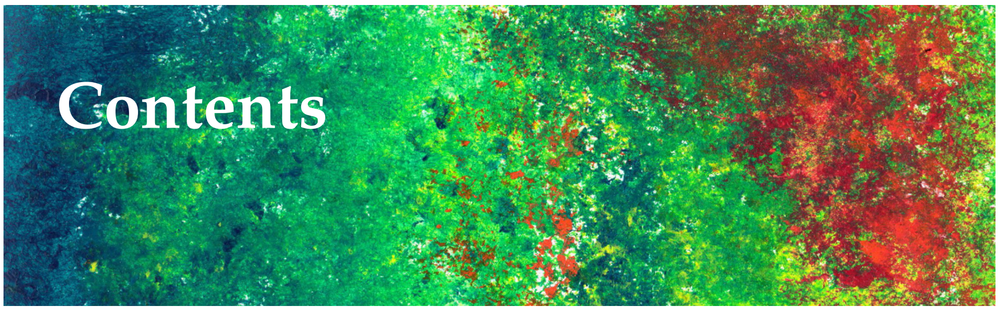


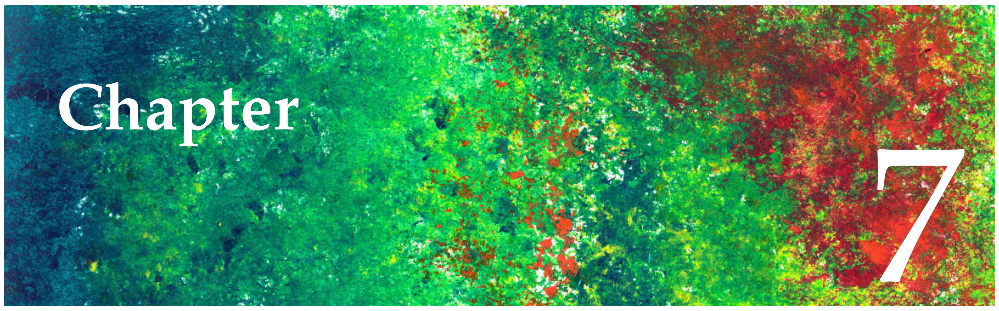
Contents

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.



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:

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.

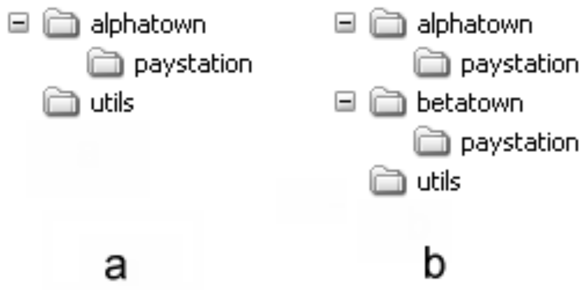


Figure 7.1: Source tree copy proposal.

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

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:

```
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.

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.

```
    [...]
    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**

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, 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.

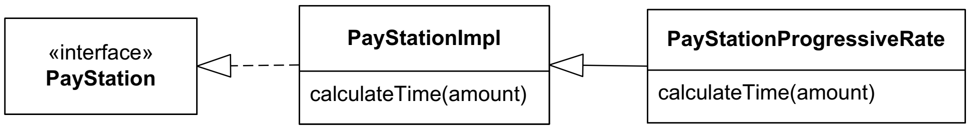


Figure 7.2: Subclassing proposal for new rate calculation.

```
 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.

- *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 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 PayStationProgressiveRateDatabaseLoggingHourMinuteDisplayOptionMobilePhone-Payment. 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.

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.

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);
```

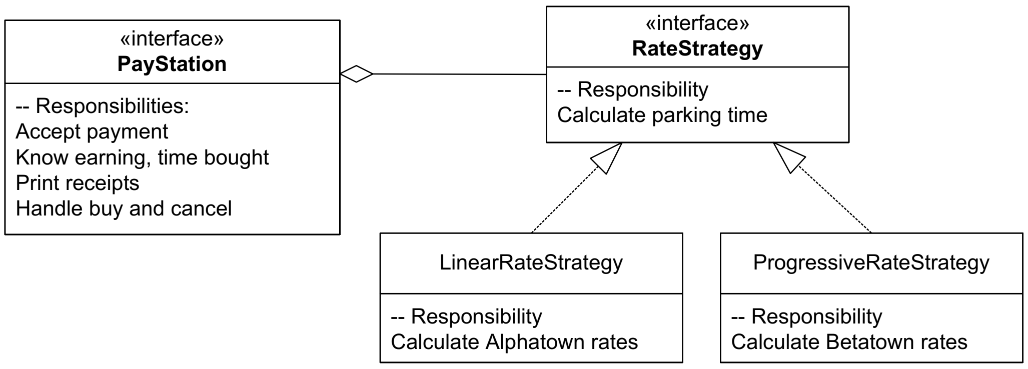


Figure 7.3: Dividing the responsibilities between two abstractions.

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

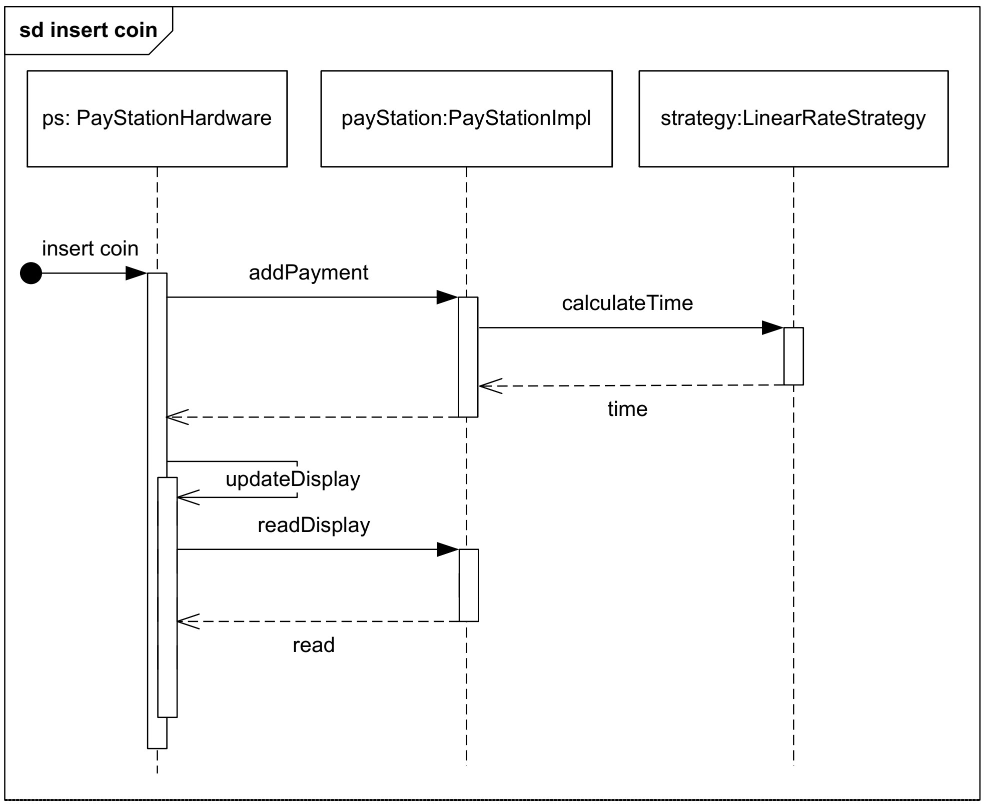


Figure 7.4: Delegating rate calculation to a delegate.

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.

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.

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.

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 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).

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?