partially handled by a subordinate object, called the **delegate**.

In the pay station, rate calculation is delegated to the rate calculation object. I often call this delegation based way of getting work done for *let someone else do the dirty job!* I have called the interface RateStrategy because objects implementing it are defining a certain strategy for calculating rates. As you will see shortly, there is also another reason for the name.

Copyrighted Material. Do Not Distribute.

7.6.1 Analysis

The compositional proposal has a number of nice properties with regards to reliability and flexibility. I will outline benefits and liabilities in an overview form here for easy reference and discuss each point in depth below.

Benefits:

- *Reliability.* The pay station has been refactored once and for all with respect to rate calculations, and new rate structures can be handled only by adding more classes.
- *Run-time binding.* The binding between the pay station and its associate rate calculation can be changed at run-time.
- *Separation of responsibilities.* Responsibilities are clearly separated and assigned to easily identifiable abstractions in the design.
- *Separation of testing.* As the responsibilities have been separated I can actually test rate calculations and core pay station functionality independently.
- *Variant selection is localized.* The code that decides what particular variant of rate calculation to use is in one spot only.
- *Combinatorial.* We can introduce other types of variability without interfering with the rate calculation variability.

Liabilities:

- *Increased number of classes and interfaces.* The number of classes and interfaces has grown in comparison to the original design.
- *Clients must be aware of strategies.* The selection of which rate policy to use is no longer in the pay station but still someone has to make this decision. Variant selection is moved to the client objects.

Reliability. The compositional proposal has the property that new functionality is defined by *change by addition* instead of *change by modification*. Thus the advantages that I described in the analysis of the polymorphic proposal are valid here as well.

Run-time binding. The compositional proposal defines a run-time binding between the pay station and its rate calculation behavior: if we want a particular pay station instance to perform rate calculations in another way it is simply a matter of changing the object reference rateStrategy—just as simple as changing the value of the town variable in the parametric proposal. Note, however, that this is possible because the rate strategies do not store any state information. In the general case, strategies can be changed at run-time if they are stateless.

Exercise 7.4: Sketch the design changes and implementation changes that are necessary in order to allow the pay station to change rate structure while it is executing.

Copyrighted Material. Do Not Distribute.

Separation of Responsibilities. Responsibilities are clearly separated and expressed in interfaces. This has several implications.

First, it counters the tendency towards "The Blob". Instead of one object trying to get everything done by itself, two objects have divided the work into manageable parts and each take on a more specific assignment: *"You make the rate calculations, and I will handle accepting coins and making the transaction."*

Second, it is clearer where a defect is located. Defects *do* make it into the final product and when those bug reports start ticking in from the users, a clear division of responsibilities is the first guideline for locating it: *"They say the rate calculation is wrong, thus it must be in the RateStrategy implementation there is a defect"* or *"No problem with the rate calculations, so the defect cannot be in the RateStrategy implementation."*

Third, it provides more readable and navigable source code when responsibilities are clearly stated and expressed. You know what kind of behavior is associated with interfaces and implementing classes—and these are distinct objects in your class browser and file structure. Contrast this to the parametric proposal where *all* behavior—variant selection, coin acceptance, rate calculation—is mixed up in the same class.

Separation of testing. I now have two interfaces with two implementing classes. This allows me to test the two implementations independently. I will elaborate on this point in detail in Chapter 8.

Variant selection in one place only. There is no place in the source code that deals with deciding the rate policy to use—the pay station object simply delegates rate calculations to whatever RateStrategy object it is configured with. Contrast this with the parametric proposal where there is code associated with the decision both at the initialization point (the town parameter in the constructor call) and in the pay station code (the conditionals in the rate calculation code).

You say that the decision is **localized** in the code. The opposite situation, when the decision is not localized, is problematic as you have to look in many different places in your code to make a change, and chances are that you overlook one or a few of these places. It is also difficult to ensure that a change is applied consistently.

I also note that the decision of what rate calculation to use is made during the initialization of the pay station object, that is, early in the program's lifetime and "near" the program's main method. This way all the setup of the system is grouped in the same part of the code. This also leads to code that is more easy to read and understand.

Combinatorial. I have not used implementation inheritance in the compositional proposal. Thus the generalization/specialization relation is still "free" to be used. The main benefit of the combinatorial proposal is that it does not get in the way of varying other types of behavior. I can apply the same compositional technique to vary, say, the coin values that are accepted. I will return to this important property in depth later in the book.

There are some liabilities as well.

Increased number of classes and interfaces. A concern is the introduction of a new interface, RateStrategy, as well as the two implementing classes. Thus the number of source files that a developer has to overview grows which has a negative impact on analyzability. I will discuss how to counter this effect in Chapter 18.

Copyrighted Material. Do Not Distribute.

Clients must be aware of strategies. If the selection of which concrete RateStrategy instance to use is made in the pay station object itself, then this solution is no better than the parametric one: you end up with conditional statements in the pay station production code itself. Thus the pay station object must be told which rate strategy object to use. Thus the client object (typically the one that instantiates the pay station object) has to know about RateStrategy. This can also make it more difficult for a developer to overview the system.

7.7 The Compositional Process

The last proposal, the compositional proposal, is actually an example of using the STRATEGY pattern. Before I describe STRATEGY, let me sum up the line of reasoning that lead to it. The argumentation went along these lines:

- *I identified some behavior that varied.* The rate calculation behavior of the pay station is variable depending on which town the station is located in. Furthermore I can expect this behavior to be variable for new customers that buy the pay station.
- *I stated a responsibility that covered the variable behavior and encapsulated it by expressing it as an interface.* The RateStrategy interface defines the responsibility to *calculate parking time* by defining the method calculateTime.
- *I get the full pay station behavior by delegating the rate calculation responsibility to a delegate.* Now the pay station provides its full behavior as a result of collaboration between itself and an object specializing in rate calculations, either an instance of LinearRateStrategy or ProgressiveRateStrategy.

I will call this three step line of reasoning the ③-①-② process. The numbers refer to three principles for flexible design that were originally stated in the introduction to the first book on design patterns, *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al. 1995). In this book the principles are numbered 1–3 but they are applied in another order: The ③-①-② process first applies the third principle (find variability), then the first (use interface), and finally the second (delegate), hence the reason that I have chosen this odd numbering. I will discuss these principles in great detail in Chapter 16.

This collaboration mind set is a fruitful one in object-oriented design and is actually familiar to everybody working in a company, a public institution, or living in a family. Work is done by coordinating the collaboration of different people with different competences and skills. One person cannot perform all tasks but has to delegate responsibility and work to others. A person having too many concrete tasks to do leads to stress, making errors, and poor performance. *Objects are no different...*

7.8 The Strategy Pattern

The ③-①-② process and the ideas of identifying, encapsulating, and delegating variable behavior has resulted in a design with a number of desirable properties—and

Copyrighted Material. Do Not Distribute.

some liabilities of course. The resulting design is actually an example of the design pattern STRATEGY. STRATEGY is a pattern that addresses the problem of encapsulating a family of algorithms or business rules, and allows implementations of these algorithms to vary independently from the client that uses them. In our case, the business rule is calculation of rates.

The STRATEGY pattern's properties are summarized in the design pattern box 7.1 on page 26. The format used is explained in more detail in Chapter 9. An important aspect of any design pattern is the list of benefits and liabilities of using it to solve your particular design problem. The design pattern box only lists a few keywords but the discussion above is of course the comprehensive version of this.

7.9 Summary of Key Concepts

Often behavior in software systems must come in different variants. The requirements for these variants may stem from customers that have special needs (like the new customer of our pay station system), from us wanting to be able to run (and sell) a system on various operating systems or using various hardware and software configurations, or from the development team itself that needs to execute the system in "testing mode" and/or without actual hardware connected. The points in the production code that must exhibit variable behavior in different variants are called *variability points*. Variants can be handled in different ways but they are all basically variations over four different themes:

- *Source code copy solution.* You copy parts of or the entire software production code and simply replace the code in the variability points.
- *Parametric solution.* You enter conditional statements around the variability points. The conditions branch on a configuration parameter identifying the configuration.
- *Polymorphic solution.* You encapsulate the variability points in instance methods. These can then be overridden in subclasses, one for each required variant.
- *Compositional solution.* You encapsulate the variability points in a well-defined interface and use delegation to compose the overall behavior. Concrete classes, implementing the interface, define each variant's behavior.

Often compositional solutions arise from a line of design thinking that is called the ③-①-② process in this book. It consists of three steps: first you *identify some behavior that needs to vary.* You next abstract the behavior into *a responsibility that covers the variable behavior and express it as an interface.* Finally, you *use delegation to compose the full behavior.*

The STRATEGY pattern is a compositional solution to the problem of supporting variations in algorithms or business rules. The algorithm is encapsulated in an interface and the client delegates to implementations of this interface.

The first comprehensive overview of design patterns was the seminal book by Gamma et al. (1995). The *intent* sections of design pattern boxes in this book are in most cases literate copies from this book and reproduced with permission. The ③-①-② process is inspired by Shalloway and Trott (2004).

Copyrighted Material. Do Not Distribute.

7.10 Selected Solutions

Discussion of Exercise 7.3:

The problem is that there are no elegant solutions! I will discuss a set of potential solutions in detail in Chapter 11 but all of them turn out to be rather clumsy.

Discussion of Exercise 7.4:

As the calculation of rates is delegated to an instance implementing the RateStrategy interface, all you have to do is to change the stored reference to refer to a new instance that implements another rate calculation algorithm.

7.11 Review Questions

Outline the four different proposals to support two products with varying rate structure: What is the idea, what coding is involved? Explain what a variability point is.

What are the benefits and liabilities of the source tree copy proposal? The parametric proposal? The polymorphic proposal? The compositional proposal?

What is *change by addition*? What is *change by modification*?

What are the three steps, ③ ① and ②, involved in the compositional proposal process?

What is the STRATEGY pattern? What problem does it address and what solution does it suggest? What are the objects and interfaces involved? What are the benefits and liabilities?

7.12 Further Exercises

Exercise 7.5:

The reputation of our reliable and flexible pay station has reached Europe and Denmark and a Danish county wants to buy it. However, it should accept Danish coin values and of course use an appropriate rate structure. The requirements are:

- Danish coins have values: 1, 2, 5, 10, and 20 Danish kroner.
- 1 Danish krone should equal 7 minutes parking time.

1. Use the ③ ① ② process to identify and design a compositional solution to the coin type aspect that needs to vary in our pay station product.
2. Another Danish county wants also to buy our system but they want another rate policy: 1 Danish krone should equal 6 minutes parking time, however if the car stays for more than 2 hours then a krone only buys 5 minutes parking time. Analyze if this requirement will affect any code that was introduced by the above requirement. Can the two types of requirements, rate policy and coin validation, be varied independent of each other?

Copyrighted Material. Do Not Distribute.

Exercise 7.6:

A software shop wants to produce a calendar application for the international market. One important requirement is the ability for the calendar to mark public holidays for the given country in the calendar, for instance by giving public holidays a different background color in a graphical week overview.

Sketch how this requirement could be handled by the four different proposals for handling variability. Consider that the calendar must handle at least US, UK, French, and Danish public holidays, and allow easy integration of other nations' public holidays.

Copyrighted Material. Do Not Distribute.

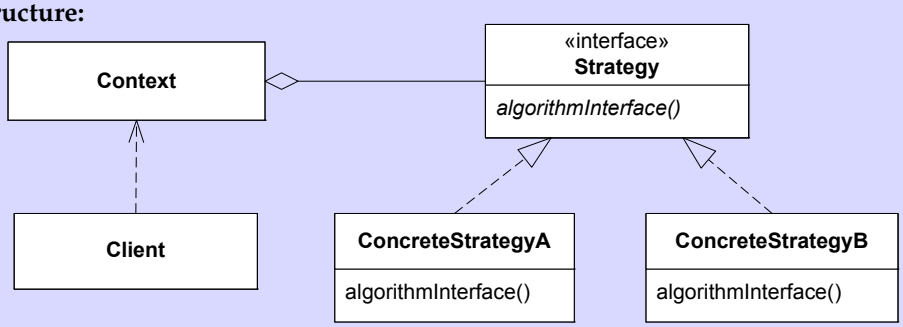
[7.1] Design Pattern: Strategy

Intent Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.

Problem Your product must support variable algorithms or business rules and you want a flexible and reliable way of controlling the variability.

Solution Separate the selection of algorithm from its implementation by expressing the algorithm's responsibilities in an interface and let each implementation of the algorithm realize this interface.

Structure:

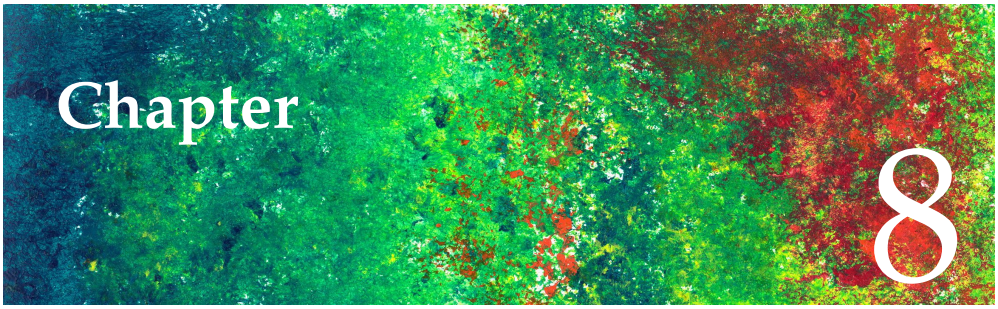


Roles **Strategy** specifies the responsibility and interface of the algorithm. **ConcreteStrategies** defines concrete behavior fulfilling the responsibility. **Context** performs its work for **Client** by delegating to an instance of type **Strategy**.

Cost - Benefit The benefits are: *Strategies eliminate conditional statements.* It is an *alternative to subclassing.* It facilitates *separate testing* of **Context** and **ConcreteStrategy**. Strategies may be changed at run-time (if they are stateless).

The liabilities are: *Increased number of objects. Clients must be aware of strategies.*

Copyrighted Material. Do Not Distribute.



Refactoring and Integration Testing

Learning Objectives

Updating the pay station design with the STRATEGY pattern also pose a challenge. In this chapter you will learn a *small steps* approach to refactoring as well as see the *Triangulation* principle in action on a more complex problem. As the system is now composed of three parts, two rate strategies and one pay station, you will also learn about the different levels in testing and the difference between unit and integration testing in particular.

8.1 Developing the Compositional Proposal

How should I introduce the Betatown rate structure? At the onset I have two options.

- Write test cases that tests the behavior of Betatown rate structure and get the new compositional design as well as the rate calculation algorithm in place.
- Refactor the existing pay station to use a compositional design first for Alphatown. Next write test cases and algorithm for the Betatown rate structure.

The mantra of TDD is *take small steps* because all too often we overestimate the steps we can take without falling. The first approach is a single, large, step while the second is two, smaller, steps. TDD tells me to follow the latter strategy.

So I define an initial test list

- ✻ refactor Alphatown to use a compositional design
- ✻ handle rate structure for Betatown

8.1.1 Iteration 1: Refactoring

Thus the challenge facing me is to refactor the pay station design in a way that I am confident that I have not introduced any defects in the process.

The important punch line here is: *I already have the Alphatown test cases* that can minimize the risk that I accidentally introduce defects while refactoring the design. The JUnit test cases for Alphatown's rate structure are written without any assumptions of the underlying pay station implementation—they only assume the PayStation interface.

I identify the following plan of steps that I will likely pursue:

i Introduce the RateStrategy interface.

ii Refactor PayStationImpl to use a reference to a RateStrategy instance for calculating rate (test to see it fails.)

iii Move the rate calculation algorithm to a class implementing the RateStrategy interface.

iv Refactor the pay station setup to use a concrete RateStrategy instance.

Step i) Introduce a RateStrategy interface: I introduce a suitable interface RateStrategy that expresses the responsibility for calculating parking time:

Listing: chapter/refactor/iteration-1/src/main/java/paystation/domain/RateStrategy.java

```
package paystation.domain;
/** The strategy for calculating parking rates.
*/
public interface RateStrategy {
  /**
   return the number of minutes parking time the provided
   payment is valid for.
   @param amount payment in some currency.
   @return number of minutes parking time.
   */
  public int calculateTime( int amount );
}
```

Step ii) Refactor to delegate to a RateStrategy object: I play it safe and start by running the old test cases. JUnit reports all tests pass.

The TDD rhythm is a bit different here, because we do not start by adding a new test but by modifying the production code—still the movement from the failed to passed tests is essential.

Now a RateStrategy instance has to perform the parking time calculation in the pay station—thus I introduce a reference as an attribute in the PayStationImpl:

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

  /** the strategy for rate calculations */
  private RateStrategy rateStrategy;
  ...
```

Copyrighted Material. Do Not Distribute.

and modify the addPayment method in its rate calculation variability point (line 11):

```
1  public void addPayment(int coinValue)
2         throws IllegalCoinException {
3    switch (coinValue) {
4    case 5:
5    case 10:
6    case 25: break;
7    default:
8      throw new IllegalCoinException("Invalid coin: "+coinValue);
9    }
10   insertedSoFar += coinValue;
11   timeBought = rateStrategy.calculateTime(insertedSoFar);
12 }
```

This change does compile but now most tests fail. I of course have to create a concrete RateStrategy instance somewhere. I can do this in numerous ways, but let us start by discussing two approaches: a) define a setRateStrategy method in the PayStation and call it with an appropriate object reference in the test setup; b) add a RateStrategy parameter to the pay station constructor and set this when constructing the pay station in the test setup.

Option a) is the more flexible one as it allows us to invoke the setRateStrategy even at run-time to change behavior of the pay station. However, it has two major drawbacks. First, a programmer could forget to invoke the method and thus leave the pay station with a null reference which makes it vulnerable to defective behavior. Second, it requires us to add another method to the PayStation interface—a method that has little to do with the pay station's stated responsibilities. Option b) on the other hand *forces* a programmer to consider the RateStrategy parameter: if he or she forgets it, the compiler will complain! It is always better to make the compiler check things for us rather than rely on our own discipline. The downside of the constructor approach is that we cannot change the RateStrategy reference after the pay station has been created. By combining the two approaches the best of both worlds can be made: Ensuring a valid RateStrategy as well as allow changes at run-time. However, we will introduce option b) only to keep the code short.

I change the setup in the test case class:

```
public void setUp() {
  ps = new PayStationImpl(new LinearRateStrategy());
}
```

and I have to change the constructor

```
/** Construct a pay station.
    @param rateStrategy the rate calculation strategy to use.
*/
public PayStationImpl(RateStrategy rateStrategy) {
  this.rateStrategy = rateStrategy;
}
```

Compiling it of course complains that there is no LinearRateStrategy. So, I apply *Fake It* by quickly defining a stub:

Copyrighted Material. Do Not Distribute.

```
public class LinearRateStrategy implements RateStrategy {
  public int calculateTime(int amount) {
    return 0;
  }
}
```

Now it compiles but of course the test fails. The final piece of the puzzle is to change the stub LinearRateStrategy into a real implementation: the old rate calculation code is put into place.

```
public class LinearRateStrategy implements RateStrategy {
  public int calculateTime(int amount) {
    return amount * 2 / 5;
  }
}
```

Tests pass—Done! I have now completed the refactoring: the external behavior is the same as before as evident by the running test cases, but the internal structure of the production code has been altered and is now much better prepared for the Betatown rate structure requirement. This leads to a key point when a new feature or requirement to an existing system requires a design change:

> **Key Point: Refactor the design before introducing new features**
>
> *Introduce the design changes and refactor the system to make all existing test suites pass* before *you begin implementing new features.*

If you try to develop the new features *and* the design changes required to support them in a single iteration, then you *lose focus* and end up *taking too large steps.* During the refactoring, the existing test cases are your lifeline that tells you when your refactoring is a success.

> **Key Point: Test cases support refactoring**
>
> *Refactoring means changing the internal structure of a system* without changing its external behavior. *Therefore test cases directly support the task of refactoring because when they pass you are confident that the external behavior they test is unchanged.*

This is an important point and a major reason that developers are convinced to make the extra effort of test-first development: the investment in the test cases pays back many fold when you dare making even radical changes of your software system.

Returning to the plan I laid out—what happened to *step iii)* and *step iv)*? Reviewing the steps I envisioned I now see that I took another path: By using *Fake It* I actually took *step iv)* before *step iii)*. You will often experience that opportunities will present themselves as you work that leads to better paths. It is like rock-climbing: You look at the rock and decide a feasible path but it is during the climb that you can really assess whether the plan is good or needs refinement. The point is that the plan is still important to make in advance but should not dictate every move as long as you work towards the goal.

- ~~refactor Alphatown to use a compositional design~~
- handle rate structure for Betatown

Copyrighted Material. Do Not Distribute.

8.1.2 Iteration 2: First Hour

Based on the refactored production code I can now turn my attention to the new rate requirement. The first thing is that I want to avoid mixing up the Alphatown and Betatown test cases. The test cases are my software reliability lifeline: if the set of test cases becomes unmanageable and unreadable I am in trouble. It is therefore wise to separate the test cases for the two cities into distinct classes and thus source code files.

I keep my old Alphatown test cases and introduce a new test case source file, TestProgressiveRate, specifically for the test cases that test the progressive rate policy calculations.

Now, for the test cases to be defined, I remember to *take small steps.* The Betatown rate structure actually consists of a series of distinct computations. I therefore rewrite my test list to allow taking small steps.

- ~~refactor Alphatown to use a compositional design~~
- First hour = $ 1.50
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

Let us go for the simple one first: *First hour = $ 1.50*. I start by *Step 1: Quickly add a test* introducing my first test case in the new JUnit test case class. Also I have to configure the pay station with a ProgressiveRateStrategy.

Fragment: chapter/refactor/iteration-2/src/test/java/paystation/domain/TestProgressiveRate.java

```
@BeforeEach
public void setUp() {
  ps = new PayStationImpl(new ProgressiveRateStrategy());
}

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment(25); ps.addPayment(25);
  ps.addPayment(25); ps.addPayment(25);

  ps.addPayment(25); ps.addPayment(25);

  assertThat(ps.readDisplay(), is(60) /*minutes*/);
}
```

This code, however, does not compile as there is no implementation of ProgressiveRateStrategy. I could introduce a minimal test stub and thereby get a successful compilation but a failed test, but I realize that the first hour rate calculation is actually *Obvious Implementation* because it is identical to that of Alphatown's rate policy. The first iteration of the rate policy calculation is therefore simply:

Listing: chapter/refactor/iteration-2/src/main/java/paystation/domain/ProgressiveRateStrategy.java

```
package paystation.domain;
/** A progressive calculation rate strategy.
```

Copyrighted Material. Do Not Distribute.

```
*/
public class ProgressiveRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount * 2 / 5;
  }
}
```

As a result all test cases now pass which means I am already at *Step 4: Run all tests and see them all succeed*.

Actually, I always get suspicious if a test case passes the first time, and you should be concerned too. Often the test case passes for the wrong reason. A recurring reason is forgetting to add the @Test annotation to the test method! Or introducing a new test class but forgetting to make JUnit execute it. One suggestion is to temporarily add a `assertThat(false, is(true));` at the end of the test case method and execute the suite! If the suite does not fail now you know that it was due to the test case not being executed.

> **Key Point: Ensure that your tests actually execute**
>
> *Test cases that are not executed does not improve our software's reliability. A simple trick to get that safe feeling in your stomach is to temporarily make the test case fail on purpose. If you do not see the test suite fail, then you have somehow forgotten to add it properly to the suite of executing tests.*

Adding the ProgressiveRateStrategy leads to the compositional design in Figure 8.1 that I have been aiming at.

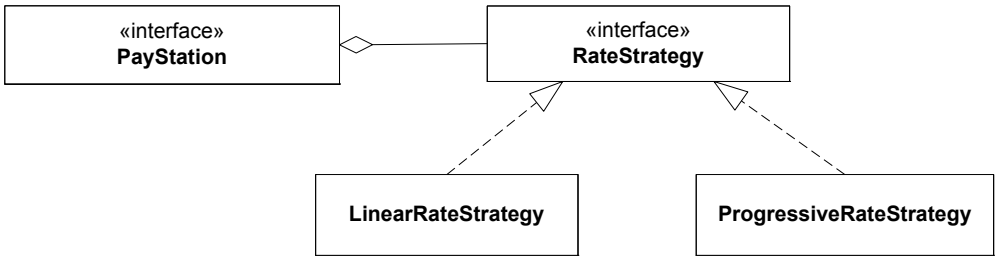


Figure 8.1: Compositional proposal for handling variants of rate calculation.

Step 5: Refactor to remove duplication. Is there any code duplication? Actually there is. The first hour rate is the same for the two towns and thus the calculation code is duplicated in both concrete implementations of the RateStrategy interface. Should I do something about that? Here, as always, I must weigh the cost of the duplication with the cost of removing it. And removing it would introduce more complexity than it would remove. A final thought is that the overlap in algorithms in the two towns is likely a coincidence which also weighs against trying to come up with a way to remove the duplication.

☞ Come up with some proposals to avoid duplicating the code.

Copyrighted Material. Do Not Distribute.

However, another duplication concern is in the testing code. I have quite a few `ps.addPayment(25);` statements. Looking over the next test cases I realize that they will contain even more of these and lead to a large amount of duplicated code. So I decide to reduce the clutter that the long lists of payment statements will generate. I must keep the *Evident Tests* principle so making a loop is not the obvious path[1]; but a private method is a readable and simple approach. So I refactor the test case implementation:

Fragment: chapter/refactor/iteration-2.5/src/test/java/paystation/domain/TestProgressiveRate.java

```
/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  addOneDollar();
  addHalfDollar();

  assertThat(ps.readDisplay(), is(60) /*minutes*/);
}

private void addHalfDollar() throws IllegalCoinException  {
  ps.addPayment( 25 ); ps.addPayment( 25 );
}
private void addOneDollar() throws IllegalCoinException {
  addHalfDollar(); addHalfDollar();
}
```

JUnit tells that all tests pass, and I am finished with this iteration.

- ~~refactor Alphatown to use a compositional design~~
- ~~First hour = $ 1.50~~
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

8.1.3 Iteration 3: Second Hour

Note how Betatown has a rate structure that demands a somewhat complex algorithm. This is where the *Triangulation* principle shows it strength! I simply add new test cases until I am certain that all aspects of the algorithm is complete. At the moment I have a single test case that covers only the algorithm's ability to correctly compute the first hour's rate. Thus the natural next triangulation step is to add a test case for the second hour calculation, update the production code algorithm to correctly handle that case, and so on. Thus *Step 1: Quickly add a test*, the next requirement *Second hour: $2.00* I express as a test case (while remembering *Evident Data*):

Fragment: chapter/refactor/iteration-3/src/test/java/paystation/domain/TestProgressiveRate.java

```
  /** Test two hours parking */
@Test public void shouldDisplay120MinFor350cent()
        throws IllegalCoinException {
```

[1] As long as the loop is simple, like here where I should just loop six times, I still consider it an evident test. But avoid more complex loops.

Copyrighted Material. Do Not Distribute.

```
    // Two hours: $1.5+2.0
    addOneDollar();
    addOneDollar();
    addOneDollar();
    addHalfDollar();

    assertThat(ps.readDisplay(), is(2 * 60) /*minutes*/);
  }
```

giving *Step 2: Run all tests and see the new one fail* in the test report: "Expected: is <120> but: was <140>".

Step 3: Make a little change. The purpose of the ProgressiveRateStrategy is to implement the progressive rate algorithm—after a bit of pen-and-pencil work with the numbers I come up with:

Fragment: chapter/refactor/iteration-3/src/main/java/paystation/domain/ProgressiveRateStrategy.java

```
  public int calculateTime( int amount ) {
    int time = 0;
    if ( amount >= 150 ) { // from 1st to 2nd hour
      amount -= 150;
      time = 60 /*min*/ + amount * 3 / 10;
    } else { // up to 1st hour
      time = amount * 2 / 5;
    }
    return time;
  }
```

leading to *Step 4: Run all tests and see them all succeed*. No need for a *Step 5: Refactor to remove duplication*.

- ~~refactor Alphatown to use a compositional design~~
- ~~First hour = $ 1.50~~
- ~~Second hour = $ 1.50 + $ 2.0~~
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

8.1.4 Iteration 4: Third and Following Hours

The rest of the items on the test list are simply further applications of the *Triangulation* principle: drive the implementation of the algorithm by introducing more test cases. I will leave these as an exercise.

Exercise 8.1: Make the TDD iteration(s) to complete the progressive strategy implementation.

Copyrighted Material. Do Not Distribute.

8.1.5 Iteration 5: Unit and Integration Testing

Testing is advocated in Extreme Programming and in the present book as one feasible path to strengthen software's reliability. By factoring out the rate calculation responsibility in a new abstraction I have also made it possible to factor out the testing: I can test the rate policy algorithms independent of the rest of the pay station behavior.

To see how this works, I can refactor the test class for the progressive rate calculation algorithm and in the fixture simply declare a RateStrategy instance directly and let the test cases exercise that instead of the full pay station.

Fragment: chapter/refactor/iteration-5/src/test/java/paystation/domain/TestProgressiveRate.java

```
public class TestProgressiveRate {
  RateStrategy rs;

  @BeforeEach public void setUp() {
    rs = new ProgressiveRateStrategy();
  }
```

The result turns out to be shorter and easier to read. For instance, compare the two hour parking cost for Betatown below with the original on page 33:

Fragment: chapter/refactor/iteration-5/src/test/java/paystation/domain/TestProgressiveRate.java

```
  @Test public void shouldGive120MinFor350cent() {
    // Two hours: $1.5+2.0
    assertThat(rs.calculateTime(350), is(2*60) /* minutes */);
  }
```

I simply avoid all the method calls to insert coins. Once all the tests pass I am confident that the progressive rate algorithm is correct.

One issue remains, though. Do I test the configured Betatown pay station anywhere? No, actually I do not: the TestPayStation class tests all the behavior of the pay station and as it is configured with the LinearRateStrategy it is testing the Alphatown configuration. However, it is only the ProgressiveRateStrategy implementation that is tested in TestProgressiveRate.

In general, you distinguish between different levels of testing. Generally, three levels are identified:

> Definition: **Unit test**
> Unit testing is the process of executing a software unit in isolation in order to find defects in the unit itself.

> Definition: **Integration test**
> Integration testing is the process of executing a software unit in collaboration with other units in order to find defects in their interactions.

> Definition: **System test**
> System testing is the process of executing the whole software system in order to find deviations from the specified requirements.

Copyrighted Material. Do Not Distribute.

Thus the testing levels represent both a difference in scale: going from the individual class or unit to the whole system; as well as difference in viewpoint: going from the developer's viewpoint to the user's viewpoint.

You cannot in general conclude that a system as a whole is tested even though you have thoroughly tested all its parts, nor can you conclude that all parts are thoroughly tested based upon a thorough test of the system as a whole. Therefore testing at all levels is important.

To give an example, I have unit tested the pay station with the LinearRateStrategy and I have unit tested the Betatown rate policy—however, I might configure the Betatown final pay station with the wrong rate strategy. Testing that they are properly integrated is therefore important. In my case, I can make a Betatown integration testing by adding a test case using the Betatown configuration and make sure to test in where its special rate policy appears, for instance in the two hour range:

Fragment: chapter/refactor/iteration-5/src/test/java/paystation/domain/TestPayStation.java

```
@Test
public void shouldIntegrateProgressiveRateCorrectly()
        throws IllegalCoinException {
  // reconfigure ps to be the progressive rate pay station
  ps = new PayStationImpl(new ProgressiveRateStrategy());
  // add $ 2.0: 1.5 gives 1 hour, next 0.5 gives 15 min
  addOneDollar(); addOneDollar();

  // Progressive Rate: 2$ should give 75 min
  assertThat(ps.readDisplay(), is(75));
}
```

To give an example of a unit that may contain defects even though the enclosing system has been thoroughly tested, consider a class that implements methods that are never called by the system (but may later be called when the class is reused in another system), or methods that are called with a limited range of parameter values (say, void foo(int x) is only called with positive values but there is a defect in the algorithm for negative values).

8.1.6 Iteration 6: Unit Testing the Pay Station

I showed how the ProgressiveRateStrategy could be independently unit tested. In this section I will show how you can go one step further and actually unit test the pay station itself independent of the rate strategy chosen. As it is now, much of the test cases for the pay stations behavior, such as buy and cancel, is dependent on the particular Alphatown rate. However, I can actually simplify the testing of the pay station by defining a RateStrategy implementation that is *extremely* simple, namely one that purchase one minute of parking time for every cents—a OneToOneRateStrategy so to speak. Expressing this simple algorithm is easy using a lambda function, as sketched in Sidebar 5.4

Fragment: chapter/refactor/iteration-6/src/test/java/paystation/domain/TestPayStation.java

```
@BeforeEach
public void setUp() {
```

Copyrighted Material. Do Not Distribute.

```
    // Given a PayStation whose rate strategy is that
    // one cent buys one minute parking time
    ps = new PayStationImpl(coinValue -> coinValue);
  }
```

Now, I can use this STRATEGY when I make the test cases for the pay station's behavior that is not related to rate calculations—the simple one-to-one relation between number of entered cents and parking time in minutes makes it less likely that I make errors in the expected output of my tests, for instance review the following testing code:

Fragment: chapter/refactor/iteration-6/src/test/java/paystation/domain/TestPayStation.java

```
  /** Test acceptance of all legal coins */
  @Test
  public void shouldAcceptLegalCoins() throws IllegalCoinException {
    // Given a paystation
    // When I enter 5, 10, and 25 cents
    ps.addPayment(5);
    ps.addPayment(10);
    ps.addPayment(25);
    // Then the display should read 40
    assertThat(ps.readDisplay(), is(5+10+25));
  }
```

This initiative works in favor of software reliability.

☞ Consider what TDD testing principles this refactoring is using.

Now, I have no explicit One2OneRateStrategy implementation here, as I just used a lambda function. If I indeed had a new Java source file for it, it is important that I put it in the test code folder hierarchy: it is only for testing purposes and is not part of the production code.

> **Key Point: Keep all testing related code in the test tree**
>
> *Implementations of delegates that are only used for testing purposes should be stored in the testing tree. This way you avoid cluttering the production code with classes that are not part of the deployed system. It also lets you make deployment files (like jar files) that contain no more code than necessary for the production system.*

The result of this improved analyzability of the pay station testing code is that there is no longer any testing of the LinearRateStrategy. I therefore have to introduce a unit testing of it.

Listing: chapter/refactor/iteration-6/src/test/java/paystation/domain/TestLinearRate.java

```
package paystation.domain;

import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

import static org.hamcrest.MatcherAssert.assertThat;
```

Copyrighted Material. Do Not Distribute.

```
import static org.hamcrest.Matchers.*;

/** Test the linear rate strategy.
*/
public class TestLinearRate {
  /** Test a single hour parking */
  @Test public void shouldDisplay120MinFor300cent() {
    RateStrategy rs = new LinearRateStrategy();
    assertThat(rs.calculateTime(300), is(300 / 5 * 2) );
  }
}
```

Exercise 8.2: Why have I only made a single test for the linear rate policy?

Am I done now? No, I still must do integration testing of the Alphatown pay station. I make yet another test case class TestIntegration and tests the integration between the pay station and the rate strategies.

Listing: chapter/refactor/iteration-6/src/test/java/paystation/domain/TestIntegration.java

```
package paystation.domain;

import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;

/** Integration testing of the configurations of the pay station.
*/
public class TestIntegration {
  private PayStation ps;

  /**
   * Integration testing for the linear rate configuration
   */
  @Test
  public void shouldIntegrateLinearRateCorrectly()
          throws IllegalCoinException {
    // Given a AlphaTown paystation / linear rate
    ps = new PayStationImpl(new LinearRateStrategy());
    // When adding 2$
    addOneDollar(); addOneDollar();

    // Then the display reads 80 minutes
    assertThat(ps.readDisplay(), is(80));
  }

  /**
   * Integration testing for the progressive rate configuration
   */
  @Test
```

Copyrighted Material. Do Not Distribute.

```
  public void shouldIntegrateProgressiveRateCorrectly()
          throws IllegalCoinException {
    // Given a BetaTown pay station / progressive rate
    ps = new PayStationImpl(new ProgressiveRateStrategy());
    // When adding 2$
    addOneDollar(); addOneDollar();

    // Then the display reads 75 minutes
    assertThat(ps.readDisplay(), is(75));
  }

  private void addOneDollar() throws IllegalCoinException {
    ps.addPayment(25); ps.addPayment(25);
    ps.addPayment(25); ps.addPayment(25);
  }
}
```

I also take the opportunity to move the Betatown integration testing into this test class to keep logically related test cases in the same class. No changes of the build script is required—Gradle always runs all JUnit tests located in the `src/test/java` folder hierarchy.

One final note is that I took the opportunity to use the (Given, When, Then) structure for my tests in this iteration.

To sum up my refactoring iteration has included

- Introducing a rate policy One2OneRateStrategy, expressed as a lambda function, that makes the pay station testing code more readable.
- By necessity introduced testing of the LinearRateStrategy in a new test class TestLinearRate.
- Added a test class for the integration testing.
- Added (Given, When, Then) comments to the test cases.

Now, this is a lot of steps—in the concrete work I have *taken smaller steps* than described here and run the test suite after each small step. Still, there has been some investment of effort in this iteration, and one may question if the gain is worth it. I have improved analyzability of the pay station test cases and separated unit tests and integration tests into separate test case classes but the downside has been the many required changes. These all risked introducing defects in the testing code. In a project where the focus was on software development and not on teaching as it is in this book I would probably have postponed this refactoring to see if I could have avoided the cost.

8.1.7 Overview of the Iterations

At this stage, I have been through six iterations. Several of these iterations were not planned initially, but made as opportunities arose in my work. The iterations were:

Copyrighted Material. Do Not Distribute.

1. I refactor to introduce a rate strategy, make all existing test cases pass to ensure the refactoring is reliable.
2. I triangulate the first hour rate calculation into the rate algorithm.
3. I triangulate the second hour rate.
4. I triangulate the third and following hours rate.
5. I discover that the rate strategies can be tested as separate software units, and refactored the test cases for the Betatown's rate algorithm to become a unit test. All this to improve analyzability of the test code.
6. I discover that the analyzability of the pay station test code can be improved by introducing a simple rate strategy. Refactored these test cases. Introduced integration testing of the pay station with the individual rate strategies.

8.1.8 Testing Levels in XP

After these iterations our test suite includes both unit testing, the individual rate strategies and the pay station, as well as integration testing, the whole pay station configured with proper rate strategies. But what about the system testing level? In our case study, the integration and system testing level overlap. This is because the pay station system is so small and only include a few classes. In larger systems, the system testing level does not necessarily overlap with the integration testing level. As an example, a system that consists of four subsystems (think Java packages with 10–20 classes in each) may have integration testing of the interactions between subsystems. These will focus on the interactions themselves. The system level testing, however, will focus on the system returning the proper answers or doing the proper actions to enable users to do their job efficiently. Thus the former test cases stem from an analysis of the interactions while the latter from an analysis of usage situations. In the pay station case, the integration test case found in iteration 5 came from my worry that the Betatown pay station was not configured correctly. Due to the small size of the system, however, you can see that all the actions a user may do to the system are already covered by our set of test cases and no additional system testing cases are needed.

In XP the system testing level is called **functional testing**. Functional tests are still automated tests, just like the unit tests, but they are defined and written in close collaboration with the customer. Unit tests must run at 100% all the time, but this is not so for the functional tests. Rather each small release of the system will generally increase the number of functional tests passing but some will fail due to the features not being free of defects or implementation work simply not begun yet. Being an agile and incremental process, the suite of functional tests also starts out small in the beginning of the project and grows until the project is concluded.

Often functional/system testing is performed using manual testing of the system's graphical user interface. Even with the best unit and integration tests, you will find new defects during this process.

A key point in TDD is how to handle this situation:

Copyrighted Material. Do Not Distribute.

Key Point: Write automated tests before fixing defects

When your manual system testing has demonstrated a defect, start by writing an automated test case that shows the defect, and then correct the production code to fix it.

As an example, I solved the HotStone project (details in a later chapter) but during my system testing of a distributed client-server variant of the game, I found that oddly, one of the attributes of the "hero" did not update correctly even though I had several test cases for the non-distributed variant for this situation that all passed.

So, I wrote a test case which reproduced the defect (the "game" instance below is simulating network communication with a server.)

```
@Test
public void shouldReduceManaOfActingHeroWhenUsePower() {
  // Given initial game in which Findus' hero has mana left = 3
  assertThat(game.getHero(Player.FINDUS).getMana(), is(3));
  // When Findus's hero uses power
  assertThat(game.usePower(Player.FINDUS), is(Status.OK));
  // Then mana is down to 1
  assertThat(game.getHero(Player.FINDUS).getMana(),
          is(3 - GameConstants.HERO_POWER_COST));
}
```

I made several other test cases covering other defects I had uncovered during a couple of hourse play testing—and could then proceed to fix them.

8.2 Summary of Key Concepts

The introduction of the Betatown variant allowed us to explore a number of different techniques and key points. Two key points are that you should *refactor your design first, and only introduce new features second* and you can *use your test cases to verify refactorings*. In this way, you keep focus and take small steps, and the risk of introducing defects are minimized.

The *Triangulation* principle was introduced in a previous chapter but it showed its potential here: I used triangulation to drive the implementation of the Betatown rate algorithm—inventing new test cases that would add more and more parts to the algorithm implementation.

The compositional design allowed me to *test the rate policy in isolation*. I did not need the pay station code. The term *unit testing* denotes testing a software unit in isolation while *integration testing* denotes testing that units collaborate correctly. I could improve analyzability of the pay station unit testing code by configuring it with a simplified rate policy explicitly developed for this purpose. The final level of testing is *system testing* that denotes testing the whole system as seen from the users' and requirements specifications' perspective. In XP this level is called *functional testing* and in contrast to unit tests they do not run at 100% until the project is concluded.

Copyrighted Material. Do Not Distribute.

8.3 Selected Solutions

Discussion of Exercise 8.1:

You will find the resulting production and test code in iteration 5 onwards.

8.4 Review Questions

Argue for the benefits of refactoring the design before introducing new functionality. Argue and exemplify how developed test cases support the refactoring process.

Define the concepts *unit*, *integration* and *system testing*. What is *functional testing* in XP?

Describe the considerations involved when deciding to put an implementing class in either the production or test source code tree. Describe the techniques used to define suites of test cases.

8.5 Further Exercises

Exercise 8.3:

Walk in my footsteps: develop the Betatown variant of the pay station using the compositional approach while maintaining the Alphatown variant.

Exercise 8.4:

In exercise 7.5 you designed a compositional proposal for allowing the pay station to either accept US or Danish coins.

1. Introduce your design by refactoring the existing pay station system.
2. Use TDD to implement the variant that can handle Danish coins and the Danish rate policy.
3. Review your test cases and classify them as either unit or integration tests.
4. Refactor your test cases so unit and integration tests are separated into distinct test classes.

Exercise 8.5:

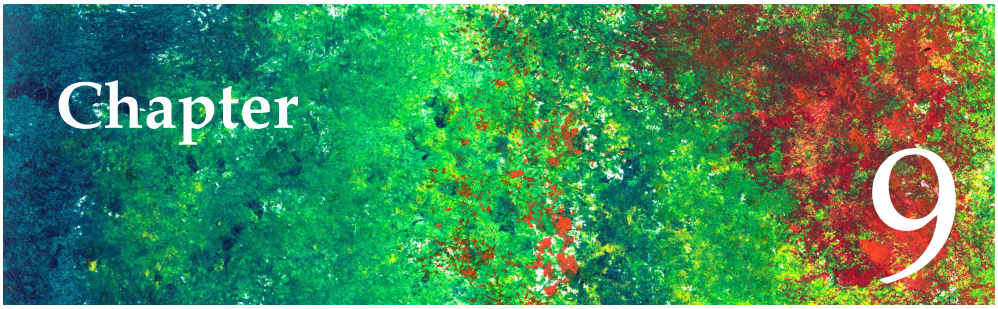
In exercise 5.4 and 6.6 the game of Breakthrough was defined and implemented. Consider that we want to support new rules for how the pawns are moved and captured.

1. Design a STRATEGY based design for handling variants with respect to how pawns are moved and captured.

Copyrighted Material. Do Not Distribute.

2. Refactor your production and test code to support your new design.
3. Invent a new movement rule and implement it using your improved design.

Copyrighted Material. Do Not Distribute.



Design Patterns – Part I

with Morten Lindholm Nielsen

Learning Objectives

So far, I have discussed a design pattern from the viewpoint of solving a problem, analyzing benefits, and studying the programming associated. The learning focus of this chapter is the historical and conceptual aspects of design patterns—the theory of patterns. Two different definitions of patterns will be presented that each represents different aspects of the design pattern concept.

9.1 The History of Design Patterns

The origin of the term "design pattern" can be traced back to the book *Notes on the Synthesis of Form*, written by American architect Christopher Alexander (1964). Alexander examined what he thought to be a new foundation for architectural design, namely that of achieving a "good fit" between form and context. This is exemplified in Figure 9.1: plus or minus symbols on the figure indicate whether the constraints have a positive agreement or are in conflict. To illustrate these concepts, he considered the task of designing a kettle. On one hand, there are constraints regarding the physical shape (the "form") the designer must keep in mind: the kettle must hold water, it must be possible to drain it without scalding oneself, it should be convenient to fill, etc. On the other hand, there are equally important constraints regarding the context of use: should it be designed for gas, electricity, or an open fire? And what about manufacturing? It should be inexpensive to produce and easy to assemble. A design that achieves a "good fit" is a design in which all the constraints are taken into consideration and have been balanced against each other.

Alexander originally strove to create a "science of design" by using mathematics to analyze the various elements, or "forces," that influence the solution to the design

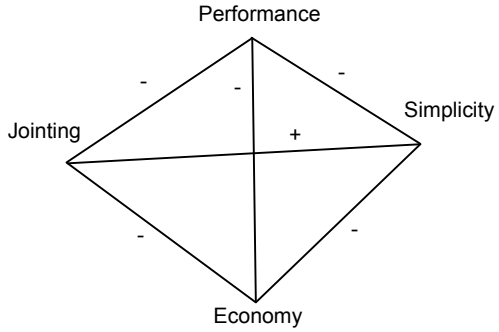


Figure 9.1: Christopher Alexander's explanation of forces.

problem. With this approach, he hoped to overcome the growing complexity of design problems. His analysis of forces later evolved into what is now called "design patterns." During the 1970s, Alexander and colleagues published three books in which they applied the pattern concept to the design of houses and urban plans (see Alexander et al. (1975), (1977), and (1979)). In his books, Alexander orders the patterns by size, beginning with the largest down to the smallest. The largest of the architectural patterns applies to a region, and the smallest applies to some construction detail. Each pattern is connected to the other so that no pattern becomes isolated. The total of all the patterns grows into a "pattern catalogue" to be used as a design tool. This term involves the interrelation between different patterns, where the larger-scale patterns are constructed from smaller-scale patterns. A house is built using small patterns, houses are then clustered following larger patterns, and finally, the relation between towns is laid out according to the largest patterns. Alexander writes his patterns in a specific way which has become the generally accepted format and is therefore known as the Alexandrian Form of pattern writing.

9.2 The Purpose of Patterns

The first question is what problem patterns attempt to solve. The answer is complexity. Alexander noted in 1964 that the modern world has become increasingly more complex and that no modern individual, no matter how capable, can hold all current knowledge. The consequence is the proliferation of specialists, each knowledgeable in a certain area. His observation was that the world, and thus also design activities, have accelerated. Consequently, there is a shorter and shorter time span to complete designs. He then used these two observations to comment on the (to him) apparent lack of quality in modern architecture. To do that, he first had to define what quality is in architecture, or more generally in design. It proved more difficult than he anticipated, because quality in anything is difficult to express verbally. It is not easy to say what characteristic of an object or house that makes it possess quality; it is easier to tell when something does not. It might have a "wrong" shape or lack harmony in some way. Alexander's definition of quality thus became a negating definition, and he named the concept a "quality without a name."

This brings us to his second observation, that of speed. Alexander found that some houses of age had this "quality without a name"—the same quality he also found in

Copyrighted Material. Do Not Distribute.

palm-leaf huts and Polynesian architecture. He found the reason to be rooted in the cultures that built these houses: they were premodern, and their techniques evolved slowly over time. Therefore, when a house was built and there was something about it that did not look "right," the wrong feature was not included in the next house to be built. If a scarf was woven and the pattern did not please, the pattern was not used again. It follows that quality is the result of a historic process in which the odd designs were culled and the pleasing designs were elaborated. Quality is viewed as an emerging concept. It was refined to a degree where even a small alteration disrupted the balance between the various elements affecting the design. The design converged on perfection—a perfect match between form and context.

The central question for Alexander became how to achieve this perfect match, the good fit, when the designer was constrained by lack of knowledge and a shortage of time. Initially, he tried to solve this problem by using mathematics and logic, but later on, his development of the pattern concept to describe interrelating parts of the design proved more fruitful.

The fact that patterns describe interrelating parts of the design leads to a recursive use of patterns (i.e., using a pattern to describe a part of another pattern). Alexander saw this as one of the great advantages of his approach, and he called this use a "pattern language"—a language for speaking about design in terms of patterns consisting of other patterns. This also reflects the structure of the books he wrote, in which large-scale patterns describing the layout and placement of towns are themselves divided into smaller-scale patterns describing neighborhoods and still smaller-scale patterns dealing with the construction of houses and rooms.

Following Alexander's development of the ideas behind patterns, it is helpful to see a pattern as a communicative device used to convey knowledge among designers about problem solving and design. The pattern itself is an open-ended structure that, by itself, does not force a specific design into the debate, but rather hints at relevant possibilities, taking surrounding structures into consideration.

9.3 Patterns as a Communication Device

Another aspect of using patterns in interpersonal communication is the implicit transfer of knowledge that takes place when a person is introduced to patterns. Because the pattern itself encompasses knowledge that former designers have found to be well-founded, the act of learning a pattern presents a special case of learning from the "masters." New ideas are not presented as monolithic solutions but rather in the context of the elements (forces) that may influence how the resulting design may deviate from the actual described solution. Combined with the notion of using pattern languages, this constitutes knowledge transfer, in a way more systematic and encompassing than the presentation of singular, optimal solutions. This communication aspect of pattern fits well with what has become the one-liner description of what a pattern is:

> *A solution to a problem in a context.*

Copyrighted Material. Do Not Distribute.

The idea of using patterns in architecture did not catch on, and, during the 1980s, the idea apparently was disavowed by architects. In another community, however, the patterns concept started to generate interest, namely in the computer science and software engineering camp. In the mid to late 1980s, rumors about "patterns" were heard at conferences on software design.

In 1987, Kent Beck and Ward Cunningham presented the first paper on patterns at a software engineering conference (Beck and Cunningham 1987). Beck and Cunningham describe the results of developing a small pattern language with the goal of guiding novice Smalltalk programmers in creating applications. Over the next few years, interest in patterns was growing, and in 1995, the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* was published (Gamma et al. 1995)[1]. This book brought the term *design pattern* into mainstream use in software development, and it is still considered an essential book on patterns.

Many other books and articles describing additional patterns have been published, but Gamma et al. (1995) remains one of the most influential books on software design of that period. Gamma et al. defined patterns for software as:

> Definition: **Design pattern (Gamma et al.)**
> Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

This definition still stands. The idea was contagious and spread to other fields. In software design, it has been more successful than in any other field we know of.

A defining characteristic of patterns is that one does not invent them; one finds them. A pattern must have proven its worth in real life, and it must have done so more than once. As a rule of thumb, a pattern is not really considered a pattern until it has been verified as a recurring principle, a solution that is general and powerful enough to have been used several times and solved problems in existing systems. A so-called "rule of three" is often used in selecting patterns. A solution must be observed at least three times in different systems before it should be considered a pattern.

9.4 The Pattern Template

As described above, patterns are written according to a specific format or template. In other words, they can be interpreted as a *particular prose form* as emphasized by a definition by Beck et al. (1996):

> Definition: **Design pattern (Beck et al.)**
> A design pattern is a particular prose form of recording design information such that designs which have worked well in the past can be applied again in similar situations in the future.

[1] In the community the four authors became known as the *Gang of Four* (or *GoF*) so you may hear the book referenced by its nickname "the GoF book".

Copyrighted Material. Do Not Distribute.

Authors generally use slightly different templates but they all share common characteristics and have to include:

- **Name.** A pattern has a name that allows us to remember it and communicate with other pattern literates. The name should be short and descriptive of the problem it addresses.
- **Problem.** The problem it addresses must be described, often using examples from real software projects.
- **Solution.** The solution the pattern proposes is described. In a software context this is often done using prose as well as diagrams that show the static and dynamic structure of the software. These diagrams are not to be copied literally but must be viewed as templates for the final solution.
- **Consequences.** Patterns represent solutions that have trade-offs. The benefits and liabilities must be described in order for a developer to make a qualified judgement whether to apply the pattern or not.

In addition to these items, the template used in the design pattern boxes in this book also contains:

- **Intent.** The intent tries to describe the problem as well as the solution in one or two sentences. The intent sections used in this book are generally verbatim copies of those defined in the original book by Gamma et al.
- **Roles.** The pattern structure is a template for object structure and collaboration. This section names each participant or role in the pattern solution.

9.5 Summary of Key Concepts

Design patterns are deeply inspired by the work of Christopher Alexander on building architecture. Alexander wanted to convey high quality solutions to architectural challenges to designers that were constrained by lack of knowledge and a shortage of time. His work led him to propose a pattern language: a set of interrelated patterns ranging from large-scale patterns describing layout of towns right down to small-scale patterns dealing with the construction of rooms. A main benefit of these patterns were as communication devices where designers could learn from the masters.

In late 1980, the software engineering community took on the idea that ultimately led to the first, seminal, design pattern catalogue *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al. 1995). In it, the authors define a design pattern as *descriptions of communicating objects that solve a design problem in a particular context.* Another definition is given by Beck et al. which focuses on design patterns as a *prose form*. This has led to a common template for describing patterns that at least includes *its name*, the *problem* it addresses, the *solution* it proposes, and finally the *consequences* of applying the pattern.

Copyrighted Material. Do Not Distribute.

9.6 Review Questions

Where does the idea of patterns come from? Who invented the first pattern language? Name some of people who brought the pattern idea into software design.

Describe the key elements in the design pattern definitions by Gamma et al. as well as that of Beck et al. Are design patterns invented or discovered? Why? Describe the elements in the template for describing design patterns, and explain why they are required.

9.7 Further Exercises

Exercise 9.1:

Discuss the similarities and differences between a design pattern and an abstract algorithm, like for instance a sorting algorithm described in pseudo-code.

Exercise 9.2:

Describe some advantages of designing and implementing software design with a knowledge of patterns compared to the situation where you do not have this knowledge.

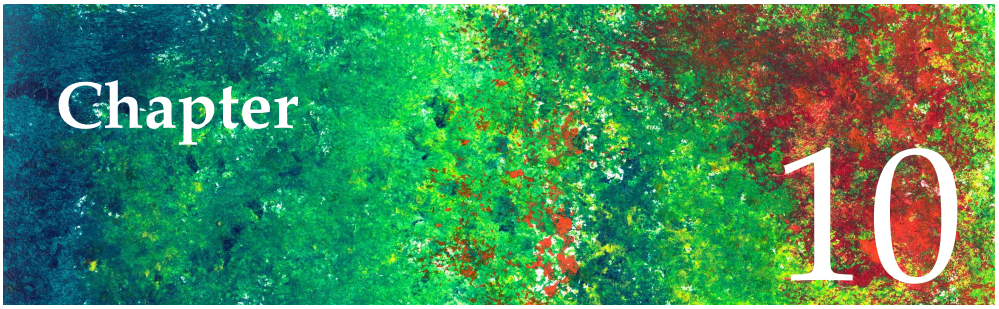
Exercise 9.3:

Discuss the two different definitions of design patterns introduced in this chapter. Explain why they are not in conflict with each other.

Exercise 9.4:

Review the STRATEGY design pattern, defined in pattern box 7.1 on page 26, in the light of the various definitions of design patterns. What aspects of the pattern box does each definition emphasize?

Copyrighted Material. Do Not Distribute.



Coupling and Cohesion

Learning Objectives

Underlying almost all design patterns is a wish for maintainability and flexibility. The objective of this chapter is to introduce two qualities that are relatively easy to judge or measure in our code: *coupling* and *cohesion*. These two qualities are interesting because if I achieve high cohesion and low coupling then maintainability and reliability increase. I will also present a concrete rule, *law of Demeter*, that aims at getting low coupling.

10.1 Maintainable Code

Chapter 3, *Flexibility and Maintainability*, introduced the ISO definition of maintainability, its subqualities, and argued that maintainability measures cost of software change. Furthermore, *flexibility* was defined as a special case of changeability, namely the capability to add/enhance functionality purely by adding software units. The coding practice *change by addition, not by modification* is of course a means to achieve flexible software.

The question is what does maintainable code look like? The problem with the definition of maintainability is that it is not operational: it does not tell me how to write maintainable software. It merely tells me a label, *not very maintainable*, to put on my source code when I later discover it is highly expensive to modify to accommodate my customer's new idea. Of course, I want some more operational concepts, concepts that I can readily apply *while* I am programming in order to keep the code maintainable. And fortunately there are. In the next sections, I will cover some central qualities that are closely linked to source code and therefore relatively easy to measure or judge, and that have a huge impact on how maintainable the system will become.

10.2 Coupling

Definition: **Coupling**

Coupling is a measure of how strongly dependent one software unit is on other software units.

As usual, a *software unit* may be of any granularity: classes, subsystems, packages, even individual methods in a class.

Dependencies between software units come in many forms. In an object-oriented language several types of dependencies come to our mind: methods in a class are coupled by their dependency on the class' instance variables, and classes are coupled as they have relations with each other. Such relations may at the code level be in the form of object creation, method calls to another class, receiving an object as parameter in a method call, etc. Dependencies also exist between packages, when a class in one package calls a method in a class in another, etc. Common to these is that they are directly visible in our source code and you can make tools to find them and thus measure the coupling.

However, dependencies can be much more subtle and less directly visible in the code. This is the reason, you have to review the unit and subjectively judge the degree of coupling, as tools can only find some of the couplings between software units.

Exercise 10.1: List dependencies between software units that exist but are not described by method calls, object references, or other constructs that are easily spotted in the code.

Consider two packages whose dependencies are programmed in two different ways as outline in Figure 10.1. In the left side (a) the *system.gui* package contains two classes that depends on two, respectively three, classes in the *system.domain* package. Thus a change in the dark gray class in the lower package will most likely induce changes to be made in three other classes, those marked light gray in the figure. For instance, a method signature may have to be changed in the dark gray class which means that all places in the light gray classes where the method is called has to be reviewed and analyzed to make the classes compile again. Even worse, if a developer on the domain code team responsible for the *system.domain* package makes changes to what a method does in the dark gray class but does not change any method signature, he may remember to tell his colleague working on the light gray class within the same package, but perhaps he does not tell the GUI team down the hallway. Having strong test cases may catch such changes in the contract, but still effort has to be invested in making the light gray classes work correctly again. The packages have *tight* or *high* coupling; and high coupling lowers maintainability.

High coupling also impacts reliability in a negative way, as any defect or even just changed behavior introduced in the dark gray class may create undesirable effects and maybe defects in all classes depending on it. Thus even a small change in the class may lower reliability in many dependent classes. Also a class that depends on a lot of other classes becomes harder to read and understand again increasing the likelihood of introducing defects. Finally, a class that has many dependencies is harder to reuse in another context.

Copyrighted Material. Do Not Distribute.

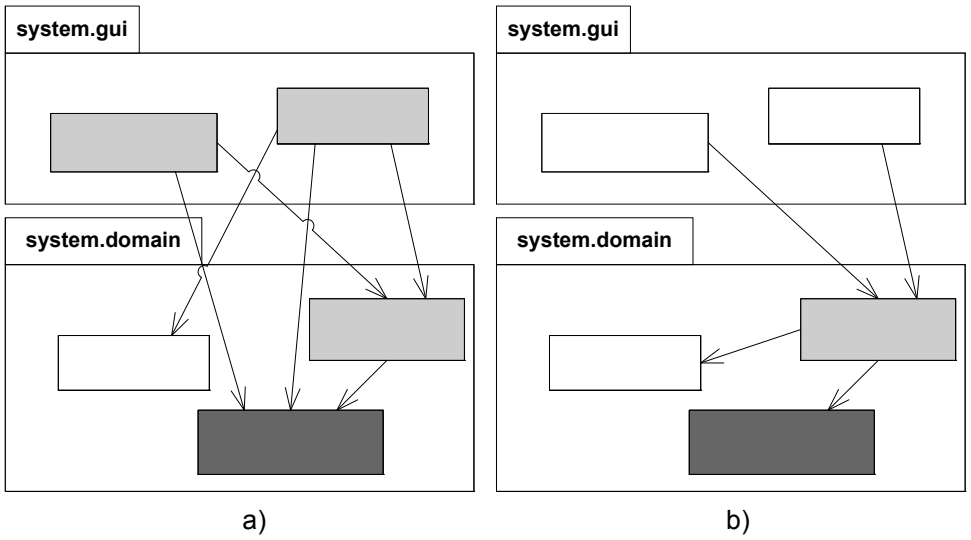


Figure 10.1: Tight (a) and low (b) coupling.

The implementation shown on the right side (b) of the figure shows the same two packages and the same number of classes, but attention has been paid to lower the coupling between classes. Thus only one class directly depends on the change marked class. The two classes in the upper package only depend on a single class in the lower package, in essence this class encapsulates the package (which is in fact the FACADE pattern, discussed in Chapter 19.) The packages have *weak* or *low* coupling.

Weak coupling generally works in favor of maintainable and reliable software. Maintainable because the change in the change marked class only has implications in a single dependent class—thus the effort to review, recode, debug and test is lower compared to the tight coupled version. And the side-effects and defects introduced in the dark gray class ideally only have reliability consequences for the single dependent class. It is only the interplay between these two classes that we must ensure the reliability of. Thus weak coupling also works towards more reliable software. And, a class with few dependencies is easier to reuse and make to work in a new software context.

10.3 Cohesion

Definition: **Cohesion**

Cohesion is a measure of how strongly related and focused the responsibilities and provided behaviors of a software unit are.

Cohesion is a term that basically means "being organized". Consider that all clean clothes of a family with a couple of children are stored in one big pile. It would take ages to find a pair of trousers and a pair of matching socks. Instead, if each member of the family has his or her chest of drawers, and each drawer contains one or two

Copyrighted Material. Do Not Distribute.

types of clothes, finding what to put on today is easy. High cohesion means placing the same type of clothes together in the same drawer. In software, it means software units should have just a few closely related responsibilities. As an example consider this remove method in a collection that one of my students wrote:

```
public void remove(Item i) {
  if ( list.isEmpty() ) {
    System.out.println(``The list is empty, cannot remove'');
  }
  [ remove behavior implemented here ]
}
```

When I read a class that must act as a collection I expect to see methods like add, and remove, and these names convey a lot of understanding of the behavior they provide. In the remove method above, however, the behavior is that of removing an item *and* occasionally printing something! These two behaviors are absolutely not related: removing items and printing. The name of the method does not adequately convey what it actually does. Cohesion is low.

Cohesion is a quality to strive for at all granularities. As an example of cohesion at the package level, consider the following two Java packages, organized by two different principles:

- Package *ABCClasses*: Contains all classes whose names begin with either the letters A, B, or C in my flight reservation system.
- Package *SeatBooking*: All classes related to booking a seat on a plane in my flight reservation system.

Clearly, package *ABCClasses* has low cohesion: it may contain all types of functionality and there is no clear relation between the responsibilities it serves. In contrast, package *SeatBooking* has high cohesion as seat booking embodies a small set of clearly defined responsibilities and the package provides behavior for just that and nothing else.

> **Exercise 10.2:** Find an example of a high cohesion and a low cohesion package in the Java libraries.

High cohesion also contributes towards reliability and maintainability. Cohesive software units greatly increases the analyzability of software, the capability to find defects or parts to be modified, just as the organized approach for storing clothes allows us to find socks quickly. If a software unit has too many responsibilities it will get changed too often. If a class only implements a single feature it will change much less frequently, thus increasing stability.

Of course, you must keep your balance. Cohesive software does not mean software where every class only has one method, and every package only one class. In the pay station case, I started out with a cohesive abstraction, represented by the PayStation interface, but it had five–six responsibilities. However, they all were highly related to the "concept of being a pay station". When need arose, it was refactored to introduce the rate strategy which is of course also very focused.

Cohesion and responsibility are two sides of the same coin per definition. Therefore the discussion on roles, responsibilities, and behavior in Chapter 15 is intimately related to the concept cohesion.

Copyrighted Material. Do Not Distribute.

10.4 Law of Demeter

A very concrete rule that addresses coupling is the law of Demeter first formulated by Lieberherr and Holland (1989). This rule states

> Definition: **Law of Demeter**
> Do not collaborate with indirect objects.

To illustrate what "indirect" and "direct" objects means, let me start with an example. Consider that the Dean of the Faculty of Science is worried about low intake of students at the Department of Computer Science. The department is organized with a head of department that is assisted by several committees which in turn are also headed by a person. One of these committees is the Teaching Committee which is responsible for the webpages that provide information for future students interested in computer science. The actual editing of the webpages is performed by a secretary. As the webpages are confusing and badly written at the moment, one plausible action to take is to improve them. So the question is what action should the dean take? One highly infeasible action is to ask the department head to tell him who is the head of the teaching committee, and next ask him or her about who is the secretary responsible for updating the public relations web pages, and finally tell the secretary detailed instructions about what to change. In pseudo code this action would look something like this:

```
getHeadOfDepartment(CS).getHeadOfCommitte(Teaching).getSecretary().
  changeSection(webpage, section, newText);
```

This is absurd in any organization. The CEO or manager does not have the time to tell each individual employee what to do in detail several times a day. The feasible action of the dean is to tell the department head to make the necessary actions to increase student intake, and leave for him to decide what to do down the chain of command.

```
getHeadOfDepartment(CS).increaseStudentIntake();
```

In the terminology of law of Demeter, the department head is a *direct object* while the secretary is an *indirect object*, and the law states that the dean should talk to his department head, not to the secretary nor to the head of the teaching committee. The rule is also known as *Don't Talk to Strangers* (Larman 2005) which perhaps more directly expresses what the rule is about.

In a software context, there is another reason that talking to strangers is a bad idea. In the first pseudo code line, the dean object actually has to know no less than three interfaces: the interface of the department head, the teaching committee head, and the secretary. Consider that some developer changes the signature of method changeSection in the secretary interface: change the parameter list, change the method name, or similar. This would then require a rewrite of the code line in the dean class and regression testing after the change. The same argument applies for changes in the interface for committee heads. Thus the classes become highly coupled. In the second pseudo code line there are only a coupling between two classes, a change in the secretary interfaces does not ripple up into the dean class. Coupling is much lower.

The law can be restated in more operational terms by defining which objects are not strangers.

Copyrighted Material. Do Not Distribute.

In a method, you should only invoke methods on

- this
- a parameter of the method
- an attribute of this
- an element in a collection which is an attribute of this
- an object created within the method

While formulated as a "law", it is better considered a good design rule that may be broken if good reasons exist. And one good reason that has become common is **fluent interfaces**, also known as *fluent APIs* (**?**). Fluent interfaces relies on method chaining to provide a compact and expressive way of defining complex behaviors. The Java stream api, introduced in Java 8, is an excellent example.

Consider a simple class, holding a Danish numerical grade given to a student at an exam (using public instance variables to keep the code short):

```
class StudentExam {
  public int grade;
  public String studentId;
}
```

and next consider a List<StudentExam> studentList collection of a set of given grades. Then I can compute the average score using the streaming api like this:

```
System.out.println("Average grade is: " +
                   studentList
                     .stream()
                     .mapToInt(exam -> exam.grade)
                     .average()
                     .orElse(0.0));
```

This is a short and expressive way of iterating through all elements, select the 'grade' field, and compute the average (or return 0.0 in case a null value is returned).

☞ Rewrite the above average computation into "classic Java" using a for loop and compare the size of the code.

This fluent interface clearly violates the law of Demeter—there are four levels of indirect objects in this single line, and each method call returns an object with its own interface that you have to know.

The tradeoff here is to ensure the interfaces are stable and mature in each indirect object, as is indeed the case with the Java stream API. If the interfaces are mature and never change, the benefits of expressiveness is obvious.

Copyrighted Material. Do Not Distribute.

10.5 Summary of Key Concepts

Coupling and cohesion are two qualities that are relatively easy to judge in our code and as they are highly related to maintainability, they are important to consider in practical development. *Coupling* is a measure of the degree of dependency of one software unit to other units. If a unit has many dependencies it is termed *high or tight coupling*. If a unit has high coupling then it is less maintainable as there are many sources that can force a change to the unit. *Cohesion* is a measure of how closely focused the behaviors of a unit is—how well defined its responsibility is. If a unit has a narrow and well defined responsibility it has *high cohesion* and is more maintainable. The reason is that it is easier to locate the relevant unit to modify in case of a defect or requirement update. The *law of Demeter* is a rule of thumb that, if followed, lowers coupling in deep dependency graphs. It states that software code should not collaborate with indirect objects. One notable exception is *fluent interfaces* which uses method chaining between indirect objects to achieve compactness and expressiveness.

10.6 Selected Solutions

Discussion of Exercise 10.1:

The indirect dependencies between units are often the most devilish when it comes to software defects. They are difficult to track down and you are often puzzled when they first appear. And the compiler has no chance of helping you out.

Typical indirect dependencies include: Sharing a common global variable, relying on entries in operating system registries or environment variables, relying on a common file format that units read or write, relying on a specific order of initialization, relying on a specific database schema, relying on specific port numbers or IP addresses, etc. All these create coupling between units in a system but are much less visible couplings than direct object references.

The list is very long and in a development team it can be quite difficult to ensure that everybody understands all couplings between units. The war story in sidebar **??** is a good example of a coupling that was hidden from one team developer. It is also an example of an unfortunate coupling that should have been avoided by the other developers.

Discussion of Exercise 10.2:

Classic examples from the Java libraries are java.util that has low cohesion as all sorts of different classes with little or nothing in common are grouped into this package: it contains collection classes, calendar classes, etc. The javax.swing package is highly cohesive as only Swing related GUI components are packaged here.

10.7 Review Questions

Define what coupling is and how to achieve weak and tight coupling.

Define what cohesion is and how to achieve low and high cohesion.

Copyrighted Material. Do Not Distribute.

Describe the relationship between coupling and cohesion and software reliability and maintainability.

Describe the law of Demeter and its implications. What is its relation to the coupling and cohesion properties?

Describe cases where the law of Demeter should not be adhered to.

10.8 Further Exercises

Exercise 10.3:

Discuss the design patterns you have learnt with respect to coupling and cohesion. Do they increase or decrease coupling? Cohesion? Argue why.

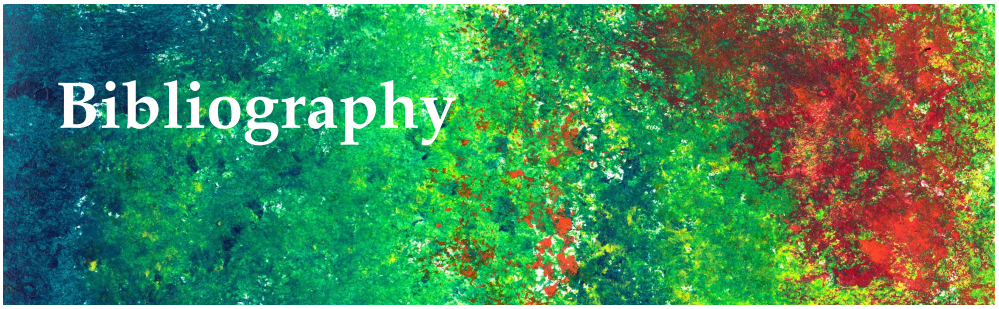
Exercise 10.4:

On page 37 is given a key point: *Keep All Testing Related Code in the Test Tree.* Reformulate this key point using the concepts of coupling and/or cohesion.

Exercise 10.5. Source code directory:
`chapter/refactor/iteration-7`

The final iteration of the pay station system exhibits a complex folder, package, and class structure. Analyze the organization with respect to coupling and cohesion. Consider at least: the contents of the source code and test code tree and the organization and grouping of test cases into separate test classes.

Copyrighted Material. Do Not Distribute.



Bibliography

Alexander, C. (1964). *Notes on the Synthesis of Form*. Cambridge, Massachusetts: Harvard University Press.

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.

Alexander, C., S. Ishikawa, and M. Silverstein (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Alexander, C., M. Silverstein, S. Angel, S. Ishikawa, and D. Abrams (1975). *The Oregon Experiment*. Oxford University Press.

Beck, K., J. O. Coplien, R. Crocker, L. Dominick, G. Meszaros, F. Paulisch, and J. Vlissides (1996). Industrial Experience with Design Patterns. In *Proceedings of 18th International Conference on Software Engineering (ICSE-18)*.

Beck, K. and W. Cunningham (1987). Using Patterns Languages for Object-Oriented Programs. In *Proceedings of the 1987 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-87)*.

Brown, W. H., R. C. Malveau, H. W. McCormick, and T. J. Mowbray (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons.

Ernst, M. D., G. J. Badros, and D. Notkin (2002). An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering 28*(2), 1146–1170.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

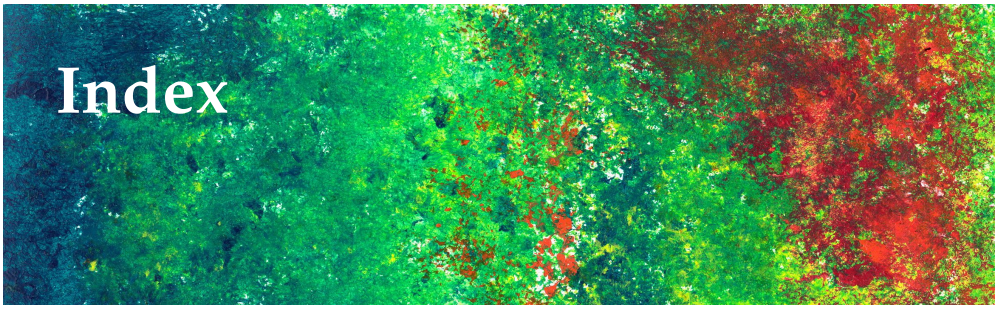
GCC (2009). GCC, the GNU Compiler Collection. http://gcc.gnu.org/, Accessed March 2009.

Larman, C. (2005). *Applying UML and Patterns*. Prentice Hall.

Lieberherr, K. and I. Holland (1989, September). Assuring Good Style for Object-Oriented Programs. *IEEE Software*, 38–48.

Meyer, B. (1988). *Object-Oriented Sofware Construction*. Prentice Hall.

Shalloway, A. and J. R. Trott (2004). *Design Patterns Explained: A New Perspective on Object-Oriented Design, 2nd ed.* Addison-Wesley.



Index

③-①-② process, 22
3-1-2, *see* ③-①-② process

blob, **13**, 21

change by addition, **15**
change by modification, **11**
code bloat, 12
cohesion, **53**
compositional variability, **17–22**, 27
conditional compilation, 13
coupling, **52**
cut'n'paste reuse, 7

delegate object, 19
delegation, **19**
Demeter, law of, 55
design pattern
    Beck et al., **48**
    Gamma et al., **48**
    history, 45–48
dirty job, *see* delegation

Extreme Programming, 40

fluent interface, 56
functional test (XP), 40

god class, *see* blob

history (design patterns), 45–48

integration test, **35**

law of Demeter, **55**
localized variation point, 21

maintainability, 51
multiple maintenance problem, 8

open/closed principle, 15

parametric variability, **9–13**
polymorphic variability, **13–17**
preprocessor directive, 13

refactoring, 28–30

strangers, don't talk to, 55
STRATEGY, **26**
STRATEGY, 22–23
system test, **35**

template (design pattern), 48
Triangulation, example, 33

unit test, **35**

variability
    compositional, **17–22**
    parametric, **9–13**
    polymorphic, **13–17**
    source code copy, **7–9**
variability point, **7**



Index of Sidebars/Key Points

Conditional Compilation in GCC, 13

Ensure that your tests actually execute, 32

Keep all testing related code in the test tree, 37

Refactor the design before introducing new features, 30

Source Tree Copying in SAWOS, 9

Test cases support refactoring, 30

Write automated tests before fixing defects, 41