

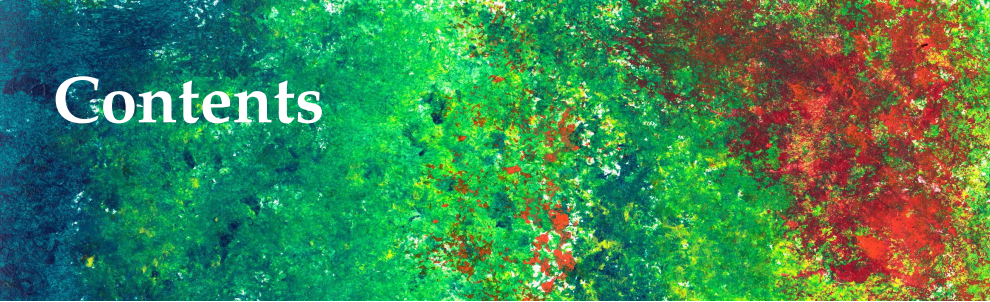
Flexible, Reliable Software:
Using Patterns and Agile Development
PreRelease - 2nd Edition
Part IV
© 2022 – Henrik Bærbak Christensen

Henrik Bærbak Christensen
Version: 09-2022
Status: Release Candidate.

September 6, 2022

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

Reprinted and revised by permission 2020



- IV Variability Management and ③-①-②** **1**
- 11 Deriving State Pattern** **5**
 - 11.1 New Requirements 5
 - 11.2 One Problem – Many Designs 6
 - 11.3 TDD of Alternating Rates 6
 - 11.4 Polymorphic Proposal 8
 - 11.5 Compositional + Parametric Proposal 13
 - 11.6 Compositional Proposal 14
 - 11.7 Development by TDD 16
 - 11.8 Analysis 17
 - 11.9 The State Pattern 18
 - 11.10 State Machines 19
 - 11.11 Summary of Key Concepts 20
 - 11.12 Selected Solutions 20
 - 11.13 Review Questions 21
 - 11.14 Further Exercises 21
- 12 Test Doubles** **25**
 - 12.1 New Requirement 25
 - 12.2 Direct and Indirect Input 26
 - 12.3 One Problem – Many Designs 27
 - 12.4 Test Stub: A Compositional Proposal 28
 - 12.5 Developing the Compositional Proposal 29
 - 12.6 Test Doubles 32

12.7	Analysis	37
12.8	Summary of Key Concepts	38
12.9	Selected Solutions	39
12.10	Review Questions	40
12.11	Further Exercises	40
13	Deriving Abstract Factory	43
13.1	Prelude	43
13.2	New Requirements	44
13.3	One Problem – Many Designs	45
13.4	A Compositional Proposal	45
13.5	The Compositional Process	53
13.6	Abstract Factory	54
13.7	Summary of Key Concepts	56
13.8	Selected Solutions	56
13.9	Review Questions	58
13.10	Further Exercises	58
14	Pattern Fragility	61
14.1	Patterns are Implemented by Code	61
14.2	Declaration of Delegates	62
14.3	Binding in the Right Place	63
14.4	Concealed Parameterization	64
14.5	Avoid Responsibility Erosion	65
14.6	Discussion	66
14.7	Summary of Key Concepts	66
14.8	Review Questions	67
	Bibliography	69
	Index	71
	Index of Sidebars/Key Points	73

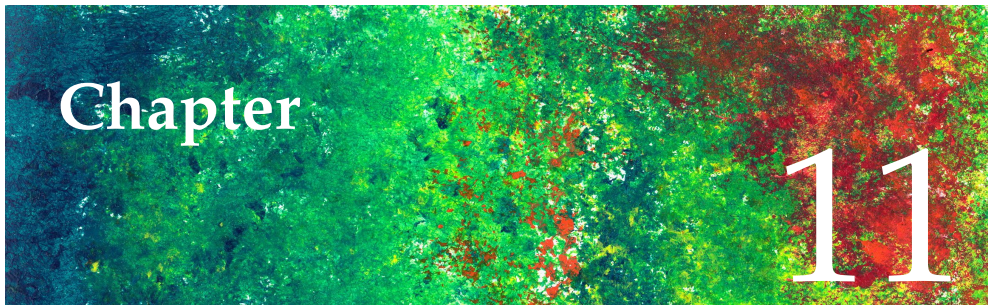
Iteration IV

Variability Management and

③-①-②

The emphasis in the *Variability Management and* ③-①-② learning iteration is on how the ③-①-② process applied to some concrete requirements to our pay station leads to flexible software, and in the process derives some new design patterns and testing techniques.

Chapter	Learning Objective
Chapter 11	<i>Deriving State Pattern.</i> A new requirement requires us to reuse the two rate policies already developed but in a combination. The ③-①-② process provides a solution to this variability problem and the result is the STATE pattern.
Chapter 12	<i>Test Doubles.</i> This time the requirement is from ourselves: we need to increase the testability of the pay station. Here you will apply ③-①-② to develop a test stub, a special case of a <i>test double</i> , and along the way learn the terminology and implications.
Chapter 13	<i>Deriving Abstract Factory.</i> Yet another requirement is handled by ③-①-② and the result is ABSTRACT FACTORY.
Chapter 14	<i>Pattern Fragility.</i> Design patterns are implemented using ordinary programming techniques. However, if you are not careful in your programming effort, many of the benefits of a pattern are lost. Here you will learn some of the pitfalls.



Deriving State Pattern

Learning Objectives

Structuring software systems using a compositional approach has many benefits and here I will demonstrate how it elegantly supports reuse. The learning objective of this chapter is in particular to analyze how the polymorphic and compositional proposals cope when faced with a requirement that combines existing solutions. A major outcome is to demonstrate how the compositional proposal leads to the STATE pattern. Finally, you will see the problems of doing test-driven development when the production code uses resources that are not under direct testing control.

11.1 New Requirements

I have been contacted by a new municipality, Gammatown. They want “almost the same” pay station but with a small twist. They would like to have different rate structures during weekdays and during weekends as they would like people to stay for only shorter intervals during the weekends to increase the number of people that visit the town’s shops during Saturday opening hours.

They require:

- *Weekdays*: The linear rate structure that is used in Alphatown.
- *Weekends*: The progressive rate structure that is used in Betatown.

Obviously, this requirement provides opportunities for a lot of software reuse as I have already designed and programmed both the rate calculation algorithms in question.

11.2 One Problem – Many Designs

Basically I can address this new requirement with the same means as I have done before. I now have three different products to support and they each differs only in a very small fraction of the production code. The question here is the technique to handle this variation.

My options for the production code are:

- *Source tree copy proposal*: Make a copy of the existing source code tree, name it Gammatown and make the alternate coding there. In this case the rate calculation algorithm will contain copies of code from both the Alphetown and Betatown rate algorithms.
- *Parametric proposal*: I add a new Gammatown clause to all relevant switches in the pay station production code.
- *Polymorphic proposal*: I make a subclass of PayStation that can perform the particulars of Gammatown's rate calculation.
- *Compositional proposal with a few if's*: I use my compositional proposal, but introduce a switch in the pay station that selects the proper rate strategy depending on the day of the week.
- *Compositional proposal*: I provide the resulting behavior by composing it, letting object collaboration instead of letting a single one do all the work.

The analysis with regards to the source copy tree proposal as well as the parametric proposal has been sufficiently dealt with in the STRATEGY pattern chapter. The techniques and thus the analyses are the same and I will not repeat the arguments here.

Exercise 11.1: Sketch or program the parametric proposal. Analyze benefits and liabilities of the parametric proposal, and compare with the analysis in the STRATEGY pattern chapter.

The polymorphic and compositional proposals, however, pose special problems and the analysis is thus worthwhile. But before that I must discuss the tests that may drive the process.

11.3 TDD of Alternating Rates

The TDD rhythm states that I must write the tests first. For the Gammatown requirement the only added behavior is the weekend-or-not selection of rate calculation. The rate calculation algorithms themselves have already passed their tests.

It turns out that this is a tricky requirement to test because the pay station's behavior depends upon something that is not under the direct control of our test cases! Remember, a test case is defined as a set of input for a unit under test as well as the expected output. Using the table format, I can write an Alphetown test case like this:

Copyrighted Material. Do Not Distribute.

Unit under test: Rate calculation	
Input	Expected output
pay = 500 cent	200 min.

The translation into a JUnit test case is easy as you have already seen. When the unit under test is the pay station in Gammatown then there is an additional input parameter, namely the day of week, and a test case must of course include this additional parameter.

Unit under test: Rate calculation	
Input	Expected output
pay = 500 cent, day = Monday	200 min.
pay = 500 cent, day = Sunday	150 min.

The problem is that the day of week is not a parameter to neither the pay station nor the rate calculation algorithm and I can therefore not define a JUnit test that expresses the test cases above.

Day of week is an example of an **indirect input parameter**. This input parameter is defined by the computing environment, by the clock in the operating system. The Java libraries can of course read the operating system clock by the class `java.util.GregorianCalendar`. The production code:

```
private boolean isWeekend() {
    Date d = new Date();
    Calendar c = new GregorianCalendar();
    c.setTime(d);
    int dayOfWeek = c.get(Calendar.DAY_OF_WEEK);
    return ( dayOfWeek == Calendar.SATURDAY
            ||
            dayOfWeek == Calendar.SUNDAY);
}
```

is a method that returns true if the present day is either Saturday or Sunday¹.

So, the question is how can I proceed using TDD? I can see several options but none of them are very good.

- I run the linear rate tests during normal work days but have to come in during the weekend to test that the Gammatown pay station uses the progressive rate. This is not what you would call automated testing. Another manual procedure is to set the date on my computer to an appropriate day before running the test case which is also a manual procedure.
- I write some script or procedure to set the date from within the testing code. This may seem like a feasible path but it is actually poor because the Ant build system depends upon the clock in order to recompile properly. Thus the likely result is that the build system fails. Also setting the clock will interfere with clock synchronization software which is standard in modern operating systems.
- I refactor the production code to accept a `Date` object instead of retrieving one by itself. This is actually not a solution because the pay station must continually request a new date as time passes. Thus it cannot simply receive one when it is instantiated.

¹To add confusion, Java 8 introduced a completely new set of data and time functions, however, I will be using the original ones in this chapter.

This date example is an instance of a more general problem, namely how we can get external resources, like the clock, external hardware and sensors, random-number generators, and others, under direct testing control. I will in Chapter 12 discuss an improved and general way of handling external resources.

For now, however, I will continue the discussion of the production code challenges and simply assume that I do manual testing of the Gammatown pay station.

11.4 Polymorphic Proposal

The question is how can I use polymorphism to reuse the two existing and reliable rate algorithms? Let us for a moment forget the compositional design, based on the STRATEGY pattern that I have introduced in the previous chapters, and let us consider how things would have looked like if I had adopted the polymorphic approach for handling Alphatown and Betatown's rate policies. Conceptually, what I want is a class that inherits behavior from both Alphatown's pay station class as well as Betatown's pay station subclass. Thus a class diagram like that in Figure 11.1 comes to mind.

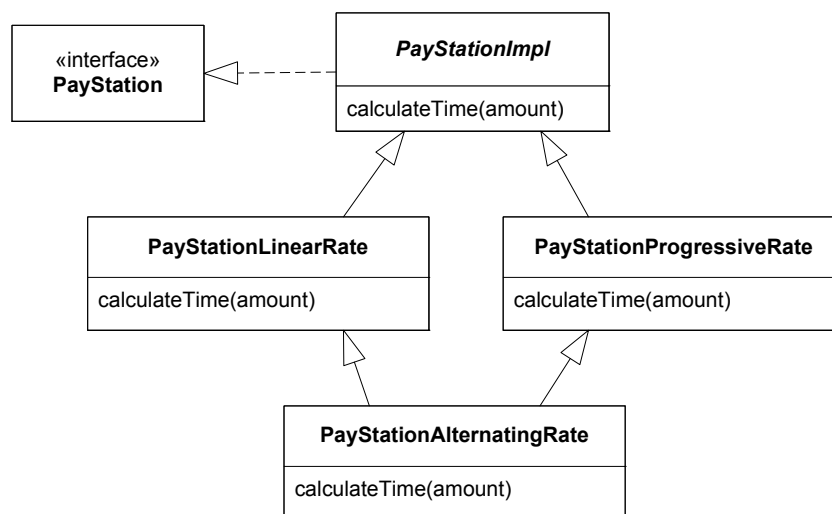


Figure 11.1: Multiple inheritance solution.

In the diagram, I have taken the liberty to refactor the previous polymorphic solution discussed in Chapter 7 a bit: I have introduced an abstract class handling most of the pay station's responsibilities and furthermore defined two subclasses that each fills in the missing rate calculation behavior. Next, I have defined the combined rate strategy as a subclass with both these as superclasses.

You can do this in languages that support *multiple inheritance of implementation* like C++. (If you are not familiar with the difference between inheritance of interface and inheritance of implementation, a short outline is given in sidebar 11.1.) However, modern object-oriented languages like Java and C# have removed the support for having more than one superclass with implementation. The reason is that practice

Sidebar 11.1: Interface and Implementation Inheritance

A class actually defines two things: the interface of objects instantiated from it; and the behavior when methods are called. Thus it combines *specification* as well as *implementation*. An interface, known in Java and C#, on the other hand is only a declaration of the specification, as no implementation is allowed.

When you make a subclass you inherit both the interface as well as the implementation. However, you run into some subtleties when you introduce multiple inheritance—that is if your subclass has two or more superclasses.

The first problem is what `super` means. `super.foo()` calls the superclass' `foo()` method but what then if you have multiple superclasses? C++ solves this as you have to indicate which superclass you mean by writing its name like

```
time = LinearRateStrategy :: calculateTime (amount);
```

The problem is now that you have high coupling between the subclass and the superclass: you are actually not invoking a method in your superclass but have hard-wired a call to a specific class in the class hierarchy. Thus if you change the inheritance hierarchy you introduce some really weird defects.

Other languages have experimented with constructs that allow you to specify in which order superclass methods are called. Experience showed that this gave rise to defects that were even more tricky to find and remove. Yet another attempt is to rename methods in one superclass path but then a developer has to remember two different names for the same method and select the proper one depending on which class it is programmed in.

Java and C# has removed this problem all together by simply removing the possibility of inheriting multiple implementations: you can only have one superclass. However, the possibility of inheriting just the specification, the interface, is a powerful tool and fully supported. Thus you can write

```
public class Foo extends Bar implements X, Y, Z {
    public void calculate () { ... }
```

Note that `Bar`, `X`, `Y`, as well as `Z` may define the `calculate` method but there is never any conflict or misinterpretation of what `calculate` is. In `Foo` calling `super.calculate()` in `calculate` can mean only one thing, namely `Bar`'s `calculate`; and a declaration like

```
private Z myZ = new Foo ();
```

is fine as the `Foo` instance of course provides the `calculate` behavior.

has shown such a construct to open a can of worms of problems: The code is difficult to understand and often leads to odd defects that are difficult to find.

The bottom line is that in Java and C# we cannot do this but have to make a subclass with a single superclass. The question is then how to reuse the implementation we already have in the linear and progressive rate strategy classes. The problem has no “nice” solution actually. I can identify several possibilities but they are all pretty bad (you can find an overview of all four solutions, side by side, in Figure 11.2 on page 11):

1. *Cut'n'paste in a subclass of PayStationImpl.* I make a `PayStationAlternatingRate` subclass of `PayStationImpl` and copy the source code of the two algorithms directly into it.
2. *Cut'n'paste in a subclass of PayStationLinearRate.* Then I only need to cut'n'paste the progressive rate calculation as the other is available by a call to the superclass.
3. *Move the two rate algorithm implementations up as protected methods in the superclass.* I can now define a direct subclass and it simply calls the respective two algorithms as they are accessible in the superclass.
4. *Making a pay station with two pay stations inside.* I make a subclass and inside of it I can instantiate one of each type to handle the calculations.

Below I will treat each solution.

1: Direct subclass. One approach is simple cut'n'paste reuse by defining a subclass `PayStationAlternatingRate` inheriting directly from the abstract `PayStationImpl`. In this I simply paste the code fragments defining the algorithms from the two other classes. This of course works but I now have duplicated code and thus the multiple maintenance problem—not across source tree copies but across classes that contain copies of identical code fragments. In my case these fragments are so small that it is unlikely that I identify defects, but consider the case where the two rate algorithms had been large and complex. Code duplication leads to maintenance problems and better solutions exist.

2: A sub-subclass. If I define the `PayStationAlternatingRate` as a subclass of, say, `PayStationLinearRate` then I can access the linear rate calculation directly in the code by a call to the superclass:

```
public class PayStationAlternatingRate
  extends PayStationLinearRate {
  [...]
  private int calculateTime( int amount ) {
    int time;
    if ( isWeekend() ) {
      [Paste progressive calculation code here]
    } else {
      time = super.calculateTime( amount );
    }
    return time;
  }
}
```

Thus I avoid “half” of the code duplication as any changes or defects removed in the linear strategy algorithm will automatically be reflected. Still, the asymmetry is odd. Why is it this superclass that is chosen and not the other? And I still have the multiple maintenance problem. The bottom line is that I have an odd design that reflects an arbitrary design decision that still has a major problem. Not good...

3: Superclass rate calculations. I can push the rate calculation algorithms up into the superclass `PayStationImpl` in two separate, protected, methods like

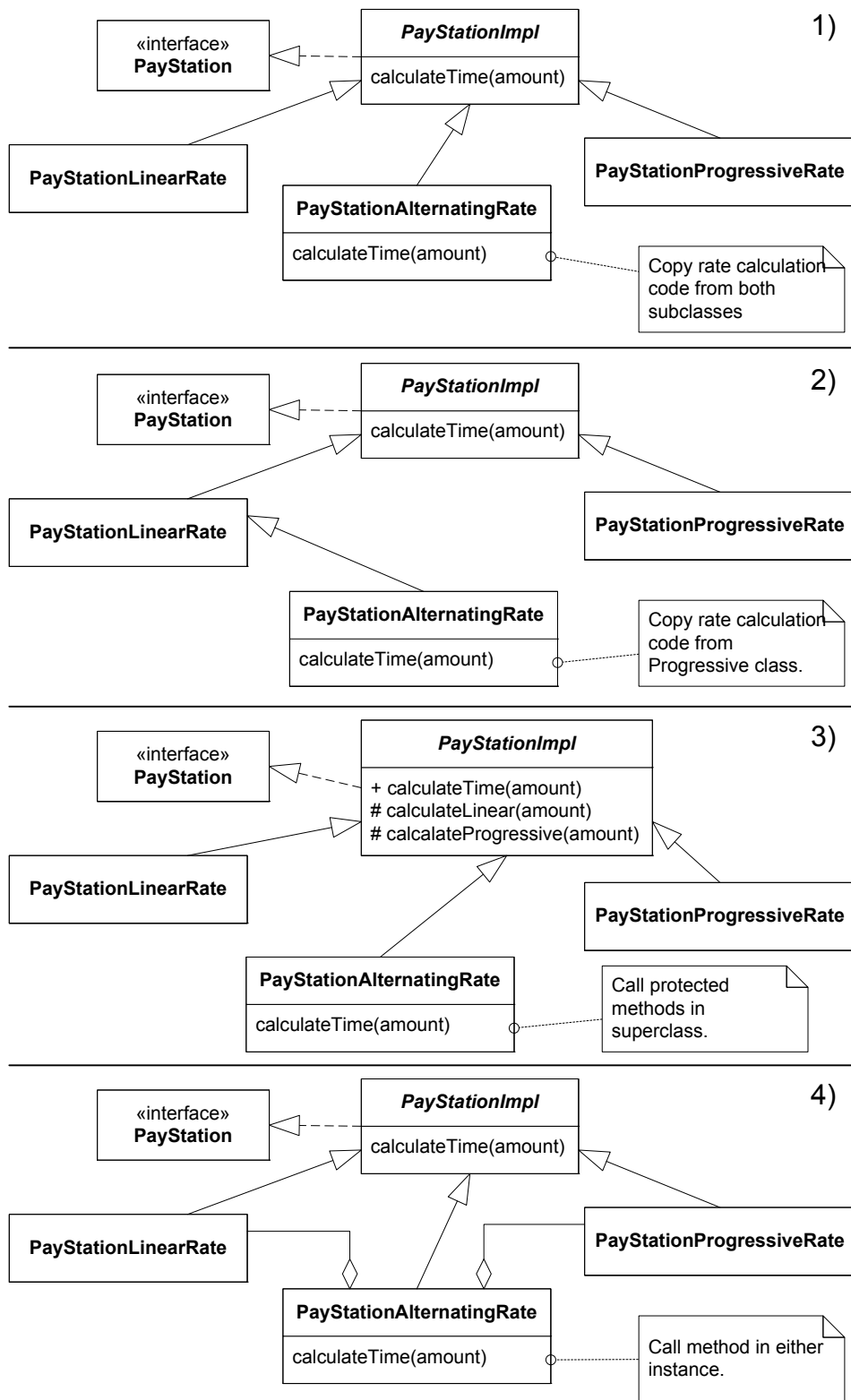


Figure 11.2: Four polymorphic proposals for reusing rate algorithms.

```
public class PayStationImpl implements PayStation {
    [...]
    protected int calculateLinearTime( int amount ) { [...] }
    protected int calculateProgressiveTime( int amount ) { [...] }
}
```

Then the two concrete pay stations for Alphetown and Betatown can call up into the superclass:

```
public class PayStationLinearStrategy
    extends PayStationImpl {
    [...]
    protected int calculateTime( int amount ) {
        return super.calculateLinearTime( amount );
    }
    [...]
}
```

The new Gammatown pay station then becomes:

```
public class PayStationAlternatingRate
    extends PayStationImpl {
    [...]
    protected int calculateTime( int amount ) {
        int time;
        if ( isWeekend() ) {
            time = super.calcProgressiveTime( amount );
        } else {
            time = super.calcLinearTime( amount );
        }
        return time;
    }
}
```

By doing this I avoid code duplication! This solution may seem appealing but it represents a dangerous path in the long run as you will see in the exercise below.

Exercise 11.2: Analyze this proposal's long term maintainability. Consider a scenario where we sell products to a large set of towns with many different rate structure requirements. Also consider whether such changes are supported by *change by addition* or *change by modification*.

4: Pay station with pay stations inside. One other solution that I can come up with is to instantiate `PayStationLinearRate` and `PayStationProgressiveRate` objects within the `PayStationAlternatingRate` object. Then this combination object can delegate rate calculation to these objects. Something like:

```
public class PayStationAlternatingRate
    extends PayStationImpl {
    private PayStation psLinear, psProgressive;
    [...]
    private int calculateTime( int amount ) {
        int time;
        if ( isWeekend() ) {
            time = psProgressive.calculateTime( amount );
        } else {
            time = psLinear.calculateTime( amount );
        }
        return time;
    }
}
```

This solution also has no code duplication. One thing nags me a lot, though. Conceptually, I have a pay station that now contains two other pay stations inside of it. This is conceptually wrong—you do not build pay stations like that. It is the rate calculation algorithms I am interested in reusing, not complete pay stations. Even at the machine level this solution appears strange as memory is used on three pay station objects of which the instance variables of two of them will never be used—not a major technical concern given the vast amount of memory in modern machines but still it points to a problem in this design.

11.5 Compositional + Parametric Proposal

Now, let me return to the compositional design from the strategy pattern chapter. As I have this design and the associated code, it is tempting simply to throw a condition into the pay station to switch on the rate strategy to use. The rationale is that Alphatown's pay stations are simply pay stations that use the same linear rate strategy both in weekends and during weekdays.

While it appears simple at first it quickly becomes a bit awkward to keep all three products intact. I wind up with an implementation like

```
public class PayStationImpl implements PayStation {
    [...]
    /** the strategy for rate calculations */
    private RateStrategy rateStrategyWeekday;
    private RateStrategy rateStrategyWeekend;

    /** Construct a pay station. */
    public PayStationImpl( RateStrategy rateStrategyWeekday,
                          RateStrategy rateStrategyWeekend ) {
        this.rateStrategyWeekday = rateStrategyWeekday;
        this.rateStrategyWeekend = rateStrategyWeekend;
    }
    public void addPayment( int coinValue )
        throws IllegalArgumentException {
        [...]
        if ( isWeekend() ) {
            timeBought = rateStrategyWeekend.calculateTime( insertedSoFar );
        } else {
            timeBought = rateStrategyWeekday.calculateTime( insertedSoFar );
        }
    }
    [...]
    private boolean isWeekend() {
        [...]
    }
}
```

and the testing code for Alphatown then becomes:

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

I have several objections to this proposal. First, of course, I modify existing code which is generally a bad idea. Not only is the pay station implementation code changed but even worse I have to change all the testing code and all other code that instantiates pay stations as the constructor signature has been changed. This is a change with a lot of ripple effect.

Second, the pay station constructor signature is simply weird and inconsistent for Alphatown and Betatown. Why provide *two* rate structures when only one is needed? Even worse, the instantiation of Alphatown pay stations is influenced by a requirement from another town! I have no encapsulation of the pay station and the solution suffers weak cohesion.

Third, I have added yet another responsibility to the pay station, namely *Determine weekend or not* that was not a part of the original design. This is one step towards making the pay station a blob object. It has sneaked in without much notice, and it has nothing to do with the abstraction of a pay station at all. It contributes to low cohesion indeed.

My conclusion: This is a terrible solution.

11.6 Compositional Proposal

In a previous chapter, I introduced the ③-①-② process so let us try to crank the handles of it again and see what comes out this time.

- ③ *I identify some behavior that varies.* The rate calculation behavior is what must vary for Gammatown and this we have already identified.
- ① *I state a responsibility that covers the behavior that varies and encapsulate it by expressing it as an interface.* The `RateStrategy` interface already defines the responsibility to *Calculate parking time* by defining the method `calculateTime`.
- ② *I compose the resulting behavior by delegating the concrete behavior to subordinate objects.* This is the point that takes on a new meaning concerning the Gammatown requirement: its rate calculation behavior can be achieved by combining the two I have already developed.

In my mind, the key to understand compositional design is to think of *collaboration!* If I view my existing design not as objects but metaphorically as a company with employees I see a “specialist worker” that has specialized in calculating linear rates and one that has specialized in progressive rate calculations. The pay station “company” also has an employee that collects money and every time he gets a new coin, he asks the associated specialist worker to calculate the minutes of parking time that he should display. He delegates this task to the rate calculation specialist in the team. The point here is that the pay station employee (alas the `PayStationImpl` object) does not himself choose which rate calculation specialist to use. . . He has simply been put in a team where there is a specialist responsible for doing these calculations—it is the “management” that has defined the team (alas the parameter given in the constructor when the pay station object was created).

Thus, the compositional way to handle this is to realize that we already have the two required specialists in rate calculation in our company. However, we do not want our pay station employee to be overburdened of choosing which one to use. Instead we simply create a team to perform the calculation instead. The team of course consists of the two specialists and then we hire a coordinator to decide which specialist should make the calculation. Officially, he is the guy responsible for calculating rates but instead of doing it himself he simply asks one of his two team mates to make it. The actual behavior required is teamwork instead of individual effort. Also note that the behavior of the coordinator is very simple: he looks at the calendar and based upon whether it is weekday or not he asks the proper specialist.

The pay station employee thus does “business as usual”. He receives a coin and asks the assigned person to calculate parking time. In the Gammatown case, this is the coordinator that delegates the calculation. Once he has received the answer from the specialist, he returns it to the pay station employee. Indeed, many persons are involved but each person’s task is small, well-defined, and easy to understand. The complex behavior arises from the way people collaborate, not because one person performs a complex task.

Objects are no different. Objects provide behavior that match a set of responsibilities and collaborate to get the work done. As long as each object fulfills its responsibility it does not matter how it does it: all by itself, by asking other objects for help, by requesting data from a database, by communicating with a server on the internet, etc. Complex behavior arises from making simple objects collaborate.

Key Point: Object collaborations define compositional designs

When designing software compositionally, you make objects collaborate to achieve complex behavior.

The resulting design is shown in Figure 11.3. I have defined a `AlternatingRateStrategy` class that implements the `RateStrategy` interface, and objects of this class aggregate two rate strategy objects: one for weekend calculations and one for weekdays.

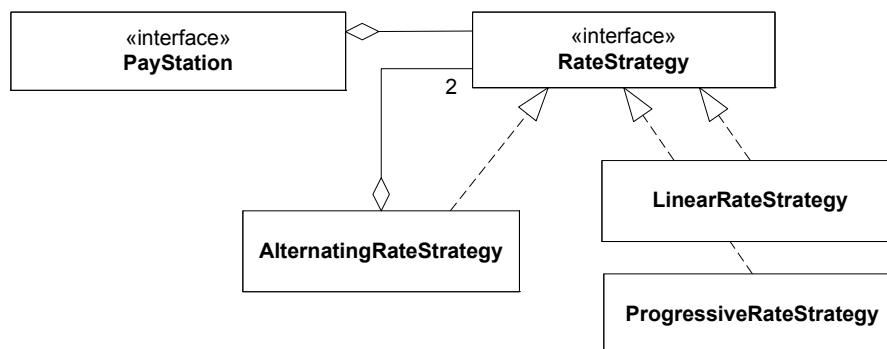


Figure 11.3: Rate calculation as a combined effort.

The required code is very simple as the only addition to the production code is the `AlternatingRateStrategy` class: *Change by addition, not by modification!*

Copyrighted Material. Do Not Distribute.

11.7 Development by TDD

The important problem of getting an external resource (such as the system clock) under automated testing control presents an important design challenge that I will deal with in Chapter 12. Until I get these techniques in place, I will make the testing semi-automatic: add a special gradle target `weekdayTest` for testing the Gamma-town variant during normal working days, and `weekendTest` for weekend testing. I would then have to run the `weekdayTest` tests on a normal working day, and next the `weekendTest` tests during a weekend day.

I will leave the development as an exercise. My own iterations look like this.

- *Iteration 1: Weekday.* In this iteration, I add the `weekdayTest` target to my Gradle build script, a manual `TestGammaWeekdayRate` Java main program that uses the Hamcrest library to test a `AlternatingRateStrategy` and has a single *Representative Data* test case for the linear rate during weekdays. As it fails due to a missing `AlternatingRateStrategy` I create it, add the first linear rate subordinate object and delegate the calculation to it if it is not weekend. *Step 4: Run all tests and see them all succeed* but only because I actually made this iteration on a Wednesday!
- *Iteration 2: Weekend.* Next, I add a `weekendTest` target, I adjust the clock to next Sunday, add a `TestGammaWeekendRate` and finally *Triangulate* the implementation of the rate policy.
- *Iteration 3: Integration.* Integration testing poses some special problems that I will discuss in Chapter 12.

The resulting rate policy implementation becomes

Listing: `chapter/state/compositional/iteration-2/src/main/java/paystation/domain/AlternatingRateStrategy.java`

```
package paystation.domain;

import java.util.*;

/** A rate strategy that uses the State pattern to vary behavior
    according to the state of the system clock: a linear rate
    during weekdays and a progressive rate during weekends.
 */
public class AlternatingRateStrategy implements RateStrategy {
    private RateStrategy
        weekendStrategy, weekdayStrategy, currentState;
    public AlternatingRateStrategy( RateStrategy weekdayStrategy,
                                   RateStrategy weekendStrategy ) {
        this.weekdayStrategy = weekdayStrategy;
        this.weekendStrategy = weekendStrategy;
        this.currentState = null;
    }
    public int calculateTime( int amount ) {
        if ( isWeekend() ) {
            currentState = weekendStrategy;
        } else {
            currentState = weekdayStrategy;
        }
        return currentState.calculateTime( amount );
    }
}
```

```
}  
private boolean isWeekend() {  
    Date d = new Date();  
    Calendar c = new GregorianCalendar();  
    c.setTime(d);  
    int dayOfWeek = c.get(Calendar.DAY_OF_WEEK);  
    return ( dayOfWeek == Calendar.SATURDAY  
            ||  
            dayOfWeek == Calendar.SUNDAY);  
}
```

Note that I have divided the `calculateTime` method into two behavioral parts. First, I decide what state the rate strategy object is in, either the current state is weekend processing or weekday processing. Next, the rate is calculated by delegating to the selected rate strategy object. I could have written this shorter, but as the analysis next shows, this is an example of the STATE pattern, and I have written the code to emphasize this.

The configuration of the pay station then becomes a matter of providing the relevant strategies for weekdays and weekends:

```
RateStrategy rs =  
    new AlternatingRateStrategy(new LinearRateStrategy(),  
                               new ProgressiveRateStrategy());  
PayStation ps = new PayStationImpl(rs);
```

Exercise 11.3: Consider what happens if a driver is entering coins while the time changes over midnight from Friday to Saturday.

11.8 Analysis

What are the properties of the compositional proposal? The benefits are:

- *Reliability.* I have not touched *any* existing production code! The pay station implementation is untouched, as is both rate calculation strategy implementations. I am thus confident that my new Gammatown product will have no effect on the production software for Alphatown and Betatown. Furthermore, I have met a new requirement by *change by addition, not by modification* and thus only have one single new class, `AlternatingRateStrategy`, that I must question the reliability of. Its complexity is low and I can do semi-automatic testing of it so I am confident. But I have admittedly no fully automated testing of it. I must return later to solve this reliability issue.
- *Maintainability.* The implementation is simple and straight forward—except perhaps for the calendar checking code, but this code is required in all solutions. It is much easier to maintain code that you understand.
- *Client's interface consistent.* A very important aspect of this proposal is that the pay station's use of rate calculation objects is unchanged. It has stayed the same even in the face of a new complex requirement. Complex and simple rate calculations are treated uniformly.

- *Reuse.* I have once and for all written the strategy to handle weekday/weekend variations. A new town that is interested in this behavior but with other concrete rate calculations for the respective two periods of time is a simple matter of configuration in the constructor call.
- *Flexible and open for extension.* The present solution is open for new requirements by *Change by addition*. This is in contrast to a parametric solution where the pay station's production code would have to be modified to support new states.
- *State specific behavior is localized.* All the behavior that is depending on the state of the system clock is encapsulated in the `AlternatingRateStrategy`. As the only aspect of the Gammatown requirements that relates to system time is indeed the rate calculation it is the right place. To put it in other words, the cohesion is high.

Some of the liabilities:

- *Increased number of objects.* This is the standard problem with compositional design namely that it introduces more classes and objects. Knowing your patterns, as described in Chapter 18, is a way to reduce this problem.

11.9 The State Pattern

In the discussion above I once again used the ③-①-② process as I did when I discovered the STRATEGY pattern. This time it was in particular the ② part that I elaborated upon: *thinking collaboration*. Instead of the `AlternatingRateStrategy` doing all the calculations itself it *delegated* the concrete rate calculations to delegate objects and itself concentrated on the task of deciding which delegate to use depending on the clock. This perspective on `AlternatingRateStrategy` is focused on how it fulfills its responsibility but let me turn the table and instead look at what it appears to do seen from the pay station's perspective. In essence, the pay station sees an *object that behaves differently depending on its internal state*: is it in its weekend state or is it in its weekday state? This formulation is exactly the intent of the STATE pattern: *Allow an object to alter its behavior when its internal state changes*. The STATE pattern is summarized in design pattern box 11.1 on page 23.

The state pattern defines two central roles: the **context** and the **state**. The context object delegate requests to its current state object, and internal state changes are affected by changing the concrete state object. In my pay station case, the `AlternatingRateStrategy` is the context while `LinearRateStrategy` and `ProgressiveRateStrategy` are state objects. It should be noted that the structure of the STATE pattern (see the pattern box on page 23) is more general than the structure in the pay station: in the pay station, the context object also implements the state interface but this is not always the case. Also the new state is "calculated" each time the `calculateTime` method is invoked which is also not a requirement of the pattern.

An important observation is that the STATE pattern structurally is equivalent to the STRATEGY pattern! Look at the UML class diagrams for the two patterns: they are alike. This is not surprising as it is the same compositional design idea, the ③-①-②

process, that leads to both. So, the interesting question is why do we speak of two different patterns? The answer is that it is two different *problems* that this compositional structure is a solution to. STRATEGY's intent is to handle *variability of business rules or algorithms* whereas STATE's intent is to provide *behavior that varies according to object's internal state*. The rate policy is a variation in business rules used by Alphatown and Betatown, hence the STRATEGY pattern. The problem of Gammatown in contrast is to pick the proper behavior based upon the pay stations internal state, the clock, hence the STATE pattern.

Key Point: Design patterns are defined by the problems they solve

It is the characteristics of the problem a design pattern aims to solve, its intent, that define the proper pattern to apply.

11.10 State Machines

The STATE pattern is a compositional way of implementing **state machines**. State machines are seen in many real life and computational contexts and describe systems that change between different states as a response to external events. A classic and simple case is a subway turnstile that grants access to a subway station only when a traveler inserts a coin. A transition table shows how the turnstile reacts and changes state:

Current State	Event	New State	Action
Locked	Coin entered	Unlocked	Unlock arms
Unlocked	Person passes	Locked	Lock arms
Locked	Person passes	Locked	Sound alarm
Unlocked	Coin entered	Unlocked	Return coin

That is, the turnstile can be in one of two states: "Locked" and "Unlocked" and it changes state depending on the events defined in the table, like entering a coin or passing the arms. The state change itself has an associated action, like sounding the alarm, unlocking the turnstile arms, etc. The STATE pattern can be used to implement the state machine behavior. The turnstile class simply forwards the "coin" and "pass" events to its state object, while the locked and unlocked state objects take appropriate action as well as ensure state changes. An example implementation is shown in Figure 11.4 on page 24.

The state pattern does not dictate which object is responsible for making the state change: the context object or the concrete-state objects. In my pay station case, the only right place to put the state changing code is in the context object, `AlternatingRateStrategy`: putting it in the concrete state objects, `LinearRateStrategy` and `ProgressiveRateStrategy`, would make them incohesive and make them malfunction in a Alphatown/Betatown setting. However, in many state machines it is the individual concrete-state object that knows the proper next state. In this case the concrete-state objects must of course have a reference to the context object in order to tell it to change state.

11.11 Summary of Key Concepts

It is a recurring situation that a requirement for behavior is a combination of behavior already developed. In such a case, reusing the existing code becomes important. In this chapter I have analyzed the polymorphic proposal and it turns out that it is very difficult to achieve a satisfactory design. The polymorphic variants suffer from either code duplication or low cohesion. The code duplication is partly due to the lack of multiple implementation inheritance in modern object-oriented languages like Java and C#; however even the multiple inheritance solution suffers from low maintainability. One of the low cohesion solutions often seen in practice leads to a design where behavior and methods that is actually unique to the subclasses nevertheless bubbles up into an ever growing (abstract) superclass that therefore suffers low cohesion.

The compositional proposal, based upon the ③-①-② process, provides a cleaner way by suggesting to *compose complex behavior from simpler behaviors*. An often useful metaphor is to consider software design as a company or organization consisting of coordinators and specialists that collaborate to get the job done. In the same vein the corner stone of compositional design is *objects that collaborate*.

In the concrete pay station case, the resulting design is an application of the STATE pattern. The STATE pattern describes a solution to the problem of *making an object change behavior according to its internal state*. In the pay station case, the rate calculation changes behavior according to the state of the system clock. The STATE pattern defines two roles, the **context** and the **state** role and require the context object to delegate all state dependent requests to the current state object. The context changes state by changing the object implementing the state role and thus appears to change behavior. Often it is the context itself that changes the state object reference, but it can also be some of the concrete state objects that do it. The STATE pattern is often used to implement *state machines*.

11.12 Selected Solutions

Discussion of Exercise 11.2:

The problem with this solution is that if we want to be consistent in the way we handle new rate requirements (and consistency is the way to keep your code understandable) then the abstract superclass just grows bigger and bigger as it fills up with rate calculation methods. You get *method bloat* in the superclass. This is also a tendency that is often seen in practice. The result is a class with lower cohesion as it contains methods unrelated to itself.

A superclass bloated with methods that are not relevant for itself but only for a large set of subclasses is less understandable. And—new rate requirements will lead to *change by modification* in the superclass.

There is also a risk that it becomes a junk pile of dead code. Consider that “Forty-ThreeTown” no longer uses our pay station product. Maybe I remember to remove the pay station subclass that was running there, but do I also remember to remove the methods in the superclass that are no longer used in any product?

Discussion of Exercise 11.3:

In the current proposal, it is the time of the last entered coin that dictates the rate policy used. So if the driver starts entering coins on Friday but ends on a Saturday it is the weekend rate policy that is used for the full amount.

11.13 Review Questions

Describe the problems associated with testing the Gammatown requirement of rate calculation based upon the day of week.

Outline the benefits and liabilities of a polymorphic proposal to handle Gammatown's requirement.

Outline the benefits and liabilities of a proposal that uses a conditional statement in the pay station to determine which rate strategy object to use.

Outline the fully compositional proposal: What are the steps in the ③-①-② process and what is the resulting design? Outline benefits and liabilities of this proposal.

What is the STATE pattern? What problem does it address and what solution does it suggest? What are the roles involved? What are the benefits and liabilities?

11.14 Further Exercises

Exercise 11.4:

Compare the STRATEGY and STATE pattern and identify similarities and differences between them. You should include aspects like the structure (interfaces, classes, and relations), intent, roles, and cost-benefits.

Exercise 11.5. Source code directory:

`exercise/state/alarm`

A simple digital alarm clock has a display, showing "hour:minute", and three buttons marked "mode", "+" and "-". Normally the display shows the time. By pressing "mode" the display instead shows the alarm time and allows setting the alarm hour by pressing "+" or "-" to increase or decrease the hour. Pressing "mode" a second time allows changing the minutes, and pressing a third time returns the display to showing the time. That is, the alarm clock can be in one of three states: "display time", "set alarm hour", and "set alarm minute" state.

The hardware buttons of the clock invoke methods in the AlarmClock interface:

Listing: `exercise/state/alarm/AlarmClock.java`

```
/** Interface for a simple alarm clock.
 */
public interface AlarmClock {
    /** return the contents of the display depending on the
     * state of the alarm clock.
     */
}
```

```

    * @return the display contents
    */
    public String readDisplay();

    /** press the "mode" button on the clock */
    public void mode();

    /** press the "increase" (+) button on the clock */
    public void increase();

    /** press the "decrease" (-) button on the clock */
    public void decrease();
}

```

To demonstrate how the interface works, consider the following (learning) test where the present time is "11:32" and the alarm set to "06:15".

```

@Test
public void shouldHandleAll() {
    // show time mode
    assertEquals( "11:32", clock.readDisplay() );
    // + and - has no effect in show time mode
    clock.increase();
    assertEquals( "11:32", clock.readDisplay() );
    clock.decrease();
    assertEquals( "11:32", clock.readDisplay() );
    // switch to set hour mode
    clock.mode();
    assertEquals( "06:15", clock.readDisplay() );
    clock.increase(); // increment hour by one
    clock.increase();
    clock.increase();
    assertEquals( "09:15", clock.readDisplay() );
    // switch to set minute mode
    clock.mode();
    clock.decrease();
    clock.decrease();
    clock.decrease();
    clock.decrease();
    clock.decrease();
    assertEquals( "09:10", clock.readDisplay() );

    // go back to show time mode
    clock.mode();
    assertEquals( "11:32", clock.readDisplay() );
    // remembers the set alarm time.
    clock.mode();
    assertEquals( "09:10", clock.readDisplay() );
}

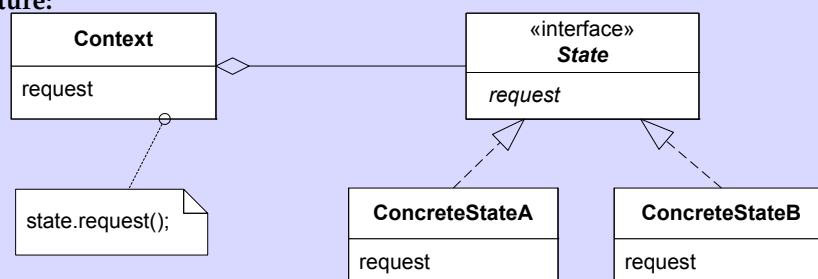
```

1. Sketch a STATE pattern based design for implementing the state machine.
2. Implement the production code. You may "fake" the time (like the clock always being "11:32") but the alarm time must be settable.

[11.1] Design Pattern: State

- Intent** Allow an object to alter its behavior when its internal state changes.
- Problem** Your product's behavior varies at run-time depending upon some internal state.
- Solution** Describe the responsibilities of the dynamically varying behavior in an interface and implement the concrete behavior associated with each unique state in an object, the state object, that implements this interface. The context object delegates to its current state object. When internal state changes occur, the current state object reference is changed to refer to the corresponding state object.

Structure:



- Roles** **State** specifies the responsibilities and interface of the varying behavior associated with a state, and **ConcreteState** objects define the specific behavior associated with each specific state. The **Context** object delegates to its current state object. The state object reference is changed whenever the context changes its internal state.
- Cost - Benefit** *State specific behavior is localized* as all behavior associated with a specific state is in a single class. It *makes state transitions explicit* as assigning the current state object is the only way to change state. A liability is the *increased number of objects and interactions* compared to a state machine based upon conditional statements in the context object.

Listing: chapter/state/turnstile/TurnstileImpl.java

```
/** State pattern implementation of a subway turnstile.
 */
public class TurnstileImpl implements Turnstile {
    State
        lockedState = new LockedState(this),
        unlockedState = new UnlockedState(this),
        state = lockedState;
    public void coin() { state.coin(); }
    public void pass() { state.pass(); }

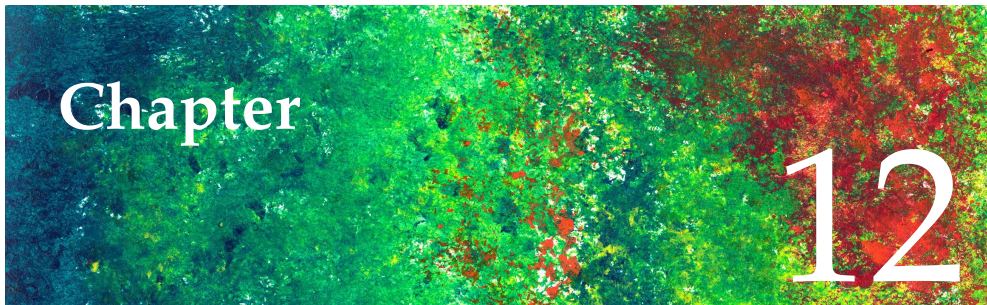
    public static void main(String[] args) {
        System.out.println( "Demo of turnstile state pattern" );
        Turnstile turnstile = new TurnstileImpl();
        turnstile.coin();
        turnstile.pass();
        turnstile.pass();
        turnstile.coin();
        turnstile.coin();
    }
}

abstract class State implements Turnstile {
    protected TurnstileImpl turnstile;
    public State(TurnstileImpl ts) { turnstile = ts; }
}

class LockedState extends State {
    public LockedState(TurnstileImpl ts) { super(ts); }
    public void coin() {
        System.out.println( "Locked state: Coin accepted" );
        turnstile.state = turnstile.unlockedState;
    }
    public void pass() {
        System.out.println( "Locked state: Passenger pass: SOUND ALARM" );
    }
}

class UnlockedState extends State {
    public UnlockedState(TurnstileImpl ts) { super(ts); }
    public void coin() {
        System.out.println( "Unlocked state: Coin entered: RETURN IT" );
    }
    public void pass() {
        System.out.println( "Unlocked state: Passenger pass" );
        turnstile.state = turnstile.lockedState;
    }
}
}
```

Figure 11.4: Example implementation of turnstile state pattern.



Test Doubles

Learning Objectives

A result of the design developed in the last chapter was the identification of a major problem, namely how to get the pay station rate policy under automated testing control when it depends upon the system clock. The focus of this chapter is for you to learn the terminology for *test doubles* and *test stubs* and see how they help us in our quest to automate testing as much as possible. This chapter

- Defines a new requirement demanding that all rate strategies are under fully automatic testing control.
- Analyzes the present production and testing code in order to isolate the problem.
- Identifies the problem as one of controlling the *input* values to a unit under test, and defines terminology, *direct* and *indirect input*, to describe the problem.
- Discusses ways to handle indirect input and classifies them as yet another example of handling variability.
- Develops the compositional proposal by using the ③-①-② process and refactoring the pay station production code.

12.1 New Requirement

The present solution for testing Gammatown's rate policy is not fully automated as it requires me to pick one of two different test targets, one for weekends and one for weekdays. Moreover I have to set and reset the system clock in order to verify the behavior. My new requirement is to make as much as possible of this verification automated by JUnit testing code.

12.2 Direct and Indirect Input

In the last chapter I identified the system clock as an indirect input parameter. So before I discuss a plausible solution I will dwell a bit on the problem from a more abstract point of view as it is something you run into all the time in testing and test driven development.

Abstractly I can describe the relation between production and testing code like in Figure 12.1. In this UML communication diagram, the numbered lines represent method calls between objects, executed in the sequence shown by the numbers. The JUnit test object symbolize the JUnit testing code that first sets up the production code, next exercises it, and finally verifies that its output matches the expected output. The testing code exercises a particular part of the production code, the *unit under test* shown as the UUT object. The right hand object, denoted DOU, symbolizes units of the production code that the UUT depends upon, and is explained in detail below.

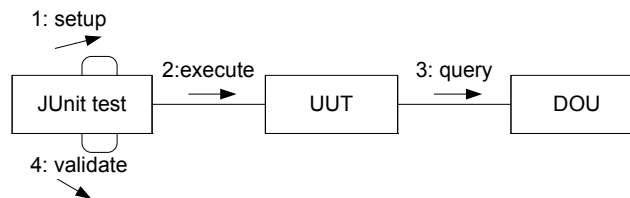


Figure 12.1: The relation between test and production code.

In our concrete context the setup, exercise and verify code was (for weekdays):

Fragment: chapter/state/compositional/iteration-2/src/test/java/paystation/manual/TestGammaWeekdayRate.java

```

System.out.println("Manual test of GammaTown Rate for Weekdays");
RateStrategy rs =
    new AlternatingRateStrategy(new LinearRateStrategy(),
                               new ProgressiveRateStrategy());
// Should show 200 minutes for 500 cents
assertThat(rs.calculateTime(500), is(500/5 * 2));
  
```

Now, a test case must state all the input parameters

Input	Expected output
pay = 500 cent, day = Monday	200 min.
pay = 500 cent, day = Sunday	150 min.

This allows us to classify input into two categories.

Definition: Direct input

Direct input is values or data, provided directly by the testing code, that affect the behavior of the unit under test (UUT).

Definition: Indirect input

Indirect input is values or data, that cannot be provided directly by the testing code, that affect the behavior of the unit under test (UUT).

In my example, the payment of 500 cent is direct input (it is a parameter to the method), while the day of week is indirect (I cannot provide the method with this value.) Note that input is not required to be direct parameters to a method to be classified as direct input, the only qualifying property is the ability of the testing code to *somehow* provide the proper values. For instance an object's instance variables may be direct input if there are methods available that allows the testing code to set their values.

Indirect input stems from units that our UUT depends upon. These are called dependent-on units (DOU):

Definition: Depended-on unit

A unit in the production code that provides values or behavior that affect the behavior of the unit under test.

Control and data flows both from the UUT towards the DOU as it calls methods in the DOU and back when the DOU returns values or invoke methods in the UUT. In our Gammatown example, the UUT is the Gammatown `AlternatingRateStrategy` instance which depends upon a DOU, the Java library behavior to compute whether it is weekend or not.

Armed with this analysis I can restate my problem: *How can I ensure that the DOU returns the values (indirect input) that are specified in my test case during testing — and the real values during normal execution?*

12.3 One Problem – Many Designs

The new problem statement certainly sounds like something that I have already analyzed: How do I vary software behavior? I may do it using several proposals:

- *Parametric proposal:* I define some boolean instance variable in the pay station that defines whether the production code is in testing or normal mode. I can switch on this parameter in the `AlternatingRateStrategy`. Of course I also have to make an instance variable that tells which day it is. This variable is then never used in normal operation.
- *Polymorphic proposal:* I can subclass the `AlternatingRateStrategy` into an `TestingAlternatingRateStrategy` that overrides the `isWeekend` method, and provide the pay station with an instance of this class instead. This class must of course also be told which day to return.
- *Compositional proposal:* I use the ③-①-② process to identify, encapsulate, and delegate to the behavior that is variable.

I of course favor the compositional solution for all the same reasons that you have already seen and will therefore focus on the compositional proposal.

12.4 Test Stub: A Compositional Proposal

The ③-①-② process:

- ③ *I identify some behavior that varies.* It is basically the behavior defined by the `isWeekend()` method that is variable.
- ① *I state a responsibility that covers the behavior that varies by an interface.* I will define an interface `WeekendDecisionStrategy` containing the `isWeekend()` method.
- ② *I compose the desired behavior by delegating.* Again, this is the real principle that brings the solution: I simply let the `AlternatingRateStrategy` call the `isWeekend()` method provided by the `WeekendDecisionStrategy` to find out whether it is weekend or not. I can then make implementations that either returns a preset value (for testing) or uses the operating system clock (for production usage).

This is another application of the STRATEGY pattern.

Exercise 12.1: Provide the arguments why this is a STRATEGY pattern and not a STATE pattern.

Thus the class diagram will look like in Figure 12.2.

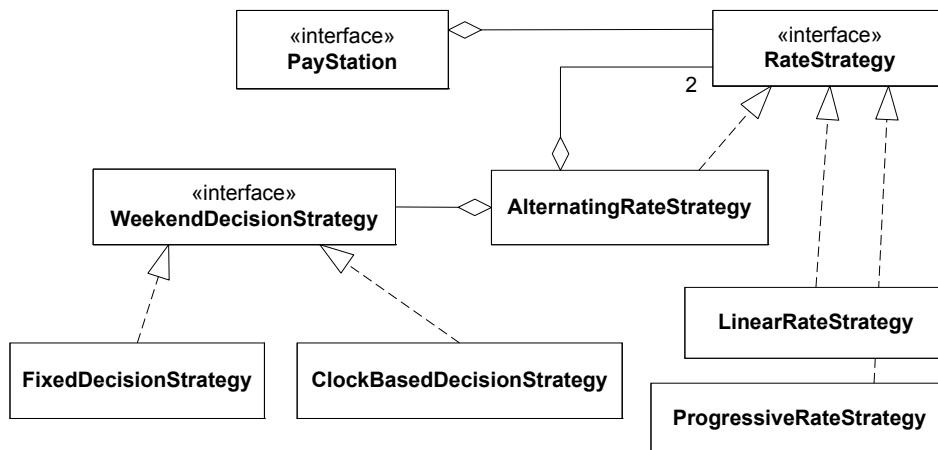


Figure 12.2: Adding a strategy to decide whether it is weekend or not.

Returning to the problem from the testing point of view, I have actually replaced the real depended-on unit with a *test stub*:

Definition: Test stub

A test stub is a replacement of a real *depended-on unit* that feeds indirect input, defined by the test code, into the *unit under test*.

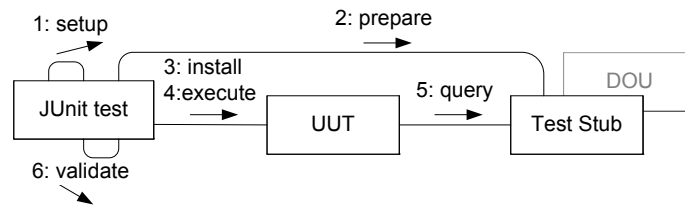


Figure 12.3: Test Stub replacing the DOU.

A test stub has the exact same interface as the DOU but normally only returns values that have been set by the testing code, as shown in Figure 12.3. Here the test code prepares the stub (usually by creating it and setting what it should return), next installs it into the UUT (using dependency injection), and finally execute the tests.

Thus, to test the `AlternatingRateStrategy` for its weekend processing I will provide it with a `WeekendDecisionStrategy` whose `isWeekend()` method is set to return whatever value the testing code has set.

Note that the ability to use test stubs hinges on the UUT's ability to collaborate with the stub instead of the real depended-on unit. That is, it is not possible if the UUT itself creates the DOU or in other ways is tightly coupled to it.

The test stub allows our test code to control indirect input to the UUT, essentially converting it to direct input. Our example was our `AlternatingRateStrategy` making a *query method call* to the system clock. But—what about **indirect output**? What if our UUT makes a *command method call* to a DOU, trying to alter its state? One example is a software unit to control the temperature in a fridge, by turning the cooling system on and off based upon measured room temperature, by invoking, say, `cooling.turnOn()`. A replacement that handles this case is *test spy* and I will return to discuss it in Section 12.6. For now, however, let us look at the process of introducing our test stub.

Exercise 12.2: In Chapter 8, I introduced a special rate strategy, `One2OneRateStrategy`, to isolate testing of the pay station from the testing of the rate calculation. Review this design in the light of the test stub terminology introduced. What software units in the pay station design match the terms above?

12.5 Developing the Compositional Proposal

As when I introduced the `RateStrategy` I first go through a refactoring phase. When all tests again pass on the refactored design I concentrate on introducing the test stub.

12.5.1 Iteration 1: Refactoring

Refactoring is of course supported by the already developed test cases and I follow the same plan as I did in the STRATEGY chapter:

Copyrighted Material. Do Not Distribute.

- Introduce the new interface.
- Refactor the existing `AlternatingRateStrategy` to take instances of this interface as parameter in the constructor. See that it compiles but the tests fail.
- Refactor the existing design to make all test cases pass again. This will require introducing the `ClockBasedDecisionStrategy`.

As the refactoring process is similar to the one I went through in Chapter I will leave out the details. In the end, I have the same functionality as before: some automated tests, and the manual `weekendTest` and `weekdayTest` tests.

12.5.2 Iteration 2: Test Stub

I can now introduce the test stub that I name `FixedDecisionStrategy`. The implementation is *Obvious Implementation*.

Listing: `chapter/test-double/iteration-2/src/test/java/paystation/domain/FixedDecisionStrategy.java`

```
package paystation.domain;

import java.util.*;

/** A test stub for the weekend decision strategy.
 */

public class FixedDecisionStrategy
    implements WeekendDecisionStrategy {
    private boolean isWeekend;
    /** construct a test stub weekend decision strategy.
     * @param isWeekend the boolean value to return in all calls to
     * method isWeekend().
     */
    public FixedDecisionStrategy(boolean isWeekend) {
        this.isWeekend = isWeekend;
    }
    public boolean isWeekend() {
        return isWeekend;
    }
}
```

 Why is this implementation located in the test folder?

Now for the first time, the testing of Gammatown's rate strategy for *both* weekday and weekend pass as the test stub can now provide the UUT with the proper indirect input. Thus a test case for the week day testing

Input	Expected output
pay = 300 cent, day = Wednesday	120 min.

can be rephrased

Input	Expected output
pay = 300 cent, day-type = weekday	120 min.

and directly expressed in my JUnit test case:

Copyrighted Material. Do Not Distribute.

Fragment: chapter/test-double/iteration-2/src/test/java/paystation/domain/TestGammaWeekdayRate.java

```
@Test public void shouldDisplay120MinFor300cent() {
    RateStrategy rs =
        new AlternatingRateStrategy( new LinearRateStrategy(),
                                    new ProgressiveRateStrategy(),
                                    new FixedDecisionStrategy(false));
    assertThat(rs.calculateTime(300), is(300 / 5 * 2));
}
```

The direct input is given as parameter to `calculateTime` while the indirect input by the test stub. Now I can run both weekday and weekend tests at any time. As there is now no need for the separate test targets I can add an item to my test list so I remember to refactor the tests to make the structure of the Gammatown tests similar to that of Alphatown and Betatown.

12.5.3 Iteration 3: Refactoring Tests

This step entails


- Making a `TestAlternatingRate`, moving all Gammatown rate policy test cases here, and deleting the two old test case classes.
- Removing the special test targets from the build script.

These are pretty trivial steps and I will not dwell on the details. However, I note that an important issue remains: *integration testing*.

12.5.4 Iteration 4: Integration Testing

Integration testing is defined as validating the interaction between software units, and here I need to test that the pay station is indeed configured with the proper rate strategy. To verify that the pay station is indeed a Gammatown pay station I will actually have to add two test cases: one in which the pay station is configured with the `AlternatingRateStrategy` which again is configured to use the test stub, `FixedDecisionStrategy`, telling it is a weekday; and one in which the latter tells it is a weekend day.

However, when I look into the `TestIntegration` class I see the existing two integration test cases that I developed earlier for Alphatown and Betatown, and the question is what increased reliability I get for the cost of making the additional test cases.

 Take a moment to review the `TestIntegration` code and evaluate if there are any additional interactions that a pay station configured with the Gammatown rate strategy will test over the interactions already tested by the two other test cases.

My conclusion is that I get very little return on my investment in coding time. The two existing integration tests verify that the pay station indeed interact correctly with the rate strategy object that is passed as parameter in its constructor. Then why should it not interact properly with the rate strategy of Gammatown? Therefore I conclude that additional integration testing of this interaction is a waste of time.

Exercise 12.3: Actually, this argumentation can be tied to one of the testing principles that was presented in Chapter 5. Evaluate the various TDD principles, find the one that matches the argumentation, and argue why it matches.

12.6 Test Doubles

Our test stub controlled indirect input to the UUT. But often I need to verify the indirect output as well; and actually there are other test related situations in which I need to use various replacements for the DOU.

Meszaros (2007) provides detailed classification of DOU replacements based upon their intended use. He defines **test double** as the general concept, and identifies these classes:

- *Test stub*: A double whose purpose it is to feed indirect input, defined by the test case, into the UUT.
- *Test spy*: A double whose purpose it is to record the UUT's indirect output for later verification by the test case.
- *Mock object*: A double, created and programmed dynamically by a mock library, that may both serve as a stub and spy.
- *Fake object*: A double whose purpose is to be a light-weight performant replacement for a slow or out-of-process DOU.

Each type is described in separate sections below, however, I can summarize the key point of test doubles as:

Key Point: Test doubles make software testable

Many software units depend on indirect input and output that influence their behavior. Typical indirect input are external resources like hardware sensors, GPS location sensors, random-number generators, system clocks, etc. Typical indirect output is commanding external hardware to open valves, start engines, or writing output to external devices like file systems, databases, etc.

A test double replaces the real Depended On Unit and allows the testing code to control the indirect input, and record the indirect output for verification.

12.6.1 Spy Example

As a simple example, consider a monitoring system, `PlantMonitor`, to control the temperature of a chemical process in a chemical plant by turning cooling on and off. If the chemical reaction temperature in the process increases beyond 67 degrees Celsius, cooling is turned on; and turned off when the process temperature drops below 62 degrees Celsius.

The `PlantMonitor` interface has only a single method, `regulateTemperature()`, whose responsibility is simple: measure the temperature and turn on/off the cooling. Some

external client will invoke this method every 5 seconds or so, but not the focus of the present discussion; our focus is to ensure correct behaviour of the `regulateTemperature()`.

```
public interface PlantMonitor {
    void regulateTemperature ();
}
```


As is evident from the interface above—there is no direct input or output in this system. Instead, a concrete implementation uses dependency injection to bind the actual hardware of the sensor and cooling, like this

```
TemperatureSensor temperatureSensor;
CoolingSystem coolingSystem;
temperatureSensor = new TemperatureSensorHardware ();
coolingSystem = new CoolingSystemHardware ();
plantMonitor = new StandardPlantMonitor(
    temperatureSensor,
    coolingSystem);
```

The two hardware systems are again defined by interfaces

```
public interface TemperatureSensor {
    double readTemperature ();
}
public interface CoolingSystem {
    void turnCoolingOn ();
    void turnCoolingOff ();
}
```

The temperature input is indirect, just as the commands to turn cooling on/off are indirect output. The temperature input is an good example of using a test stub while the cooling is a good example of using a test spy.

 Argue why the above statements are true.

Now, the unit under test is the `regulateTemperature()` method which is admittedly rather simple but still important to get under automated test control.

Using a stub and a spy I can do that. As shown in the test case below, I can configure the stub to return a specific temperature; while inspecting the output of the spy after the regulation should have been done.

Fragment: `chapter/test-double/spy/src/test/java/chemicalplant/TestTemperatureRegulation.java`

```
@Test
public void shouldTurnOnCoolingAbove67degrees () {
    // Given a temperature above 67
    temperatureSensor.setTemperature (67.2);
    // When the monitor needs to regulate the temperature
    plantMonitor.regulateTemperature ();
    // Then cooling is commanded to turn on cooling
    assertThat (coolingSystem.lastMethodCalled (), is ("turnCoolingOn"));
}
```

While the test case reads naturally, there are some important things to note. First, apparently there are new methods on the `TemperatureSensor` and `CoolingSystem`, methods that do not appear in the interface? Actually no. It is quite common to provide special methods, called **retrieval interfaces**, for inspecting and setting state in doubles, but these are *only defined in the test double classes*. (They could also be handled by a private/role interface, discussed in Chapter 16.) Therefore the fixture of the above test looks like this:

```
Fragment: chapter/test-double/spy/src/test/java/chemicalplant/TestTemperatureRegulation.java
public class TestTemperatureRegulation {
    private PlantMonitor plantMonitor;
    private TemperatureSensorStub temperatureSensor;
    private CoolingSystemSpy coolingSystem;

    @BeforeEach
    public void setup() {
        temperatureSensor = new TemperatureSensorStub();
        coolingSystem = new CoolingSystemSpy();
        plantMonitor = new StandardPlantMonitor(temperatureSensor,
                                                coolingSystem);
    }
}
```

Note how both the stub and the spy *are declared by the concrete classes* to have access to these retrieval methods.

Second, as these retrieval methods do not appear in the hardware interfaces, there are no testing code “creeping” into the production code. The hardware temperature sensor of course cannot implement a `setTemperature()` method, right?

Test spies must record interactions and parameters passed, for later verification by the JUnit test code. Depending upon the complexity of the interaction, that I need to verify, the code to do the “recording” vary: are many methods called in a particular sequence, and do I need to verify a lot of parameters? In our case, the interaction is simple, and the only interesting information to record is “what was the last method called?”, so I use a simple string as recording device:

```
Fragment: chapter/test-double/spy/src/test/java/chemicalplant/CoolingSystemSpy.java
public class CoolingSystemSpy implements CoolingSystem {
    private String lastCalledMethod = "none";
    @Override
    public void turnCoolingOn() {
        lastCalledMethod = "turnCoolingOn";
    }

    @Override
    public void turnCoolingOff() {
        lastCalledMethod = "turnCoolingOff";
    }

    public String lastMethodCalled() {
        return lastCalledMethod;
    }
}
```

Note that the code illustrates a subtle but important point—the spy does not *record the resulting state* but it does *record the interaction (the indirect output)*. The interesting issue is not that the state of the cooling system is set to “on”; the interesting issue is that the *plant monitor told the cooling system to turn the cooling on*.

12.6.2 Mock Object Example

Mock objects are doubles that are created dynamically by a mock library based upon an interface. There are quite a few Java mock libraries, like EasyMock, jMock, and Mockito. By using a mock library, you avoid writing the test double code yourself which (may) increase your speed when developing test code. However, the mocks have to be told what to return and what sequence of method invocations to expect and this programming can turn out to be somewhat cumbersome, and is always highly specific to the particular mock library, you use. So, your test code becomes highly coupled to the library, meaning moving to another will require you to do a lot of test code refactoring.

In the code below, I have used Mockito as mock library, on the same PlantMonitor cooling system as in the previous section (the Gradle build file of course is updated to pull the Mockito library from Maven Central). The setup method does not need to declare the sensor and cooling system by concrete types, and the mock library takes care of creating the objects, as shown below:

Fragment: chapter/test-double/mock/src/test/java/chemicalplant/TestTemperatureRegulation.java

```
public class TestTemperatureRegulation {
    private PlantMonitor plantMonitor;
    private TemperatureSensor temperatureSensor;
    private CoolingSystem coolingSystem;

    @BeforeEach
    public void setup() {
        temperatureSensor = mock(TemperatureSensor.class);
        coolingSystem = mock(CoolingSystem.class);
        plantMonitor = new StandardPlantMonitor(temperatureSensor,
                                                coolingSystem);
    }
}
```

In the test code, you program the mocks for their *expectations* on the indirect input (stub behaviour) and indirect output (spy behavior). In Mockito you do that using `when()` and `verify()` for indirect input and output respectively. Therefore, the same test method as in the previous section looks like this:

Fragment: chapter/test-double/mock/src/test/java/chemicalplant/TestTemperatureRegulation.java

```
@Test
public void shouldTurnOnCoolingAbove67degrees() {
    // Given a temperature sensor which returns 67.2
    when(temperatureSensor.readTemperature()).thenReturn(67.2);
    // When the monitor needs to regulate the temperature
    plantMonitor.regulateTemperature();
    // Then cooling is commanded to turn on cooling
    verify(coolingSystem).turnCoolingOn();
}
```

A classic presentation of mocks is given by Freeman et al. (2004).

12.6.3 Fake Object Example

Fake objects are doubles that replace “expensive” units with “light-weight” but functionally correct alternatives. By “expensive” I means units that are slow and/or cumbersome to execute. The classic example is a database. A database is tedious from a

testing perspective as it usually requires an external database server to be started and ensure properly running which is difficult to control from our test code. In addition it is comparatively slow to query and update, and it is expensive to reset it to an initial state to allow *Isolated Test*. A fake object may be implemented as a simple in-memory hashmap that provides limited but correct *store()* and *query()* behavior.

As an example, I had a project with a simple accounting role, that should store debit transactions for users in a MariaDB SQL database.

```
public interface Accounting {
    /** Add a debit transaction , representing
     * a user using a paid service for some amount.
     * @param debit the user/amount data to debit.
     */
    void addDebitTransaction(Transaction debit);

    /**
     * Get the complete list of transactions for
     * a given user , using the userId (not the pincode!)
     * @param userId
     * @return List of all transactions
     */
    List<Transaction> getTransactionsFor(int userId);

    /** Compute the total amount of cents
     * spent by given user , using the userId (not the pincode!)
     * @param userId
     * @return amount in cents.
     */
    int getTotalInCentFor(int userId);
}
```

This interface was developed as part of my TDD process, and having a MariaDB running and resetting it between each “run all tests” does not adhere to the values of speed and simplicity, so I developed the following *fake object*, which simply uses an internal HashMap of transactions, with the user’s ID as key:

```
public class FakeObjectAccount implements Accounting {
    Map<String , List<Transaction>> db = new HashMap<>();

    @Override
    public void addDebitTransaction(Transaction debit) {
        String userId = ""+debit.getUserId();
        List<Transaction> list = db.get(userId);
        if (list == null) {
            list = new ArrayList<>();
        }
        list.add(debit);
        db.put(userId , list);
    }

    @Override
    public List<Transaction> getTransactionsFor(int userId) {
        List<Transaction> list = db.get(""+userId);
        return list;
    }

    @Override
```

```
public int getTotalInCentFor(int userId) {
    List<Transaction> list = db.get(""+userId);
    int total = list.stream()
        .mapToInt( element -> element.getAmountInCent())
        .sum();
    return total;
}
```

As a side note: I also developed a small set of test cases to verify the behaviour of my fake object implementation. Later when I proceeded to implementing the real MariaDB driver code for each method, these tests came in handy to validate that the SQL expressions were indeed correct.

12.7 Analysis

So, what have I achieved in this exercise? First, I have increased the fraction of the production code under testing control, and the testing is fully automatic. At the start of this iteration, I had three different test targets and some only worked on specific days. Now, all test cases are again run as a single suite. Second, the main point of this exercise, of course, is the *test double* concept. Test doubles allow us to replace “real” software units (the *depended on units*) with surrogates that can be instrumented by our testing code to provide the proper *indirect input values*, or record the *indirect output values*. And third, the process of defining the test stub demonstrates how a strong focus on testing goes hand in hand with compositional design and the ③-①-② process.

It is difficult to introduce test doubles if the behavior that provides the indirect input/output is not well encapsulated. The ③-①-② process is basically all about solving this problem and tells me to do it by abstracting and expressing the behavior by defining an interface and by using delegation.

You may argue that my solution is overly complex for such a minor problem as testing a single `if` statement in a single method. And of course you are right. The key point is that the technique becomes much more valuable once the size and complexity of the problem grows, so take the simple Gammatown pay station problem as a vehicle for demonstrating the technique in practice. Consult sidebar 12.1 for a realistic case of using stubs.

However, this iteration also has its weaknesses. There are no more any specific test cases for the real production code implementation of the `isWeekend` method. Let us consider a situation where I have actually coded a defect into the `ClockBasedDecisionStrategy` that leads to failures. My refactored test cases will not expose this failure and thus the defect may go undetected until our Gammatown customer starts complaining. From this perspective, the refactored solution I have now is automatic, yes, but the quality of the test is slightly lower as the old manual tests *did* test the real clock based behavior. The bottom-line is that you can drive automatic testing to a certain point and no further, and the key is to have as little code as possible left for manual or system testing. *The fewer lines of code, the lesser the risk of having defects is.* In our case, only the simple `isWeekend` method in the `ClockBasedDecisionStrategy` has to be manually tested.

Sidebar 12.1: Wind Computation Testing in SAWOS

Data on wind is essential for aircraft pilots during landing and take-off. Therefore the SAWOS system (see sidebar 2.1 in Chapter 2) calculated a *two minute mean* wind value that averaged the wind direction and wind speed in the last two minute interval. Wind direction calculations were especially tricky as the valid interval is between 0 and 360 degrees where 0 degrees is due north, 90 degrees is due east, etc. However, if the wind is in north then the set of readings vary from 0 to perhaps 358, back to 0, to 3, etc., so a trivial summation and average algorithm is incorrect. Obviously, we could not wait for the weather to be correct in order to test the computations, so we developed a test stub that could be instrumented with a set of wind data readings, and configured the wind computation object with this instead of the real wind sensor.

Later we reused the test stub (and those developed for all the other types of sensors: temperature, humidity, clouds, RVR, ...) for demonstration purposes where the SAWOS system ran on a stand-alone computer without any sensors attached. This allowed us to go through the first acceptance tests by the customer at our premises.

It is also important to note that I rely on the Java library code to perform as specified. I do not make test cases to test `get(Calendar.DAY_OF_WEEK)`, or stub the `GregorianCalendar` class. Testing is about getting as much reliability as possible with the least possible effort, and I do not think the investment in testing Java library implementation will find any defects, and therefore is a waste of time.

12.8 Summary of Key Concepts

The basic idea of unit testing is to execute a specific software unit (the *unit under test*, UUT) with specific input values and verify that the computed output match the expected output. The input can be classified as either *direct* or *indirect* input. Direct input is values that the testing code can directly control, typically by passing values as parameters in method calls, or by setting the receiving object's state beforehand. Indirect input is values that are provided by other software units that the unit under test depends upon (*depended-on units*, DOUs). Often it is difficult or impossible for the testing code to force these depended-on units to provide specific values relevant for the particular test.

In this case, these units should be replaced by *test stubs* i.e. test specific software units that have the same interface but can be programmed by the testing code to provide specific indirect input relevant for the test case. This makes automated testing feasible.

The requirement that a certain type of input, the indirect input, should be able to stem from two different sources is basically a variability requirement, and the ③-①-② process can be applied. This means that the indirect input should be encapsulated by an interface and that the production code should use delegation to request the indirect input.

A typical example of using test stubs is to encapsulate external resources like random-number generators, hardware sensors, etc. In this case, there will often be some code

that cannot be verified by automated tests, namely the code that do the actual hardware handling, random-number generation, etc. The key point is to make this portion of the code as small as possible. Manual and system testing should be employed to ensure the reliability of these code sections.

Test stubs have been used throughout computing history and as such one of those “tricks of the trade” that everybody is supposed to know about but is seldom discussed in depth in literature. Its prominent role in test driven development has changed this somewhat and I can recommend “xUnit Test Patterns: Refactoring Test Code” by Meszaros (2007). Meszaros introduces the *test double* concept as a generalization of various types: the spy, the mock, and the fake object.

The *test spy* records the interaction between the UUT and the external DOU (the indirect output), so our test code can later verify that the right methods were called with the right parameters in the right order. This usually requires a *retrieval interface*, that is, special methods to access the recording which are only declared in the test spy class.

The *fake object* is a light-weight implementation of a DOU, usually to replace a database, which is cumbersome to control from within our JUnit code.

Finally, the *mock* is a test stub or spy that can be generated by a mock library, so I do not have to code the stub/spy behaviour myself.

12.9 Selected Solutions

Discussion of Exercise 12.1:

The structure of object relations in STRATEGY and STATE are identical. However, their *intent* are different. STATE is to *Allow an object to alter its behavior when its internal state changes*. In contrast the intent of STRATEGY is *Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable*. The former speaks of behavior based on state, the latter of interchangeable algorithms. Here the question is what algorithm to use to calculate if it is weekend, hence it is the STRATEGY pattern.

Discussion of Exercise 12.2:

One2OneRateStrategy can also be classified as a test stub. The unit under test was the pay station and the rate strategy was the depended-on unit that this test stub ease the testing of. In this case, though, the reason for the stub was not that the indirect input (the calculated minutes) was not under testing control, but that the verification code, the asserts, was easier to write and maintain. In this light, we can state that the rate calculations were “altered” by the testing code for the purpose of easing the verification.

As it provides realistic behaviour (not a fixed behavior), I could argue that it is actually more a kind of *fake object*.

Discussion of Exercise 12.3:

I find that it is the *Representative Data* principle at work. The basic testing challenge is if the pay station interacts with the proper rate strategy object. The *Representative Data* principle states that I should use a small set and that each element should represent a particular aspect or a particular computational processing. In this case, there is no particular computational processing associated with the Gammatown rate policy that is not already demonstrated by the two other test cases.

12.10 Review Questions

Explain the difference between *direct input* and *indirect input*. Why is direct input not always simply values to be provided by method parameters?

Explain the terms *depended-on unit* (DOU) and *unit under test* and their relations to each other and to the test case code.

Explain what a *test stub* is: what is its purpose and how does it programmatically serve this purpose?

Relate how the use of test stubs relate to the ③-①-② process and the compositional proposal for handling variable behavior.

Explain what purpose the *test spy*, and the *fake object* has in testing.

12.11 Further Exercises

Exercise 12.4:

In a computer game the program typically validates that the user moves his pieces correctly, and testing correctness of this algorithm is of course important. Many games rely on throwing dice or other random behavior to determine the number of positions a piece may move.

1. Describe a design that allows testing the move validation algorithm in a die based game using a stub that encapsulate the die throwing algorithm. Describe the design using class and sequence diagrams.
2. Define a Java interface for the die throwing algorithm, and define two implementations: one that uses Java's random libraries to make real random die values in the range 1 to 6; and a second that can be set by JUnit testing code.

Exercise 12.5. Source code directory:

exercise/test-stub/sevensegment

The hardware producer of a *seven segment LED display* provides a very low-level interface for turning on each of the seven LED (light-emitting diode) segments on or off, see Figure 12.4.

Listing: exercise/test-stub/sevensegment/SevenSegment.java

```
/** Defines the contract for a seven segment LED display.
 */
public interface SevenSegment {
    /** turn a LED on or off.
     * @param led the number of the LED. Range is 0 to 6. The LEDs are
     * numbered top to bottom, left to right. That is, the top,
     * horizontal, LED is 0, the top left LED is 1, etc.
     * @param on if true the LED is turned on otherwise it is turned
     * off.
     */
    void setLED(int led, boolean on);
}
```

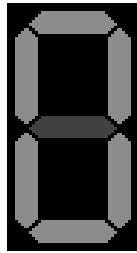


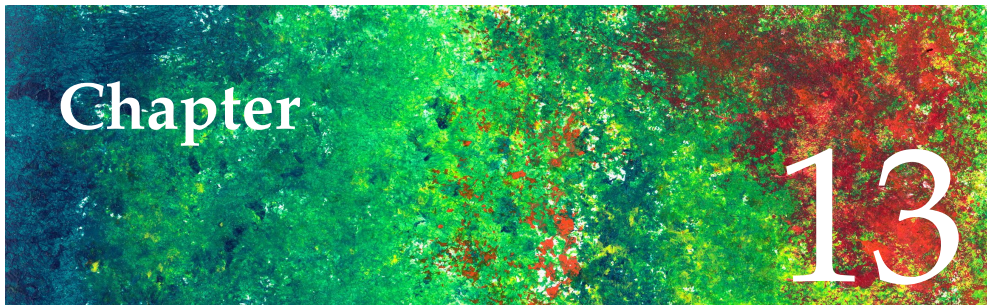
Figure 12.4: Seven segment LED showing 0.

Clearly, this is cumbersome in practice, so it is much better to define an abstraction that can turn on and off the proper LEDs for our ten numbers 0 to 9:

Listing: exercise/test-stub/sevenssegment/NumberDisplay.java

```
/** The interface to a seven segment LED that displays numbers 0-9. */
*/
public interface NumberDisplay {
    /** display a number on a seven segment. */
    * @param number the number to display.
    * Precondition: number should be in the range 0 to 9.
    */
    void display(int number);
}
```

1. Sketch how a TDD process, based upon automated testing and no visual inspection of the seven segment display, can develop a reliable implementation of `NumberDisplay`.
2. Classify the developed test double according to the classification defined in Section 12.6
3. Use TDD to develop an implementation of `NumberDisplay`.



Deriving Abstract Factory

Learning Objectives

New requirements are the reality of successful software development—and this time it has to do with *creating* objects. In this chapter, the objective is to establish the ABSTRACT FACTORY as a solution to the problem of creating variable types of objects. As was the case in the introduction of the two preceding design patterns, another objective is to show how this pattern also comes naturally from a compositional design philosophy. However, this time you will see that blindly following the ③-①-② process do not necessarily provide the best solution—you have to analyze the result and constantly refactor to optimize the design.

In this chapter, I will also spend some time on the actual refactoring to introduce the ABSTRACT FACTORY and the associated code, as it is a more complex design pattern compared to STRATEGY and STATE and people often misunderstand it or implement it incorrectly.

13.1 Prelude

So far, I have not spent much attention on the Receipt class—its responsibility is simply to know its value in minutes parking time. In this chapter, I will add another relevant responsibility to the receipt class, namely the ability to print some kind of visual representation of itself.

Receipt

- know its value in minutes parking time
- print itself

For the sake of simplicity, I will introduce a very simple scheme for printing, namely a single method:

```
public void print(PrintStream stream);
```

A `PrintStream` from the Java libraries allows character only printing, so a visual design of a parking receipt is here simply:

```
-----
----- P A R K I N G   R E C E I P T -----
          Value 049 minutes.
          Car parked at 08:06
-----
```

A more fancy graphical design would perhaps have been more appealing, but the principle is the same with a character based output medium.

For manual testing, I can pass the `System.out` print stream instance to a receipt object's `print` method and then manually inspect what gets printed. Fortunately, the Java IO libraries also provides the `ByteArrayOutputStream` that you can print to. This class is a stream that writes to a byte array that can then be converted to a string. Therefore I can use this to write automated tests of the `print` method.

Exercise 13.1: Sketch or program a JUnit test case for testing the `print` method. You should use the `ByteArrayOutputStream` and in order to “chunk” the resulting string into an array of individual lines you can use the following code:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
PrintStream ps = new PrintStream(baos);
// let the receipt print itself
receipt.print(ps);

// get the string printed to the stream
String output = baos.toString();
// split the string into individual lines
String[] lines = output.split("\n");
```

13.2 New Requirements

Change is the only constant in software development. Betatown would like to make some statistics on the receipts that parking meter checkers come across. To do this they require us to print a *bar code* on the receipts so the parking meter checkers can simply scan all receipts they see, instead of manually writing down data.

Thus I am faced with a new challenge: one of the products must issue a special type of receipts, different from standard receipts. Of course, developing software to print real bar codes is out of the scope of the present book, and my interest is in the way our pay station can reliably and flexibly issue different types of receipts. Thus in my treatment the actual implementation will only print a *Fake It* bar code like

```
-----
----- P A R K I N G   R E C E I P T -----
          Value 049 minutes.
          Car parked at 08:06
||  ||||| | || ||| | |  ||| | || |||| | || ||||
-----
```

That is, an extra line is added to the printed receipt that consists of a set of “|” and space characters.

13.3 One Problem – Many Designs

The above requirement conceptually boils down to having two different implementations of the `Receipt` interface as outlined in Figure 13.1. In the diagram I have made a small refactoring, namely to rename the present implementation named `ReceiptImpl` to `StandardReceipt`.

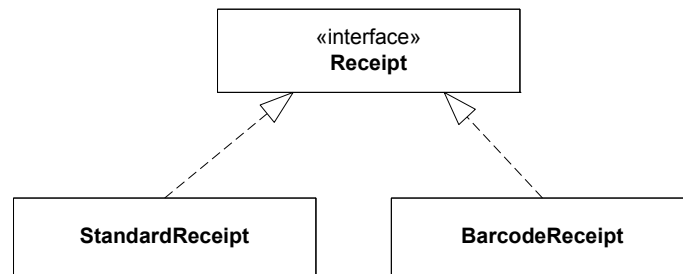


Figure 13.1: New types of Receipts.

Note that this requirement is independent of the previous requirements concerning rate calculations. I can therefore make a configuration table, see Table 13.1, that outlines how pay stations should be configured for the present three towns, stating the type of rate calculation and type of receipt that the town has required.

Table 13.1: The three different configurations of the pay station.

Product	Variability points	
	Rate	Receipt
Alphatown	Linear	Standard
Betatown	Progressive	Barcode
Gammatown	Alternating	Standard

The new requirement can of course be handled with all the same proposals as we have already analyzed in the strategy and state chapters: by source code copy, by parameterization, by polymorphism, and by a compositional approach. The argumentation is well-known by now, and I will therefore concentrate on the compositional proposal.

Exercise 13.2: Table 13.1 shows the configurations of the pay station but only the production variants (those that are in operation in the different towns). Update the table to show all variants, including those in the testing code.

13.4 A Compositional Proposal

In this section, I will crank the handles of the ③-①-② machine once again and come up with a compositional solution. However, be warned that this first attempt will

be flawed. Nevertheless, I will detail this failed attempt as it is important to understand *why* it fails: you appreciate good designs better if you are exposed to less good designs.

So, my first attempt goes like this:

- ③ *I identify a behavior that needs to vary.* The printing of receipts is the behavior that needs to vary. As receipts are objects in the pay station software, the verb “print” is actually the same as “instantiation”: it is the way I instantiate `Receipt` objects that needs to vary. In Alphatown and Gammatown the pay station should instantiate `StandardReceipt` objects while it should instantiate `BarcodeReceipt` objects in the Betatown configuration.
- ① *I state a responsibility that covers the behavior and express it as an interface.* One plausible way was to define a `ReceiptIssuer` interface whose responsibility it is to issue receipts, that is create `Receipt` objects. This responsibility is at present handled by the pay station so this would also remove one responsibility from it.
- ② *I compose the resulting behavior by delegating to subordinate objects.* The pay station should ask the receipt issuer to instantiate the receipt instead of doing it itself.

This leads to a design shown in Figure 13.2.

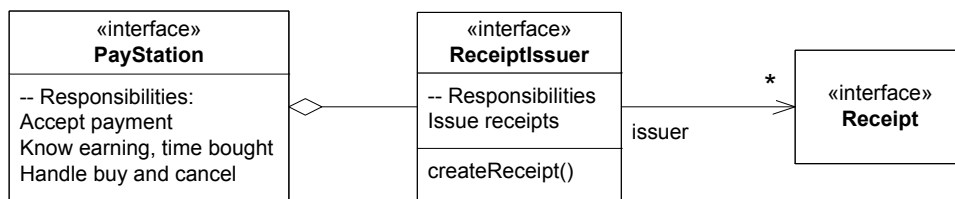


Figure 13.2: Factoring out the receipt issuing responsibility.

Something nags me though. Do I really need this additional `ReceiptIssuer` abstraction? If I look at the original design in Figure 4.3 in Chapter 4, it has quite a lot of similarity with the way I handled varying the rate calculations. In the original design, I have a pay station object that is associated with a receipt object; and in the strategy chapter I have a pay station associated with a rate strategy. Why can't I simply provide the pay station with the receipt types to issue; just as I provided the pay station with the rate calculation strategy to use? Something along the lines of configuration the Betatown pay station like this:

```

PayStation ps
= new PayStationImpl(new ProgressiveRateStrategy(),
                    new BarcodeReceipt());
  
```

This would do the trick, would it not? And it certainly would simplify the design a lot compared to the above one with the extra `ReceiptIssuer` class.

☞ Spend a few minutes considering what the difference is between issuing receipts and calculating rates before reading on. Looking into the code should provide you with a clue.

Copyrighted Material. Do Not Distribute.

It turns out that the situation is quite different and I cannot apply a STRATEGY pattern for receipts. The reason is found in the source code for `PayStationImpl`'s `buy` method:

Fragment: `chapter/abstract-factory/iteration-0/src/main/java/paystation/domain/PayStationImpl.java`

```
public Receipt buy() {
    Receipt r = new ReceiptImpl(timeBought);
    reset();
    return r;
}
```

The big problem is the `new` statement! The pay station does not just *use* a receipt object as is the case with the rate strategy; it *creates* a receipt object. Each receipt is unique and has its own value of parking time. Therefore it does not make sense to provide the pay station with a single receipt object during construction as it cannot use this object to create new ones¹. Second, the code that instantiates the pay station above does not compile. The `ReceiptImpl` constructor takes an argument, namely the number of minutes parking time to print on the receipt. This again highlights the difference between the pay station using an object versus creating new objects.

If I introduce an intermediate object, like `ReceiptIssuer`, I instead delegate the responsibility of creating receipts to it and can avoid the problem.

13.4.1 Iteration 1: Refactoring

OK, let me try to figure out if the design based on a `ReceiptIssuer` is a feasible path. I will confront it with reality: quickly develop it and see if it feels right, and be prepared to backtrack if it turns out bad before the costs get too high.

Test-driven development tells me to take small steps, and as was the case earlier, the best path is to refactor the existing design to introduce the new design and make all test cases pass, and next introduce the new bar code receipt.

- * refactor to introduce `ReceiptIssuer`
- * add bar code receipts to Betatown.

The present fixture in `TestPayStation` looks like this:

Fragment: `chapter/abstract-factory/iteration-0/src/test/java/paystation/domain/TestPayStation.java`

```
/** Fixture for pay station testing. */
@BeforeEach
public void setUp() {
    // Given a PayStation whose rate strategy is that
    // one cent buys one minute parking time
    ps = new PayStationImpl(coinValue -> coinValue);
}
```

Thus, according to my design, I must configure it with an issuer object:

```
@BeforeEach
public void setUp() {
    ps = new PayStationImpl(coinValue -> coinValue,
        new StandardReceiptIssuer());
}
```

¹I should mention that Java does provide a number of techniques to do this anyway. The `clone()` method defined in `Object` would allow me to get a new receipt object that I could change the state in and then use. This is actually the PROTOTYPE design pattern.

But—this design worries me quite a bit. The reason is that the *configuration responsibility* is assigned two *different* objects. The client code that defines the pay station (like `setUp()` above) is responsible for configuring the pay station with the proper rate calculation strategy to use. The responsibility of creating the proper receipt types to issue, however, is handled by the receipt issuer. It is “all configuration behavior” but handled by two different objects and thus in disjoint portions of the code. This is again potentially dangerous because a developer that needs to modify a given configuration may more easily miss the point that he or she should review and potentially change in *two* different places—for all the same reasons that I made against handling variable behavior using a parametric proposal. Therefore this solution is not very cohesive.

☞ Read the definition of cohesion and rephrase the argumentation above in terms of the cohesion property.

As I aim for a maintainable solution, it is important to achieve high cohesion and therefore I do not like the way things are moving. I see no other way but to undo the implementation and rethink my design. This is actually also the essence of a testing pattern so I do a small sidestep into TDD and present it.

TDD Principle: Do Over

What do you do when you are feeling lost? Throw away the code and start over.

The cost is not high here as I have barely begun designing any code. The difficult aspect of this testing pattern is applying it when you have spent three hours coding and it does not go well. Your inclination is to “keep your investment”. This pattern says it is better to view the three hours as an experiment that has taught you how *not* to do it and use this new knowledge to make a much better design!

Back to the “two ways of creating objects”. I have to rethink the responsibilities. I really would like that one object alone is responsible for creating *all* objects that are related to the pay station configuration. I would like to define a responsibility *To make objects* and collect all object creation in a single place. Such an abstraction is often called a *factory*:

PayStationFactory

- Create receipts
- Create rate strategies

Thus I would get a cohesive design: “Aha, a problem with object creation? I know where to look—in the factory.” Concrete factories then instantiate appropriate objects for each pay station product as shown on Figure 13.3. Armed with this new design proposal, I can once again start the refactoring iteration using a test list

- * refactor to introduce PayStationFactory
- * add bar code receipts to Betatown

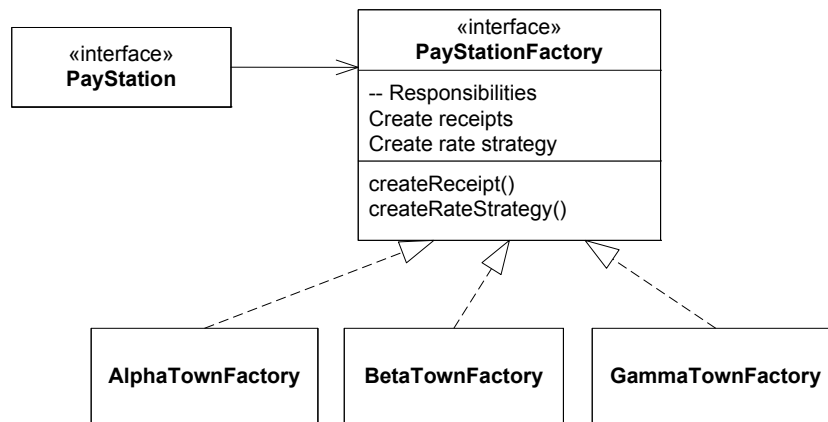


Figure 13.3: Factory handles object instantiation.

13.4.2 Iteration 1: Refactoring (Do Over)

I will make a special factory, named `TestTownFactory`, that configures the pay station to operate with the `One2OneRateStrategy` and `StandardReceipt`, as this is what all the test cases in the `TestPayStation` class assumes.

Fragment: chapter/abstract-factory/iteration-1a/src/test/java/paystation/domain/TestPayStation.java

```

/** Fixture for pay station testing. */
@BeforeEach
public void setUp() {
    // Given a PayStation whose rate strategy is that
    // one cent buys one minute parking time
    ps = new PayStationImpl(new TestTownFactory());
}
  
```

This, of course, does not compile. I therefore introduce the factory interface:

Listing: chapter/abstract-factory/iteration-1a/src/main/java/paystation/domain/PayStationFactory.java

```

package paystation.domain;
/** The factory for creating the objects that configure
    a pay station for the particular town to operate in.
*/

public interface PayStationFactory {
    /** Create an instance of the rate strategy to use. */
    public RateStrategy createRateStrategy();

    /** Create an instance of the receipt.
        * @param the number of minutes the receipt represents. */
    public Receipt createReceipt(int parkingTime);
}
  
```

and a `TestTownFactory` class that simply returns null objects to get to *Step 2: Run all tests and see the new one fail.*

Listing: chapter/abstract-factory/iteration-1a/src/test/java/paystation/domain/TestTownFactory.java

```

package paystation.domain;
  
```



```

/** Factory for making the pay station configuration
    for unit testing pay station behavior.
    */
class TestTownFactory implements PayStationFactory {
    public RateStrategy createRateStrategy() {
        return null;
    }
    public Receipt createReceipt(int parkingTime) {
        return null;
    }
}

```

Huhh—it still does not compile? Ahh, of course, I need to refactor the `PayStationImpl` to use the factory. Something like:

```

public class PayStationImpl implements PayStation {
    [...]
    /** the strategy for rate calculations */
    private RateStrategy rateStrategy;
    /** the factory that defines strategies */
    private PayStationFactory factory;

    /** Construct a pay station.
        @param factory the factory to produce strategies and receipts
        */
    public PayStationImpl(PayStationFactory factory) {
        this.factory = factory;
        this.rateStrategy = factory.createRateStrategy();
        reset();
    }
    [...]
    public Receipt buy() {
        Receipt r = factory.createReceipt(timeBought);
        reset();
        return r;
    }
    [...]
}

```

Compiling—still fails... The integration tests in `TestIntegration` still use the old constructor. I silence both test cases by configuring the pay station with the test town factory:

```
ps = new PayStationImpl(new TestTownFactory());
```

Again this is terribly wrong, but I *keep focus* on getting everything to compile. I know that the integration tests will not pass before I get the proper factories for Alphatown and Betatown working, so there is absolutely no fear in my mind that I will forget these fake configurations—the failed tests will tell me.

Compilation success. Good feeling... However, seven test cases fail. No wonder, as the create methods of the factory just returns null. The change is *Obvious Implementation*:

Listing: chapter/abstract-factory/iteration-1b/src/test/java/paystation/domain/TestTownFactory.java

```

package paystation.domain;
/** Factory for making the pay station configuration
    for unit testing pay station behavior.
    */

```



```
*/
class TestTownFactory implements PayStationFactory {
    public RateStrategy createRateStrategy() {
        return coinValue -> coinValue;
    }
    public Receipt createReceipt(int parkingTime) {
        return new StandardReceipt(parkingTime);
    }
}
```

I am moving forward—only two tests fail now: The integration tests. To make these pass, I need to make the proper factories for Alphatown and Betatown. Again, obvious implementation, for instance the one for Betatown:

Listing: chapter/abstract-factory/iteration-1b/src/main/java/paystation/domain/BetaTownFactory.java

```
package paystation.domain;
/** Factory to configure BetaTown.
 */
class BetaTownFactory implements PayStationFactory {
    public RateStrategy createRateStrategy() {
        return new ProgressiveRateStrategy();
    }
    public Receipt createReceipt( int parkingTime ) {
        return new StandardReceipt(parkingTime);
    }
}
```

(Remember, adding the proper bar code receipt is the next iteration.) Great, all tests now pass. However, it appears to me that I still have not made a factory for the Gammatown configuration. The old argumentation was that the interaction between Gammatown's rate strategy and the pay station was already well tested, but I have added complexity by introducing the factory, and I of course also need to implement the factory for Gammatown. However, this is not the purpose of the refactoring iteration, so I note it on the test list.

- * refactor to introduce PayStationFactory
- * add bar code receipts to Betatown
- * test the Gammatown configuration

13.4.3 Iteration 2: Unit Test Barcode Receipts

Looking at the test list item *add bar code receipts to Betatown* I realize that it contains two iterations: unit testing the new bar code receipt, and next integration testing that the Betatown pay station actually delivers the proper type of receipts. I note the last aspect on the test list to be done in a later iteration.

As stated in the introduction, the bar codes on Betatown's receipts are not real bar codes, merely an additional line added to the receipt to make it different from the standard receipt. I will therefore not dwell on this iteration, but refer you to the source code for the iteration.

Exercise 13.3: If you study the production code for iteration 2 you will notice that I have used a parametric proposal to handle the two variants of the receipts: standard and bar code. Sketch the design for handling the variation using a compositional approach, analyze benefits and liabilities, and argue why a parametric solution is better in this special case.

13.4.4 Iteration 3: Integration Tests

I still have no test cases that tests that for instance the Gammatown pay station is actually configured properly—and as a result no tests that drive the development of Gammatown’s factory. This is the purpose of this iteration. I will, however, only describe the steps abstractly and leave the TDD process to you or you may find the outcome in the chapter’s source code folders.

Opening `TestIntegration` I note that the original integration tests for Alphatown and Betatown actually only test that the proper rate strategies are used. They must be updated to validate that also the proper receipts are issued. It is not the focus of this iteration, so I note it on the test list.

In my TDD process for Gammatown, I make a test case that tests both that the proper rate strategy is used as well as the proper type of receipt is issued. Next, I use the failed test case to introduce the `GammaTownFactory`.

Though I have kept my integration test for `GammaTown`, it is actually quite complex and the resulting code is not very *Evident Test*: I have to inject my `FakeDecisionStrategy` into the factory ala:

```
// Given a paystation to be the Gamma town pay station , BUT
// using the stub to set for normal working days
ps = new PayStationImpl(new GammaTownFactory() {
    public RateStrategy createRateStrategy() {
        return new AlternatingRateStrategy(
            new LinearRateStrategy(),
            new ProgressiveRateStrategy(),
            new FixedDecisionStrategy(false));
    }
});
```

The bottom line is that this test case is more complex and sprawls over many more code lines than the actual lean and analyzable `GammaTownFactory` code:

```
class GammaTownFactory implements PayStationFactory {
    public RateStrategy createRateStrategy() {
        return new AlternatingRateStrategy(
            new LinearRateStrategy(),
            new ProgressiveRateStrategy(),
            new ClockBasedDecisionStrategy());
    }
    public Receipt createReceipt(int parkingTime) {
        return new StandardReceipt(parkingTime);
    }
}
```

This is one of the cases in which, from a TDD viewpoint, you may argue for not writing the test case at all, and leave the reliability to be handled by manual and system testing. From a classical testing point of view, in which tests are written to ensure your production code is reliable, it makes sense to write it.

13.4.5 Iteration 4: City Configurations

Finally, the integration test for both Alphatown and Betatown lacks tests that validate that they issue the right type of receipts. I augment these integration tests with the validation code. During this process I discover that I can refactor the testing code to avoid duplication of the code that verifies the type of receipt issued. Also the names of the test cases are updated to reflect the type of test.

During this iteration, I discover that I had actually configured Betatown with a standard receipt! Tests are good. . .

13.5 The Compositional Process

Even though the initial attempt at using the ③-①-② process lead me into a less cohesive design, the resulting design is nevertheless the same principles applied, and the result is the ABSTRACT FACTORY pattern. The line of thought was:

- ③ *I identified some behavior, creating objects, that varies between different products.* So far products vary with regards to the types of receipts and the types of rate calculations.
- ① *I expressed the responsibility of creating objects in an interface.* `PayStationFactory` expressed this responsibility.
- ② *I let the pay station delegate all creation of objects it needs to the delegate object, namely the factory.* I can define a factory for each product variant (and particular testing variants), and provide the pay station with the factory. The pay station then delegates object creation to the factory instead of doing it itself.

If you look over the production code that is common to all pay station variants, you will see that it *only* refers to, and collaborates with, delegates through their interfaces. In no place does it create objects, cast references to concrete types, nor declare instance variables by a class type. Therefore, it does not depend on concrete types at any level. As argued, this leads to a low coupling and this principle is so important that it has received its own term.

Definition: Dependency inversion principle

High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions. (Martin 1996)

Rephrasing the definition for our case, it states that the pay station code shared by all variants (the “high level modules”) should not depend on the variant’s implementation (the “low level modules”), but only on interfaces (the abstractions).

Of course, some part of the code then has to make the actual binding, i.e. tell the high level modules the actual low level modules to communicate with. This binding process also has a name:

Copyrighted Material. Do Not Distribute.

Definition: Dependency injection

High-level, common, abstractions should not themselves establish dependencies to low level, implementing, classes, instead the dependencies should be established by injection, that is, by client objects. (Fowler 2004)

In our case, the factory object is asked to create concrete objects to be used by the pay station, thus the factory injects the dependencies. Dependency injection is central in frameworks as discussed in Chapter 32.

13.6 Abstract Factory

The intent of the ABSTRACT FACTORY is to *Provide an interface for creating families of related or dependent objects without specifying their concrete classes*. In our pay station case, it is the family of objects that determines the variant of our pay station product. The general structure and condensed presentation of the pattern is given in design pattern box 13.1 on page 59. The central roles in abstract factory is the **client** that must create a consistent set of **products**. In our case, the client is the pay station and the products are receipts and rate strategies. Instead of creating the products itself, the role **abstract factory** does it. In the classic design pattern book, *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al. 1995), the two previous design patterns, STRATEGY and STATE, are classified as *behavioral* design patterns while ABSTRACT FACTORY is classified as *creational*. The intention of creational patterns is of course to *create objects*. Behavioral patterns deal with variability in the behavior of the system and the use of state and strategy in the pay station are clear examples of this.

The benefits of using an abstract factory in our designs are:

- *Low coupling between client and products*. There are no new statements in the client to make a high coupling to particular product variants. Thus the client only communicates with its products through their interface.
- *Configuring clients is easy*. All you have to do is to provide the client with the proper concrete factory. In our case, the pay station is configured simply by passing the proper pay station factory object during construction.
- *Promoting consistency among products*. Locality is important in making software easy to understand and maintain. The individual factories encapsulate the pay station's configuration: To review how a Betatown configuration looks like, you know that the place to look for it is in `BetaTownFactory`—and no other place. Thus any defects in configuration are traceable to a single class in the system. And as you are not required to look and change in five or ten different places in your code in order to make a configuration, the risk of introducing defects is substantially lessened.
- *Change by addition, not by modification*. I can introduce new pay stations product types easily as I can add new rate strategies and new receipt types, and provide

the pay station with these by defining a new factory for it. Existing pay station code remains unchanged giving higher reliability and easier maintenance. Note, however, this argument is only true when it comes to introducing new *variants* of the existing rate strategies and receipt types! See the liability discussion below...

- *Client constructor parameter list stays intact.* Even if I need to introduce new products in the client, the client's constructor will still only take a single factory object as its parameter. Compare this to the practice of adding more and more configuration parameters to the constructor list.

Of course, ABSTRACT FACTORY has liabilities, some of which should be well known to you by now.

- *It introduces extra classes and objects.* Compared to a parametric solution where decision making is embodied in the client code I instead get several new interfaces and classes and thus objects: the factory interface as well as implementation classes. Therefore the pattern requires quite a lot of coding—you should certainly not use it if you only have a single type of product.
- *Introducing new aspects of variation is problematic.* The problem with abstract factory is that if I want to introduce new aspects that must be varied (logging to different vendor's databases, accepting other types of payment, etc.) then the abstract factory interface must be augmented with additional create-methods, and all its subclasses changed to provide behavior for them. This is certainly *change by modification*.

A classical example of the use of abstract factory is the Java Swing graphical user interface toolkit. A major problem for Java is that it runs on a variety of platforms including Windows, Macintosh, and Unix. However, Swing allows you to write a graphical user interface form (a dialog with text fields, radio buttons, drop-down list boxes, etc.) that will run on all the supported platforms. Thus to instantiate an Swing button the Java run-time library has to make a decision whether to create a Win32 button, a Mac button, or a Unix windows manager button; to instantiate a list box is has to make a similar decision; etc. Thus, the source code could be thick with zillions of if-statements—but it is all handled elegantly by an ABSTRACT FACTORY. The Swing code delegates any instantiation request for a graphical user component to its associated factory. The factory has methods for creating buttons, list boxes, radio buttons, text fields, etc. Each concrete factory, one for Win32, one for Mac, etc., then creates the proper graphical component from each platform's graphical toolbox.

In this light, it is also fair to say that the use of a factory to create the rate strategy in the pay station is sort of a special case because it only happens initially in contrast to receipts that are created continuously. Usually, abstract factory is considered the solution to the "continuous creation" problem. Still, it makes sense to group all object creation responsibility into a single abstraction as I have done in the pay station.

13.7 Summary of Key Concepts

Object-oriented systems need to create objects. However, when you strive to make a loosely coupled design you face the problem that the `new` statement expresses the tightest coupling possible: you once and for all bind an object reference to a particular concrete class.

```
Receipt r = new StandardReceipt(30);
```

While the first part of this statement `Receipt r` is loosely coupled as you only rely on the interface type `Receipt` the exact opposite is true for the right hand side of the assignment. The creational pattern `ABSTRACT FACTORY` provides a compositional solution to the problem of loosening the coupling between a client and the concrete products it needs to create, by delegating the creation responsibility to an instance of a factory. Furthermore the factory becomes a localized class responsible for configuration leading to a design that promotes consistency among products. A liability of the pattern is that it is somewhat complex and requires quite a lot of coding.

This chapter also showed that the ③-①-② process should not be used mechanically without considering the resulting design's cohesion and coupling. Always consider how the existing design could be refactored in such a way that the overall compositional design has high cohesion and low coupling. In our case, the creation of rate strategies had to be included in the analysis.

13.8 Selected Solutions

Discussion of Exercise 13.1:

```
/** Test that the receipt's show method prints proper info */
@Test public void shouldPrintReceiptsCorrectly() {
    Receipt receipt = new ReceiptImpl(30);
    // Prepare a PrintStream instance that lets me inspect the
    // data written to it.
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    PrintStream ps = new PrintStream(baos);
    // let the 30 minute print itself
    receipt.print(ps);

    // get the string printed to the stream
    String output = baos.toString();
    // split the string into individual lines
    String[] lines = output.split("\n");
    // test to see that the receipt consist of five lines
    assertEquals( 5, lines.length );
    // test parts of the contents
    assertEquals( "---", lines[0].substring(0,3) );
    assertEquals( "---", lines[4].substring(0,3) );
    assertEquals( "P A R K I N G", lines[1].substring(9,22) );
    // test the receipt's value
    assertEquals( "030", lines[2].substring(22,25) );
    // test that the format of the "parking starts at" time
    // is plausible
    String parkedAtString = lines[3].substring(28,33);
```

```

assertEquals( ':', parkedAtString.charAt(2) );
// if the substring below is not an integer a
// NumberFormatException is thrown which will
// make JUnit fail this test
Integer.parseInt( parkedAtString.substring(0,2) );
Integer.parseInt( parkedAtString.substring(3,5) );
}

```

This is a plausible test case for print, but a major problem with this test code is that it is longer than the corresponding production code thus there is a high probability that the defects are in the testing code.

Discussion of Exercise 13.2:

The testing code includes also the use of the `One2OneRateStrategy`, and the testing of Gammatown actually includes an additional dimension of variance, namely the choice of strategy to determine if it is weekend or not. Therefore a complete table will include these aspects.

Product	Variability points		
	Rate	Receipt	Weekend
Alphatown	Linear	Standard	–
Betatown	Progressive	Barcode	–
Gammatown	Alternating	Standard	Clock
PayStation unit test	One2One	Standard	–
Gammatown rate unit test	Alternating	–	Fixed

Here “–” means that the selection is not applicable.

Discussion of Exercise 13.3:

The only difference in the two receipt types is in one additional line printed. A compositional approach would encapsulate the responsibility of outputting this line in an interface and use delegation to print it. Something along of the following line embedded in the receipt's print method:

```

[... ]
stream.println("                Car parked at "+nowstring);
additionalInfoPrinter.print(stream);
stream.println("-----");

```

One concrete implementation would then print nothing and the other print the bar code. The problem I see with this solution is that I do not see viable future variations: what is the obvious new requirement for something new to print? And with only two variations, the parametric solution:

```

[... ]
stream.println("                Car parked at "+nowstring);
if ( withBarCode ) {
    stream.println("||  ||||| | || ||| || |  ||| | || |||| | || ||||");
}
stream.println("-----");

```

is much shorter and does not require additional interfaces and implementation classes. If, at a later time, new requirements pop up with regards to this variability point, I would consider refactoring this design.

13.9 Review Questions

Why can't the pay station simply take a receipt object as parameter in the constructor and use this; like it did with the rate strategy?

What is the ABSTRACT FACTORY pattern? What problem does it address and what solution does it suggest? What are the roles involved? What are the benefits and liabilities?

Argue why ABSTRACT FACTORY is an example of using a compositional design approach.

Define the *dependency inversion principle* and *dependency injection* and why they are considered important principles. Argue how it relates to the use of the factory object in the pay station case.

13.10 Further Exercises

Exercise 13.4:

Walk in my footsteps. Develop the new product variant that has a new receipt type with a (fake it) bar code.

Exercise 13.5:

In the present design, each new combination requires at least one new class, namely a new instance of the `PayStationFactory`. Thus, if 24 product combinations are required, there will be 24 factories.

Outline a number of different techniques to avoid this multitude of factories. For each describe benefits and liabilities.

Exercise 13.6:

The first graphical user interface for Java was the Abstract Window Toolkit or AWT. AWT allowed graphical applications to be written once and then allowed to run on multiple computing platforms: Macintosh, Windows, and UNIX Motif. A central challenge for AWT was thus to couple the AWT abstractions to the concrete graphical elements of the underlying platform. That is, the statement:

```
java.awt.Button b = new java.awt.Button("A Button");
```

must create a button object that in turn creates a Win32 button when running on the windows platform; or creates a Mac toolkit button on the Mac OS; etc.

Sketch how the implementation of `java.awt.Button` may use an ABSTRACT FACTORY to create the proper buttons depending on the platform.

Note: The complete solution to this problem is complex and requires several of the design patterns mentioned in part , so focus on the ABSTRACT FACTORY aspect only.

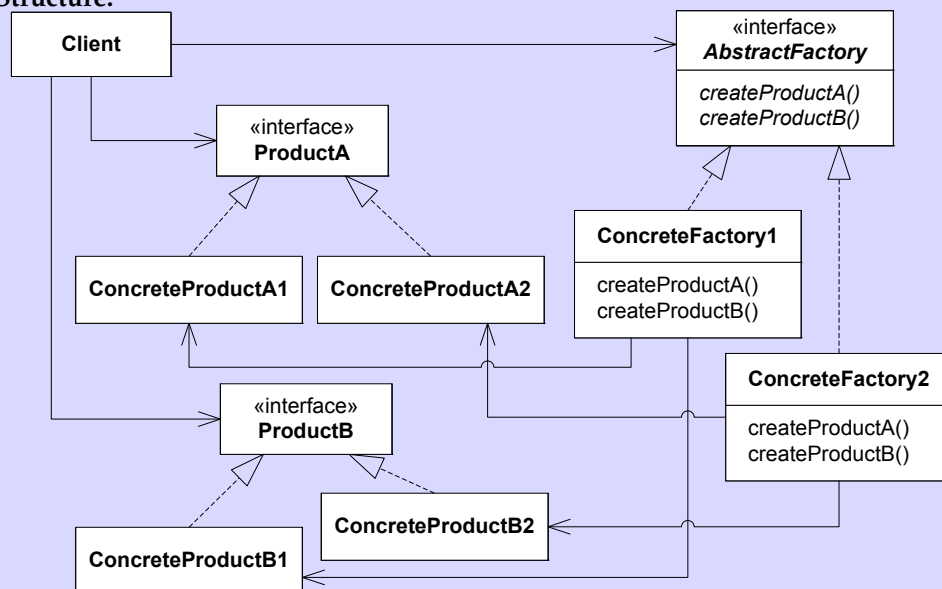
[13.1] Design Pattern: Abstract Factory

Intent Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Problem Families of related objects need to be instantiated. Product variants need to be consistently configured.

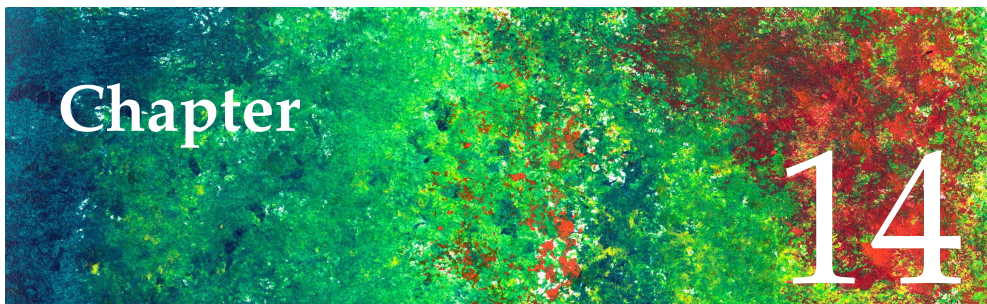
Solution Define an abstraction whose responsibility it is to create families of objects. The client delegates object creation to instances of this abstraction.

Structure:



Roles **Abstract Factory** defines a common interface for object creation. **ProductA** defines the interface of an object, **ConcreteProductA1**, (product A in variant 1) required by the client. **ConcreteFactory1** is responsible for creating **Products** that belong to the variant 1 family of objects that are consistent with each other.

Cost - Benefit *It lowers coupling between client and products as there are no new statements in the client to create high coupling. It makes exchanging product families easy by providing the client with different factories. It promotes consistency among products as all instantiation code is within the same class definition that is easy to overview. However, supporting new kinds of products is difficult: every new product introduced requires all factories to be changed.*



Pattern Fragility

Learning Objectives

Naturally, design patterns must be implemented in our production code: they are not just fancy ideas of software designers. The objective of this chapter is to highlight the importance of *getting the implementation right* as I will demonstrate how even small errors may cripple the advantages a pattern was supposed to have. I will also provide a small (and by no means complete) set of mistakes that occur in practice.

14.1 Patterns are Implemented by Code

Design patterns organize and structure code in a particular way. The structure is both in terms of the static aspects of the architecture, division of code into classes and interfaces, as well as the dynamic aspects, assigning responsibilities and defining object interactions. The reason I use a certain design pattern to structure my code is that software engineering knowledge and experience has shown that this particular structure has a certain balance between benefits and liabilities that my analysis shows is the appropriate one for the design problem I am tackling.

The bottom line is that applying design patterns is not a goal in itself—patterns are means to achieve a certain balance of qualities in my software. Most design patterns focus on maintainability and flexibility, especially those defined in the original GoF book. I do not get flexible software by writing in the documentation that I have used the STRATEGY pattern nor do I get it from naming one of my classes `LinearRateStrategy`. I get it because the structure and interactions between objects at run-time obey those rules that are set forward by the STRATEGY pattern. The consequence is that you should pay great attention to detail when you introduce design patterns into your designs or make use of those introduced by others. Otherwise you may easily “amputate” the pattern which leaves you with all the liabilities and none of the benefits. I call this *pattern fragility*:

Definition: Pattern fragility

Pattern fragility is the property of design patterns that their benefits can only be fully utilized if the pattern's object structure and interaction patterns are implemented correctly.

Below I will discuss some of the coding mistakes that may invalidate the benefits of the STRATEGY pattern as an example.

14.2 Declaration of Delegates

A primary purpose of many design patterns is to strengthen flexibility and reusability by allowing a particular object that answers a given request to be replaced by another, like the pay station that can interact with several different instances of `RateStrategy`.

This means declarations must be written in terms of *interfaces* instead of concrete classes. If a programmer accidentally declares an object by its concrete class the flexibility property is essentially lost. Consider the following example from the pay station. If I am working intensively on the Betatown pay station I may write the declaration of the rate strategy in the `PayStationImpl` as:

```
public class PayStationImpl implements PayStation {
    [...]

    /** the strategy for rate calculations */
    private ProgressiveRateStrategy rateStrategy;

    [...]
}
```

instead of declaring the delegate object by its interface type:

```
public class PayStationImpl implements PayStation {
    [...]

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

    [...]
}
```

If you develop without automated tests or if you do not run the Alphatown and Gammatown test cases then your development will proceed fine. The problem is, however, that the pay station implementation is no longer flexible—it is hardwired to Betatown's rate strategy. Essentially the simple declaration mistake has removed all benefits of the STRATEGY pattern but sadly retained all its liabilities: more interfaces and classes to consider, and more objects in play.

Key Point: Declare delegate objects by their interface type

Declare object references that are part of a design pattern by their interface type, never by their concrete class type.

🔍 Compare this discussion with the dependency inversion principle on page 53.


14.3 Binding in the Right Place

As argued above, the declaration of the delegate must be in terms of the interface to support flexibility. Essentially, this is the way to ensure loose coupling: the coupling relation between the pay station and the rate strategy is weak because the pay station only assumes the contract / the interface. However, the coupling must be made at some time: a concrete rate strategy object must be instantiated and associated with the pay station implementation. Care must be exercised to make this association in the proper place just as the declarations must be made with care. Consider the following pay station implementation code fragment I have seen students write:

```
public class PayStationImpl implements PayStation {
    [...]
    public void addPayment( int coinValue )
        throws IllegalCoinException {
        switch ( coinValue ) {
        case 5:
        case 10:
        case 25: break;
        default:
            throw new IllegalCoinException("Invalid coin: "+coinValue);
        }
        insertedSoFar += coinValue;
        RateStrategy rateStrategy = new LinearRateStrategy();
        timeBought = rateStrategy.calculateTime(insertedSoFar);
    }
    [...]
}
```

Again, no test case for Alphatown with its linear rate policy will fail. The system is behaviorally correct. But the pay station implementation is no longer flexible with respect to new rate policies, it is hardwired to the Alphatown's policy. I have none of the pattern's benefits but all of its liabilities.

From an abstract perspective, the problem here is that the `PayStationImpl` code is shared by all pay station products. It is specifically designed for flexibility. It therefore must only be weakly coupled to its delegate objects, like the rate strategy and receipt type. The direct instantiation in the above code makes a tight coupling and invalidates the implementation's flexibility and generality. The binding, i.e. the instantiation of delegate objects, must be made somewhere of course but it cannot be done in a part of the code that is reused across all product variants.

 This is actually the *dependency injection* principle (page 54). Read this principle again and argue how it relates to the outlined problem.

The question then is where should the binding be made? Well, the obvious answer is that it must be in the part of the code that is product specific. In the pay station system I started out instantiating the proper rate strategy in the testing code and then passed it on in the pay station's constructor. Later I refactored to use an abstract factory. The latter I find a good solution because it is highly cohesive: an object whose only responsibility is to create the proper delegates for a specific configuration.

Key Point: Localize bindings

There should be a well-defined point in the code where the creation of delegate objects to configure the particular product variant is put.

ABSTRACT FACTORY and other creational patterns are obvious units that have the configuration and binding responsibility. Typically, configuration and binding may also be part of the “main” unit of the code, that is, the part that executed first when a system starts, like in the main method of Java programs, the JFrame constructor of Swing applications, etc.

In some patterns, the binding is the issue of the pattern. For instance the STATE pattern’s intent is altering the binding. Still, it is important that you make a careful analysis of which object that makes the binding and make it explicit.

14.4 Concealed Parameterization

Let me assume that the mistake I showed in the previous section has somehow survived unnoticed in my production code. The instantiation of the LinearRateStrategy directly in the addPayment method works fine for Alphatown and no new releases of the software have been made for Betatown and Gammatown for ages. Suddenly one of the colleagues of my team has to dig out the old code again because Betatown has requested some changes. Now she realizes that PayStationImpl does not work properly for Betatown anymore. She has not really worked with the pay station code before, and is not very well trained in design patterns—and Betatown wants the new software update urgently. In a situation like this, parameterization may easily sneak in:

```
public class PayStationImpl implements PayStation {
    [...]
    public void addPayment( int coinValue )
        throws IllegalArgumentException {
        switch ( coinValue ) {
        case 5:
        case 10:
        case 25: break;
        default:
            throw new IllegalArgumentException("Invalid coin: "+coinValue);
        }
        insertedSoFar += coinValue;
        RateStrategy rateStrategy;
        if ( town == Town.ALPHATOWN ) {
            rateStrategy = new LinearRateStrategy ();
        } else if ( town == Town.BETATOWN ) {
            rateStrategy = new ProgressiveRateStrategy ();
        }
        timeBought = rateStrategy.calculateTime(insertedSoFar);
    }
    [...]
}
```

The solution is behaviorally correct. All test cases pass. The design is terrible of course as it disables the benefits of the compositional design and introduced a parametric design instead. The result is the worst of two worlds: we got all the added

complexity and added objects of the compositional design and all the *change by modification* of the parametric design.

Key Point: Be consistent in choice of variability handling

Decide on the design strategy to handle a given variability and stick to it.

This does not mean that you should handle all variability by a compositional approach. It simply means you must handle one specific variability consistently with the same technique. For instance while I generally favor compositional designs I did choose a parametric design for handling the “bar code or not” variability of the receipts as outlined in Section 13.4.3.

14.5 Avoid Responsibility Erosion

Software grows with the users’ requests: *Software changes its own requirements*. Minor requests are often handled by minor “fixes” to the production code. The problem is that many “minor” fixes over a long period of time have a distinct tendency to erode the design. It is often termed *software aging* or *architectural erosion*. The bad decision to mix compositional and parametric design in the previous section is a typical example of erosion.

A particular type of erosion is that some object of a particular variant needs to do a bit extra compared to the other variants. This “extra” may be handled by adding an additional method. As a contrived example, let me state a really weird requirement from Gammatown that the receipt should include a text explaining how the rate was calculated. I may therefore add another method to `AlternatingRateStrategy`:

```
public class AlternatingRateStrategy implements RateStrategy {
    [...]
    public int calculateTime( int amount ) {
        if ( decisionStrategy.isWeekend() ) {
            currentState = weekendStrategy;
        } else {
            currentState = weekdayStrategy;
        }
        return currentState.calculateTime( amount );
    }

    public String explanationText() {
        if ( currentState == weekdayStrategy ) {
            return [the explanation for weekday];
        } else {
            return [the explanation for weekend];
        }
    }
}
```

Because this method is defined in the subtype and not in the `RateStrategy` interface I cannot simply invoke this method but must check the object type before using it:

```
if ( rateStrategy instanceof AlternatingRateStrategy ) {  
    AlternatingRateStrategy rs =  
        (AlternatingRateStrategy) rateStrategy;  
    String theExplanation = rs.explanationText();  
    [use it somehow]  
}
```

This way the compositional design is again moving towards a parametric design where the object's type becomes the parameter to switch upon. Therefore there is the danger that the design creeps towards a parametric one.

An alternative is to move the `explanationText` method up into the `RateStrategy` interface. It is better as I avoid the object type switching code but now I am in the process of adding responsibilities to the rate strategy role. It is defined as a rate calculator but now it also must handle human readable text. It may be the proper decision but make it based on an analysis instead of just making it happen by accident.

Key Point: Avoid responsibility erosion

Carefully analyze new requirements to avoid responsibility erosion and bloating interfaces with incohesive methods.

14.6 Discussion

One interesting observation is that all the above examples of coding uses the STRATEGY pattern at a superficial glance: there is the context object, the strategy interface, and the concrete strategy objects. The devil is in the detail. One mistaken declaration, a `new` statement in the wrong place, a quick-and-dirty parameterization, and the pattern benefits are nevertheless lost. The examples above are *not* examples of using the STRATEGY pattern because its intent has been lost in coding mistakes.

Above I have talked about how coding practices or mistakes may invalidate the benefits of patterns. *Patterns are fragile*. It is therefore important that the programmers that handle the code with design patterns are well trained: know the patterns, know the roles and interaction patterns they involve, and know the implications of how they are coded.

14.7 Summary of Key Concepts

Design patterns manifest themselves in our code. While they do have some inherent benefits in terms of maintainability and reliability, these benefits are easily destroyed by coding mistakes. Such mistakes usually appear because people do not deeply understand the patterns but they can also appear because of carelessness. The problem about such coding errors is that the liabilities remain, like an overhead of interfaces, classes and objects, but the benefits are gone.

Some common problems are forgetting to declare instance variables by their interface type; making bindings to delegate objects in the common/shared implementation; or accidentally mixing different design strategies for handling variability.

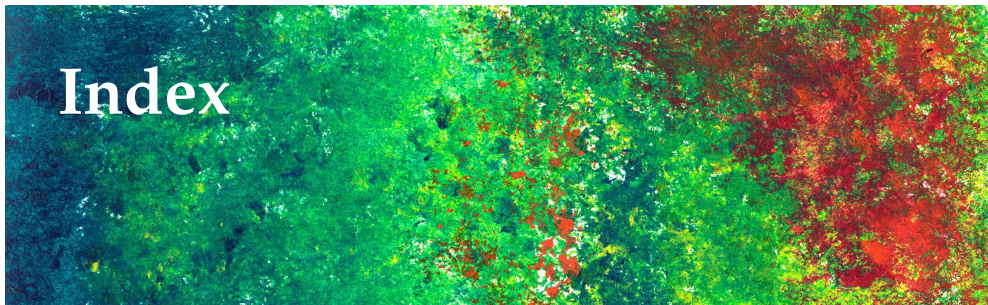
14.8 Review Questions

How is *pattern fragility* defined and what does the concept cover? Name some mistakes in code that spoil the benefits of design patterns.



Bibliography

- Fowler, M. (2004, January). Inversion of Control Containers and the Dependency Injection Pattern. <http://martinfowler.com>.
- Freeman, S., T. Mackinnon, N. Pryce, and J. Walnes (2004). Mock Roles, Not Objects. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, pp. 236–246. ACM Press.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Martin, R. C. (1996, May). The Dependency Inversion Principle. *C++ Report 8*.
- Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.



- ③-①-② process, 28–29
- ABSTRACT FACTORY, 59
- aging, of software, 65
- architectural erosion, 65
- behavioral pattern, 54
- bloat, of methods, 20
- blob, 14
- cohesion
 - composite example, 20
- compositional variability, 14–15, 28
- configuration table, 45
- creational pattern, 54
- depended-on unit, 27
- dependency injection, 54
- dependency inversion, 53
- design pattern
 - fragility, 62
- direct input, 26
- Do Over*, 48
- DOU, *see* depended-on unit
- erosion, architectural, 65
- fake object, 32
- fragility, pattern, 62
- implementation inheritance, 9
- indirect input, 7, 26
- interface inheritance, 9
- method bloat, 20
- mock object, 32
- multiple inheritance, 9
- pattern fragility, 62
- polymorphic variability, 8–13
- Retrieval interface, 34
- ripple effect, 14
- software aging, 65
- STATE, 23
- STATE, 18–19
- state machine, 19
- test double, 32
- test spy, 32
- test stub, 28
- transition table (state machine), 19
- turnstile, example, 19
- UUT, *see* unit under test
- variability
 - compositional, 14–15, 28
 - polymorphic, 8–13



Index of Sidebars/Key Points

- Avoid responsibility erosion, 66
- Be consistent in choice of variability handling, 65
- Declare delegate objects by their interface type, 62
- Design patterns are defined by the problems they solve, 19
- Interface and Implementation Inheritance, 9
- Localize bindings, 64
- Object collaborations define compositional designs, 15
- Test doubles make software testable, 32
- Wind Computation Testing in SAWOS, 37