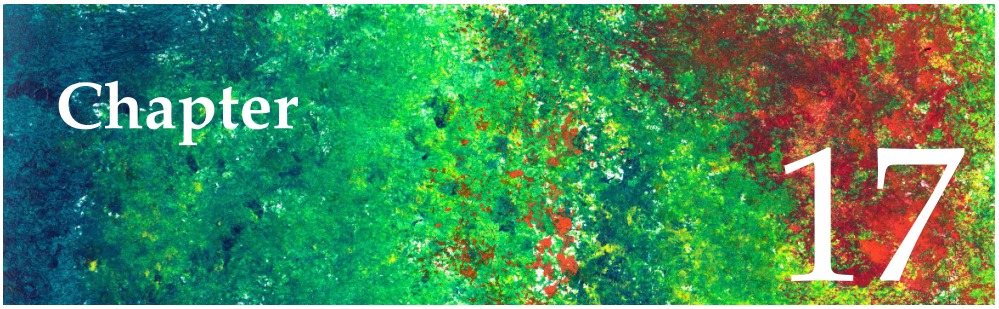

Multi-Dimensional Variance

Learning Objectives

So far in this learning iteration, I have focused on principles and concepts of compositional design. In this chapter I will return to the pay station case and the practical aspects. The learning focus is *variability along several dimensions*, i.e. when our production code must support combinations of variable aspects. Even in simple systems, you often see this kind of combined variability: a system must interface different types of hardware, run both in a production and test environment, use different types of persistent storage, handle different customer requirements, etc.

17.1 New Requirement

Alphatown is creative and comes up with a pretty reasonable new requirement. The value shown on the display is presently the number of minutes parking time that the entered amount entitles to. However, people prefer thinking in terms of the time when parking expires. Thus they want us to change the pay station so this value is shown on the display instead. As the hardware display can only display 4 digits, they want the time to be shown in the 24-hour clock format. That is, if I buy 10 minutes of parking time at 5:12 PM then instead of the display reading "0010" it should read "1722" to show that parking end time is 17:22 in the 24-hour clock. An example of the display output is shown in Figure 17.1.

17.2 Multi-Dimensional Variation

Analyzing the problem, it is apparent that I actually have a software system that is required to *vary along a set of distinct dimensions*. These dimensions are

Figure 17.1: Displaying parking end time.

- *Rate calculation.* There are several different requirements to how the pay station must calculate rates: Linear, progressive, alternating, etc.
- *Receipt information.* There are at the moment two requirements regarding the information on the receipts: the "standard" and one with bar code.
- *Display output.* There are requirements of what output the pay station should give on the display: Either number of minutes parking time bought or the time when parking must end.
- *"Weekend" control.* Our team must be able to get control of whether the pay station believes it is the weekend or not in order to bring the Gammatown's alternating rate calculation under full testing control.

If I restrict myself to the non-testing related three dimensions, I can describe each product variant as the proper configuration of the three variability points in a **configuration table**:

	Variability points		
Product	**Rate**	**Receipt**	**Display**
Alphatown	Linear	Standard	End time
Betatown	Progressive	Barcode	Minutes
Gammatown	Alternating	Standard	Minutes

This way, a product variant becomes a *point* in this three dimensional variability space as shown in Figure 17.2. The figure plots the Alphatown variant at the (Linear rate, standard receipt, end time display) point in the coordinate system of the pay station's variability space. The variability space has three dimensions: Rate policy, receipt type, and display output type.

The interesting aspect is that these dimensions are *independent* of each other. There is no logical binding between them that dictates that if a customer wants, say, linear rates, then they are forced to use, say, end time display. It then follows that all combinations are valid, legal, and indeed possible. I will term this **multi-dimensional variability** which denotes variability of multiple, independent, aspects of the software product.

The number of possible variants of the current pay station design is already $3 \times 2 \times 2 = 12$ (3 different rate policies, 2 receipt variations, and 2 display policies). If I add, say, two new rate policy variants the equation yields 20 combinations—the number of combinations grows rapidly—I have a **combinatorial explosion** of variants.

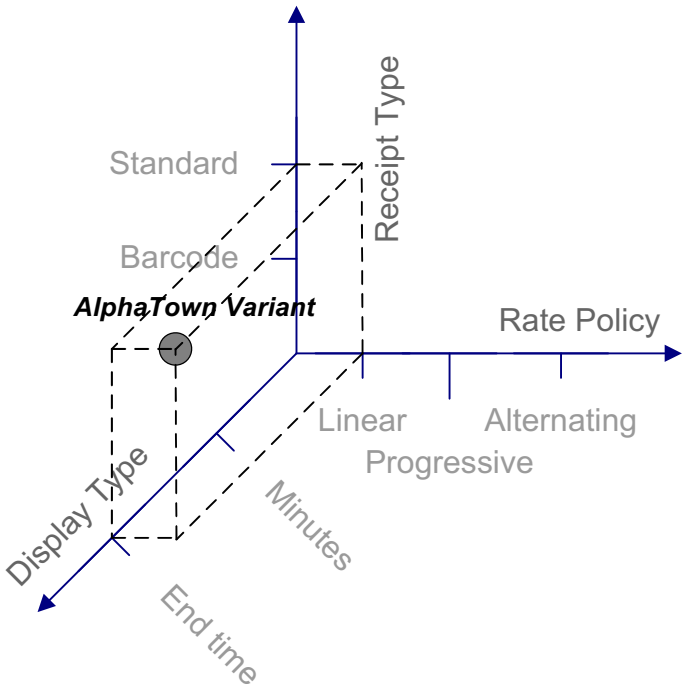

Figure 17.2: The Alphatown variant plotted in varibility space.

17.3 The Polymorphic Proposal

The polymorphic approach does not handle multi-dimensional variability well. In Figure 17.3 is shown the potential development of the class hierarchy for the three product variants. The classes containing the Alphatown, Betatown, and Gammatown variants are marked, and a potential fourth product for Deltatown that wants alternating rates, bar code receipts, and parking end time displayed is marked. The question for the Deltatown developers is which class is best to subclass (the question marks on the inheritance relation). No matter what the choice is, either the algorithms are code duplicated in the subclasses or they are moved into protected methods in the root class which therefore becomes a pile of methods that are only relevant for a few subclasses in the inheritance hierarchy (as discussed in Chapter 11). Cohesion suffers as does analyzability.

The problem is that inheritance is a one-dimensional mechanism (in Java and C#) and therefore it must handle multi-dimensional variability by "flattening" the variability space. This leads to odd names like PayStationAlternatingRateBarcodeReceiptEndTimeDisplay. In our case, as there are 12 potential product variations I would have to implement and maintain 12 different subclasses.

Note also that the reason that the immediate subclasses of PayStation are distinguished by the choice of rate policy is purely historical! If I had to make the polymorphic design with my present knowledge of the three products, I might just as well have chosen to make the first subclasses vary by choice of output on the display: minutes or end of parking time. The design would have been just as poor, but the

point is that design is driven by the time a certain type of variation is introduced. This is in contrast to the compositional design that does not show a similar problem.

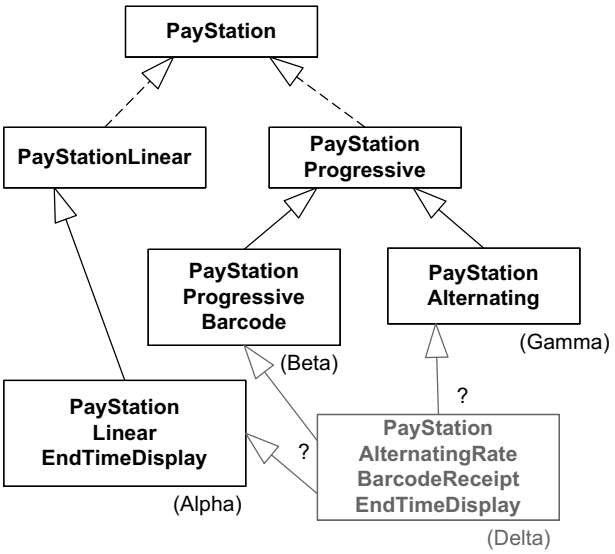

Figure 17.3: A combination of variability handled by inheritance.

Key Point: Do not use inheritance to handle multi-dimensional variation

As Java and C# only support single implementation inheritance and the inheritance mechanism therefore is one-dimensional, it cannot accommodate multi-dimensional variations without a combinatorial explosion of subclasses.

17.4 The Compositional Proposal

Taking the ③-①-② process the new requirement simply pinpoints a new behavior that may vary: the display output behavior. This insight is then the ③ step: *consider what is variable*.

The ① step, *program to an interface*, is to express the responsibility in an interface. As this is an algorithm to compute the output to display, it is the STRATEGY pattern.

Listing: chapter/multi-variance/iteration-1/src/main/java/paystation/domain/DisplayStrategy.java

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
```

```
   */
  public int calculateOutput(int minutes);
}
```

The pay station must then be refactored to use objects realizing the DisplayStrategy interface instead of doing the job itself. This is the ② step: *favor object composition*. The refactoring and test-driven development process is well known by now and will not be repeated here. You can find the resulting source code on the web site. The variability point in the pay station is of course in the readDisplay method.

Fragment: chapter/multi-variance/iteration-2/src/main/java/paystation/domain/PayStationImpl.java

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

The compositional design does not suffer a combinatorial explosion of classes. You can make all 12 variants of the pay station by configuring the set of delegate objects that the PayStationImpl should use.

Exercise 17.1: What about the factory classes? Will I not get a combinatorial explosion of these as I have to define a factory object for each product variant I can imagine? Sketch one or two solutions to this problem.

17.5 Analysis

Compositional designs handle multi-dimensional variance elegantly, because each type of variable behavior is encapsulated in its own abstraction. In contrast, both the parametric and polymorphic designs have a single abstraction that must handle all responsibilities.

The compositional pay station design adheres to the definition of object orientation as *a community of interaction objects in which each object has a role to play.* One object plays the role of rate calculator, another receipt creator, and a third display output calculator, while the pay station object's primary function is to coordinate and structure the collaboration. The pay station fulfils its **Pay Station** role by defining a protocol of interaction with these different roles. The multi-dimensional variability then becomes a question of configuring the right set of objects to play each of the defined roles. The pay station system has become a framework or product line for building pay station systems. Frameworks are discussed in detail in Chapter 32.

17.6 Selected Solutions

Discussion of Exercise 17.1:

The problem is real: if I end up in a situation where all 12 variants of the pay station system are needed then I end up with 12 factory classes: AlphaTownFactory, . . . , TwelfthTownFactory. This may seem just as bad as the 12 subclasses in the polymorphic case. However, on closer inspection the situation is less problematic. First, the

amount of code that is duplicated in the factories is much smaller, as the factory's create methods should not contain much more than a single new statement. Second, in case all 12 variants are really needed one would most probably drop the idea of individual factory classes and instead write a single implementation class that reads a configuration file stating the particular configuration and then create the proper delegates. Java's ability to dynamically load classes at run-time allows such code to be become very compact and to contain no conditional statements.

17.7 Review Questions

Describe how the pay station's different variants can be classified according to variability dimensions and as points in a multi-dimensional variability space.

Define *multi-dimensional variance.* What does *combinatorial explosion of variants* mean?

Why is it problematic to handle variability along multiple dimensions using inheritance in single implementation inheritance languages like Java and C#?

17.8 Further Exercises

Exercise 17.2:

Sketch the Java code for the PayStation class that uses parametric variability to handle all pay station variants.

Exercise 17.3:

Sketch the Java code for the PayStation subclasses for the Gammatown and Deltatown variants in Figure 17.3.

Exercise 17.4. Source code directory:
`chapter/compositional/iteration-0`

Walk in my footsteps. Implement the pay station using the compositional approach so it can handle the new Alphatown requirement.

Exercise 17.5:

Refactor your pay station software so you can change delegate objects while running, that is, an Alphatown pay station should become, say, a Betatown pay station while executing. Be careful that the pay station keeps it state (the amount of payment entered so far) during reconfiguration.