

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

Henrik Bærbak Christensen  
Version: 07-2022  
Status: Release Candidate

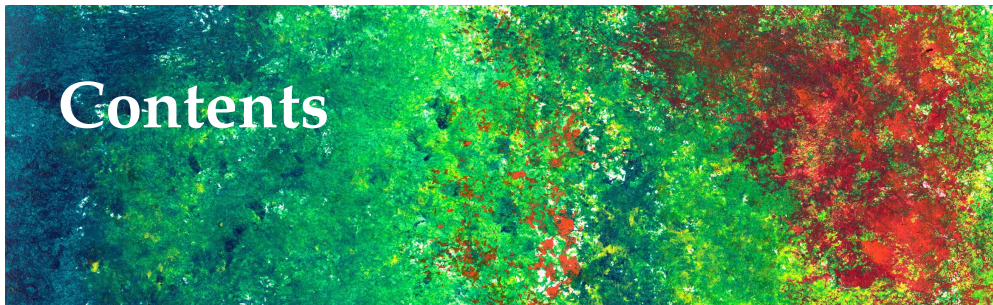
September 6, 2022



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

Reprinted and revised by permission 2020





<b>V</b>	<b>Compositional Design</b>	<b>1</b>
<b>15</b>	<b>Roles and Responsibilities</b>	<b>5</b>
15.1	What are Objects? . . . . .	5
15.2	The Language-Centric Perspective . . . . .	6
15.3	The Model-Centric Perspective . . . . .	7
15.4	The Responsibility-Centric Perspective . . . . .	8
15.5	Roles, Responsibility, and Behavior . . . . .	9
15.6	The Influence of Perspective on Design . . . . .	14
15.7	The Role–Object Relation . . . . .	16
15.8	Summary of Key Concepts . . . . .	19
15.9	Review Questions . . . . .	19
<b>16</b>	<b>Compositional Design Principles</b>	<b>21</b>
16.1	The Three Principles . . . . .	21
16.2	First Principle . . . . .	22
16.3	Second Principle . . . . .	24
16.4	Third Principle . . . . .	28
16.5	The Principles in Action . . . . .	28
16.6	Role Interfaces . . . . .	30
16.7	Interface Segregation Principle . . . . .	33
16.8	SOLID . . . . .	34
16.9	Summary of Key Concepts . . . . .	34
16.10	Selected Solutions . . . . .	35
16.11	Review Questions . . . . .	35
16.12	Further Exercises . . . . .	36

<b>17 Multi-Dimensional Variance</b>	<b>39</b>
17.1 New Requirement . . . . .	39
17.2 Multi-Dimensional Variation . . . . .	39
17.3 The Polymorphic Proposal . . . . .	41
17.4 The Compositional Proposal . . . . .	42
17.5 Analysis . . . . .	43
17.6 Selected Solutions . . . . .	43
17.7 Review Questions . . . . .	44
17.8 Further Exercises . . . . .	44
<b>18 Design Patterns – Part II</b>	<b>45</b>
18.1 Patterns as Roles . . . . .	45
18.2 Maintaining Compositional Designs . . . . .	47
18.3 Summary of Key Concepts . . . . .	51
18.4 Selected Solutions . . . . .	51
18.5 Review Questions . . . . .	52
18.6 Further Exercises . . . . .	52
<b>Bibliography</b>	<b>53</b>
<b>Index</b>	<b>55</b>
<b>Index of Sidebars/Key Points</b>	<b>57</b>

**Iteration V**

**Compositional Design**





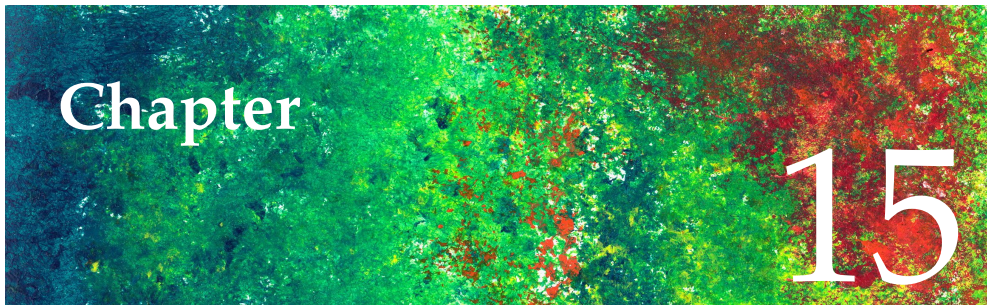
---

The primary focus of the preceding learning iterations has been on concrete requirements that I solved by analysis and by hinting at some underlying principles. In this learning iteration I will focus on defining these principles rigorously and extend our vocabulary regarding compositional design. This will allow me to explain the list of design patterns in the next part, the pattern catalogue, easily as they are simply applications and variations of the principles.

Chapter	Learning Objective
Chapter 15	<i>Roles and Responsibilities.</i> I will discuss three different perspectives on what object-oriented design and programming is. The point is not to nominate one as superior, rather they each build upon the other to form a deeper understanding of object-oriented design. Next, I will define and discuss the concepts of behavior, responsibility, roles, and protocol that are central to the responsibility centered perspective that is central in designing flexible systems.
Chapter 16	<i>Compositional Design Principles.</i> The seminal design pattern book, <i>Design Patterns: Elements of Reusable Design</i> , described a small set of principles that show up on almost all design patterns. In this chapter, I describe the principles and show how I have already applied them in the form of the ③-①-② process.
Chapter 17	<i>Multi-Dimensional Variance.</i> Returning to the pay station and a new requirement, I will define the combinatorial explosion problem that is handled elegantly by compositional designs but poorly by polymorphic and parametric designs.
Chapter 18	<i>Design Patterns – Part II.</i> Armed with our knowledge of compositional design and its terminology I will review the design pattern concept. I will also look at how a good working knowledge of design patterns provides us with a roadmap that lessens the problem of over-viewing large compositional designs.

---





# Roles and Responsibilities

## Learning Objectives

Emphasis has primarily been put on the practical aspects of design patterns up until now. In this chapter, I will change to a more analytical and theoretical view. The learning objective of this chapter is to introduce three different perspectives on object-oriented programming, starting by asking the fundamental question: *What is an object?* Your answer to this question of course influences how you design object-oriented software in its most fundamental way. Knowing that there are indeed different perspectives and understanding the strengths of each will make you a better designer and software architect by extending the set of tools you have available to tackle design challenges.

One of the perspectives, the *role* perspective, is the central key to understand flexible software designs such as design patterns and frameworks so this chapter's focus is also on defining the central concepts within this perspective.

## 15.1 What are Objects?

Some time ago, I made a small study of how the concept "object" was defined and used in a number of object-oriented teaching books (Christensen 2005). It turned out that the definitions were broadly classified in three groups that I denote *language*, *model*, and *responsibility*-centric perspectives. The perspective taken has profound impact on the way systems are designed, the quality of the designs, and the type of problems that designs can successfully handle. One can see each perspective as a refinement of the preceding, so it is not a question of voting one as the champion, they all have their applicability. This said, the responsibility-centric perspective is the most advanced one, and it is important in order to deeply understand flexible and configurable designs. Over the next few sections, I will present and discuss each perspective.

## 15.2 The Language-Centric Perspective

The focus of the language-centric perspective is classes and objects as concrete building blocks for building software. A typical definition is:

An object is a program construction that has data (that is, information) associated with it and that can perform certain actions. (Savitch 2001, p. 17)

The emphasis is thus on the program language structure: fields and methods. This is inherently a *compile-time* or *static* view—the definition speaks in terms of what we see in our editor. The classic example is the `Account` class that you can find in almost any introductory textbook on objects.

```
public class Account {
    int balance;
    public void withdraw(int amount) {
        balance -= amount;
    }
    public int balance() { return balance; }
    ...
}
```

The implicit consequence of this view is that an object is an entity in its own right—the definition is closed and any object can be understood in isolation—you simply enumerate the fields and methods and you are done. The advantage of this perspective is of course that it is very concrete, closely related to the programming language level, and therefore relatively easy to understand as a novice. Also, the perspective is fundamental: you cannot develop or design object-oriented software if you do not master this perspective.

The disadvantage is, however, that it remains relatively silent about how to structure the collaboration and interplay between objects, the dynamics, and this is typically where the hard challenges of design lie. As an example, an account has an owner who has a name. Taking the perspectives “class = fields + methods” literally the design below is fine:

```
public class Account {
    int balance;
    String ownerName; Date ownerBirthDate;
    public String getOwner() {
        return ownerName;
    }
    ...
    public void withdraw(int amount) {
        balance -= amount;
    }
    public int balance() { return balance; }
    ...
}
```

This design has several limitations, but the perspective itself offers little guidance in the process of designing any realistic system. It is the perspective that was most prominent in my survey of text books.

☞ Discuss the limitations of the design above.

Copyrighted Material. Do Not Distribute.

## 15.3 The Model-Centric Perspective

The model-centric perspective perceives objects as parts in a wider context namely a *model*. A model is a simplified representation or simulation of a part of the world. The inspiration for the perspective was toy-models, simulations, and scientific models. For instance a remote-controlled model car has parts that are similar to a real car while others are abstracted away; a toy railway also displays many of the features of real-world railway systems while other features are missing.

A typical definition is:

Java objects model objects from a problem domain. (Barnes and Kolling 2005, p. 3)

The model view has roots tracing back to the Simula tradition of simulation, Scandinavian object-orientation (Madsen et al. 1993), and the Alan Kay notion of *computation as simulation* (Kay 1977). Here the program execution is a simulation of some part of the world, and objects are perceived as the model's parts. A definition of object-orientation from the Scandinavian school is (Madsen et al. 1993, §18):

### Definition: Object-orientation (Model)

A program execution is regarded as a physical model simulating the behavior of either a real or imaginary part of the world.

The key process for designing software, denoted “modeling”, is the process of abstracting and simplifying (real-world) phenomena and concepts and representing them by objects and classes; and modeling (real-world) relations like association and aggregation by relations between objects.

Going back to the account case, this perspective would look at the world of banking and note the concepts “Account” and “Owner” as well as the relations “has-an-account” and “owned-by”. As an account can have several owners, and a single person can have several accounts, this perspective would reject the design in the previous section. The modeling process would conclude that the real world concepts of account and owner should be reflected in the design leading to classes **Account** and **Owner** that are associated by a many-to-many relation, as shown in Figure 15.1.



Figure 15.1: Modeling the Account Owner system.

This perspective stresses objects as entities in a larger context (they are parts of a model) as opposed to the self-contained language centric definition. This naturally leads to a strong focus on what the relations are between the elements of the model: association, generalization, composition.

Dynamics is an inherent part of the concept simulation and the explicit guideline for designing object interaction is to mimic real world interactions. Thus, the real-world

scenario *the owner withdraws money from his account* tells the designer that the objects must have methods to mimic this behavior. The question is should the withdraw behavior belong to `Account` or `Person`? The encapsulation principle tells us to put it in the account class because it encapsulates the balance. However, this obvious decision that designers do all the time is actually not very faithful to the “real world”: real accounts do not have behavior. Before computers did the hard work, bank accounts were simply inanimate records kept by the bank—a clerk did the paperwork to withdraw from the account, not the account itself. Thus, object-oriented objects are animistic mirrors of their real world role models, and a program execution resembles a cartoon full of otherwise inanimate objects springing to life, sending messages to each other.

☞ Find examples in programming books or your own programs of classes that model inanimate concepts but exhibit behavior.

This model has limitations as well, because it is relatively silent about those aspects of a systems that has no real world counterpart. Generally object-oriented design books like examples like the account system because the classes mirror real world concepts. But where do I get inspiration for designing a graphical user interface? For keeping multiple windows synchronized? For accessing a database?

## 15.4 The Responsibility-Centric Perspective

In the responsibility-centric perspective a program’s *dynamics* is the primary focus and thus the object’s behavior and its responsibilities are central.

One definition is the following:

The best way to think about what an object is, is to think of it as something with responsibilities. (Shalloway and Trott 2004, p. 16)

This perspective can be traced back to the focus on responsibilities in Ward Cunningham’s work on the CRC cards (Beck and Cunningham 1989) and the work by Wirfs-Brock on responsibility driven design (Wirfs-Brock and McKean 2003).

Both the language and model-centric perspective tend to focus on static aspects: objects, classes, and relationships. However, what makes a program interesting and relevant is the *behavior* that these parts have, individually and jointly. We can also state this more pragmatically: The customers of our software are quite indifferent about classes and relations; they care about the software’s *functionality*—what does it do to make my work easier, more fun, more efficient? Thus, software behavior is the most important aspect, as it pays the bills! Wirfs-Brock and McKean express this:

*Success lies in how clearly we invent a software reality that satisfies our application’s requirements—and not in how closely it resembles the real world.*

**Copyrighted Material. Do Not Distribute.**

Responsibility-centric thinking states that the two fundamental concepts, object and relation, have to be supplemented by a third equally fundamental concept: *role*. The role concept helps us to break the rigid ties between object and functionality. This leads to another definition of object-orientation and Budd (2002) has made the following:

### Definition: Object-orientation (Responsibility)

An object-oriented program is structured as a community of interacting agents called objects. Each object has a role to play. Each object provides a service or performs an action that is used by other members of the community.

This statement emphasizes that the overall behavior and the functionality of a program is defined by lots of individual objects working together.

In this view, an executing program is perceived as a community. Human communities get work done by organizing a lot of individuals, defining roles and responsibilities for them and state the rules of collaboration between the different roles. The overall behavior of a community is the sum of many individual but concerted tasks being accomplished. One thing that Budd's definition is not very specific about, however, is that often a single actor performs multiple roles in a community depending on the specific context he or she is involved in. I will return to this point later.

Going back to the pay station case, the original design (see Chapter 4) was the result of a model perspective, looking at a real pay station that issues receipts. However, the analysis in the STRATEGY chapter ended up by deviating from the "real world model" perspective. Looking at the pay station down on the parking lot, I see the machine and the receipts, but no "rate calculator". The `RateStrategy` does not model anything real—it is purely a role with a single responsibility which I require *some* object to be able to play once the software starts executing.

## 15.5 Roles, Responsibility, and Behavior

I will elaborate the responsibility-centric perspective by looking at the central concepts that allow us to understand and design the functional and dynamic aspects of software systems. I start from the most basic level, behavior, and work my way up in abstraction level.

### 15.5.1 Behavior

Abstractly, behavior may be defined as:

#### Definition: Behavior

Acting in a particular and observable way.

That is, *doing something*. As an everyday example, I take the bag of garbage out from under the kitchen sink, tie a knot on the bag, and walk to the garbage can and throw it in. I acted in a particular and observable way.

In object-oriented languages behavior is defined by objects having methods that are executed at runtime. Methods are templates for behavior, algorithms, in the sense that parameters and the state of the object itself and associated objects influence the particular and observable acting.

Collective behavior arises when objects interact by invoking methods on each other (sometimes referred to as message passing). Message passing occurs when one object requests the behavior of one of its associated objects.

Method names and parameter lists, however, convey little information about the actual behavior. Often an object's behavior is the result of a concerted effort from a number of collaborating objects. For instance, when a flight reservation object is requested to print out, it will request the associated person object to return the person's name. UML sequence diagrams are well suited for describing object interaction.

Behavior is the “nuts-and-bolts” of a program execution in the sense that what actually gets done at runtime is the sum of the behavior defined by the methods that have been invoked. However, the concept of behavior is too low-level to be really useful when designing systems—we need a more abstract concept.

## 15.5.2 Responsibility

### Definition: Responsibility

The state of being accountable and dependable to answer a request.

Responsibility is intimately related to behavior but it is at a more abstract level. Going back to the garbage example, I am *responsible* for getting the garbage from under the kitchen sink to the garbage can; but no particular behavior is dictated. I may decide to just throw the bag on the door step and wait to bring it to the can until next time I have to go there anyway—or more likely I yell at one of my sons that they must do it. How they do it is also not relevant—probably my youngest son will run to the can while fighting imaginary monsters. The point is that many different observable behaviors are allowed as long as the “contract” of the responsibility is kept: that the garbage ends in the can.

Responsibility is a more abstract way of organizing and describing object behavior and thus much better suited to the abstraction level needed for software design. It is a way to maintain the overview and avoid getting overwhelmed by algorithm details.

Responsibility is often best communicated in a design/implementation team in short and broad statements. For instance, in the PlayStation interface header, the responsibilities are stated as:

Responsibilities:

- 1) Accept payment;
- 2) Calculate parking time based on payment;
- 3) Know earning, parking time bought;
- 4) Issue receipts;
- 5) Handle buy and cancel events.

**Copyrighted Material. Do Not Distribute.**



Responsibilities should not be stated too programming specific (like “have an `addPayment` method taking an integer parameter `amount`”) nor too broad and vague (like “behave like a pay station”).

The technique of stating responsibilities was elaborated in the **CRC Card** technique, where classes are described at the design level using three properties: **Class name**, **Responsibilities**, and **Collaborators**. A CRC card for the pay station would look like below: The **Class name** is shown on top, the **Responsibilities** in the left pane below and **Collaborating classes** in the right pane.

<b>PayStation</b>	
Accept payment	PayStationHardware
Calculate parking time based on payment	Receipt
Know earning, parking time bought	
Issue receipts	
Handle buy and cancel events	

Behavior is defined by methods on objects at the programming language level—an object behaves in a certain way when a method is called upon it. What about responsibilities?

The language construct that comes closest is the interface. An interface is a description of a set of (related) method signatures but no method body (i.e. implementation) is allowed. The `PayStation` interface contains the method declarations that encode the responsibilities mentioned on the CRC card. Thus, an interface is much more specific than the above high level description, however an interface only specifies an *obligation* of implementing objects to exhibit behavior that conforms to the method signatures, not any specific algorithm to use. We say “conforms to the method signatures” but in practical programming an object that implements a particular method in an interface must also conform to the underlying contract—that “it does what it is supposed to do” according to the method documentation. The signature of method `buy` only tells me that a `Receipt` instance is returned, but the contract of course is that its value must match the parking time matching the entered amount.

The reason I argue that the language construct of interface is better than classes for expressing responsibility is that the developer is explicitly forced *not* to describe any algorithm—and thus keep focused on the contract instead of getting hooked on details too early. Objects are free to implement the methods in any way as long as the behavior adheres to the specification.

In the pay station, there are five responsibilities and four methods in the interface. Some responsibilities, like *know parking time bought* corresponds quite closely to a method, `readDisplay`, whereas others, like *calculate parking time*, have no associated method. This is common. The main point is that the interface exposes the proper set of methods for the responsibilities to be carried out in the context of the collaborating objects. For instance, the *calculate parking time* responsibility is served as the pay station hardware (or our JUnit testing code) invokes `addPayment` and next `readDisplay`.

### 15.5.3 Role

Responsibilities are only interesting in the context of collaboration: If no-one ever wants to insert coins in the pay station, there is no need for a “accept payment” responsibility. Thus, what makes the responsibility interesting and viable is its inherent obligation to someone else. In an executing program objects are collaborating, each object having its specific responsibilities. To collaborate properly they must agree upon their mutual responsibilities, the way to collaborate, and the order in which actions must be taken.

Just as responsibility is more abstract than behavior, we need a term for expressing mutual responsibilities and collaboration patterns.

#### Definition: Role (General)

A function or part performed especially in a particular operation or process.

This definition embodies the dual requirement: both “function performed” (responsibilities) as well as “in a particular process” (collaboration pattern/protocol). The role concept allows us to express a set of responsibilities and a specification of a collaboration pattern without tying it to a particular person or particular object.

The role concept is central for understanding any community or human society. Roles define how we interact and understand what is going on. At the university I play the role of **lecturer** a couple times a week, and around one hundred people play the role of **student**. We know the responsibilities of each other and therefore what to expect. It is my responsibility to talk and hopefully tell something interesting related to the course material; it is the responsibility of the students to stop talking when the lecture begins—and try hard not to fall asleep. Similarly, if I go to the hospital I will probably not know the individuals working there but I know the roles of **physician**, **nurse**, and **patient**, and thereby each individual knows and understands what to expect of each other. A community has severe problems functioning if people do not know the roles.

The relation between role on one hand and person on the other is a many-to-many relation. Taking myself I go in and out of roles many times a day: father, husband, teacher, supervisor, researcher, textbook author, etc. A single person can and usually does play many different roles although not at the same time. From the opposite perspective, a role can be played by many different persons. If I become sick, someone else will take on my teacher role. At the hospital, the person playing the ward nurse role changes at every shift. The same many-to-many relation exists between software design roles and objects. I will explore this in detail at the end of the chapter.

#### Key Point: The relation between role and object is a many-to-many relation

*A role can be played by many different types of objects. An object can perform many different roles within a system.*

Sometimes roles are invented to make an organization work better. As a simple example a pre-school kindergarten had the problem that the teachers were constantly

interrupted in their activities with the children to answer the phone, deliver messages from parents, fetch meals, etc. They responded by defining a new role, the **flyer**, whose responsibility it was to answer all phone calls, bring all the meals, etc. Thus all other teachers were relieved of these tasks and allowed to pay attention to the children without interruptions. The teachers then made a schedule taking turns on having the role as flyer.

## 15.5.4 Protocol

Roles express mutual expectations. Students expect the lecturer to raise his voice and start presenting material. It will not work if he instead just sits down on a seat at the back row and keeps silent. Likewise the lecturer expects the students to keep silent while lecturing. A hospital will not work if all nurses jump into the hospital beds and start asking the patients for aspirin. Thus roles rely on more or less well-defined protocols.

### Definition: Protocol

A convention detailing the sequence of interactions or actions expected by a set of roles.

Remember that an old interpretation of protocol is indeed “diplomatic etiquette”, that is, the accepted way to do things. The student–lecturer protocol is clear though unspoken: “Lecturer starts talking, students fall asleep.”

Software design roles depend even more heavily on understanding the protocols. Humans cope with minor defects in the protocol, but software is not that forgiving. The protocol between the pay station and the rate strategy dictates that the pay station is the active part and the rate strategy the reactive: the pay station requests a calculation and the rate strategy responds with an answer. At a more abstract level, this is the protocol of the STRATEGY pattern: **Context** initiates the execution of an algorithm, and a **ConcreteStrategy** reactively responds when requested. Of course, this is a very simple protocol, but some patterns in learning iteration 66, *A Design Pattern Catalogue*, have elaborate protocols that are the core of the pattern. For instance, OBSERVER’s protocol requires objects to register before they can expect notifications using yet another protocol.

Protocols are extremely important to understand the dynamic, run-time, aspects of software. UML sequence diagrams are well suited to show protocols, both at the individual object level but more important also at the role level: instead of showing an object’s timeline, you draw method invocations between roles. I will use sequence diagrams to show design pattern protocols for the complex patterns.

## 15.5.5 Roles at the Design Level

Another, more software oriented, definition of role is:

### Definition: Role (Software)

A set of responsibilities and associated protocols.

Copyrighted Material. Do Not Distribute.

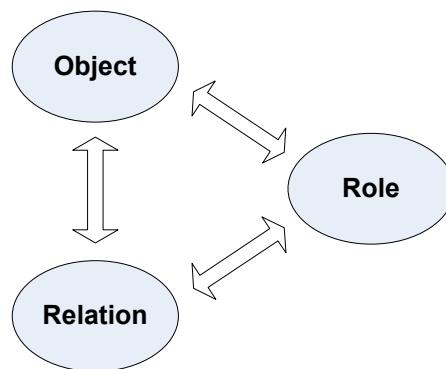


Figure 15.2: The three fundamental concepts in design.

Wirfs-Brock et al. simply defines role as “set of responsibilities” but I find the protocol aspect important as roles interact to fulfill complex responsibilities.

Roles do not have any direct counterpart in main-stream programming languages. The programming language construct that comes closest is again the “interface” that supports aspects of the concept of a role. However, main-stream languages have no way of forcing specific protocols, that is no way of enforcing that method A on object X is invoked before method B on object Y, etc. This is left for the developers to adhere to—and a constant source of software defects.

Throughout this book I have used **role description boxes** or just **role boxes** to define roles, like the pay station role:

#### PayStation

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

The diagram is a simple adaptation of the CRC card: instead of class name at the top, I write the name of the role, and I have removed the collaborators pane. I find that the collaborators are more easily read from the UML class diagrams.

## 15.6 The Influence of Perspective on Design

Returning to the three perspectives, language, model, and responsibility-centric, these have strong impact upon our design. As indicated, the language-centric perspective is important at the technical programming level, but has little to say about design, so I will concentrate on the latter two.

In the model-centric perspective, the focus is modeling a part of the (real or imaginary) world which is then translated into the elements of our programming language: classes and objects. This leads naturally to a strong focus on structure first, and behavior next. That is, I first create a landscape of relevant abstractions and next assign

responsibility and behavior to these abstractions. To paraphrase it, I first draw a UML diagram of my classes, and next try to find the most appropriate class to assign the responsibilities, the most appropriate place to put the methods in. Budd calls this the **who/what** cycle: first find *who*, next decide *what* they must do. In this perspective, you do not really need the role concept; objects and their relations are sufficient: the objects are already in place when you get to the point of assigning behavior.

The responsibility-centric perspective, in contrast, focuses on functionality but at the higher level of abstraction of responsibilities and roles. Here the design process first considers the tasks that have to be carried out and then tries to group these into natural and cohesive roles that collaborate using sensible protocols. Next, objects are assigned to play these roles. That is, the landscape of behavior and responsibilities is laid out first, and next objects are invented and assigned to fulfill the responsibilities. Budd calls this the **what/who** cycle: find out *what* to do, next decide *who* does it. There, the role concept is vital as the placeholder of responsibilities until it can be assigned.

Shalloway and Trott (2004) present a nice story to illustrate the different perceptions of these perspectives. They tell that they have an umbrella that shields them from the rain—very convenient as they live in Seattle that gets its fair share of rain. Actually it is a very comfortable umbrella, because it can play stereo music, and in case they want to go somewhere, it can drive them there—of course without getting wet. It is all quite mystical until the punch line is revealed: their “umbrella” is a car.

In the model perspective, this story is absurd. In this perspective, an umbrella is a set of stretchers and ribs covered by cloth. A car is not. Thus a car is *not* an umbrella. The Java declaration

```
public class Car extends Umbrella { ... }
```

does not make sense.

However, in the responsibility perspective, it *does* make sense, because it is not the object umbrella but the responsibility of an umbrella that is the focus. In this perspective, an umbrella is a canopy designed to protect against rain. The focus is on what it “does” not what it “is”.

```
public class Car implements UmbrellaRole { ... }
```

Thus, a car does play the umbrella role when Mr. Shalloway or Mr. Trott drive around Seattle: it is responsible for protecting them from rain.

Sometimes, the two perspectives arrive at the same design. As an example, consider a temperature alarm system design to warn if the temperature of, say, a refrigerator comes above a threshold temperature. The system consists of a sensor in the refrigerator and a monitor station responsible for periodically measuring the temperature measured by the sensor and alarm if the threshold is reached. The model perspective *who/what* would identify two phenomena: the sensor and the monitor that are associated. Next the tasks are assigned: the monitor must periodically request temperature readings from the sensor; the sensor must measure and return temperature upon request. The UML class diagram ends up like in Figure 15.3.

In the *what/who* cycle, the designer would identify the responsibilities *to monitor temperature*, *to alarm if temperature threshold is exceeded* and *to measure temperatures*. “To measure” expresses an algorithm that depends upon the specific manufacturer of

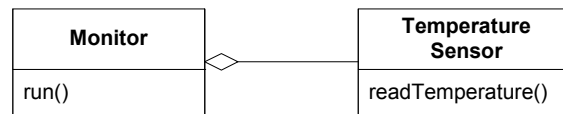


Figure 15.3: Model perspective design.

the sensor, so it points to a STRATEGY pattern. As it has two roles it seems natural to group the first two responsibilities in a **monitor** role (**context** role in STRATEGY) and the last responsibility in a **temperature measure strategy** role (**strategy** role in STRATEGY). Thus the designs are almost identical except for the naming that focuses on either hardware objects or functionality.



Figure 15.4: Role perspective design.

## 15.7 The Role–Object Relation

As argued above the introduction of roles as a fundamental design concept for object oriented design loosens the coupling between functionality and object. I will give some examples in this section.

### 15.7.1 One Role – Many Objects

The one-to-many relation between interface and objects manifests itself in that many different objects may implement a particular interface. This means that objects with otherwise rather different behavior may be used in a particular context as long as they adhere to the role this context expects.

A good example is from the Java library in which the only thing the sorting algorithm in the Collections class expects is that all objects in a given list implements the `java.util.Comparable` interface. Thus if we make a class representing apples then we can sort apple objects simply by implementing this interface:

```

public class Apple implements Comparable {
    private int size;
    [other Apple implementation]
    public int compareTo(Object o) {
        [apple comparison algorithm]
    }
}
  
```

Thus, in the context of the sorting algorithm it is irrelevant what the objects are (here apples), the only interesting aspect is that the objects can play the **Comparable** role. And the **Comparable** role simply specifies that objects must have the responsibility

to tell whether it is greater than, equal to, or less than some given object; and the protocol is quite simple: it must return this value upon request.

In another application the sorting algorithm is reused as, say, `Orange` objects can also implement the `Comparable` interface.

## 15.7.2 Many Roles – One Object

The many-to-one relation between interface and object is possible in Java and C# as these languages support multiple interface inheritance. That is, a class may implement multiple interfaces. To rephrase this, we can assign several roles to a single object.

You may wonder if an object playing several roles may easily end as a *blob* class. It may be so, but it does not happen if you design your roles small and cohesive. This is best illustrated by an example. The example is from `MiniDraw`, the two dimensional graphics framework that I will present in learning iteration 7, *Frameworks*. A central role is the “drawing” that contains the set of figures to show: think of an UML class diagram editor in which you may add, move, and remove graphical objects like class rectangles, association lines, etc. The drawing also allows a set of figures to be selected and then, for instance, be moved as a group. Thus, the responsibilities of the drawing role<sup>1</sup> are

### Drawing

- Be a collection of figures.
- Allow figures to be added and removed.
- Maintain a temporary, possibly empty, subset of all figures, called a *selection*.

If you look at the first two responsibilities it is basically the responsibilities of a *collection of figures*. And the last responsibility is the ability to handle a subset of figures, and as such independent of the two first responsibilities. Thus I can define more fine-grained roles.

### FigureCollection

- Be a collection of figures.
- Allow figures to be added and removed.

### SelectionHandler


- Maintain a selection of figures.
- Allow figures to be added or removed from the selection.
- Allow a figure to be toggled in/out of the selection.
- Clear a selection.

Thus the `Drawing` interface is actually defined as

<sup>1</sup>The actual role has a few more responsibilities but for the sake of the example they are not considered here.

```
public interface Drawing extends FigureCollection , SelectionHandler {
    ...
}
```

A key point is that the `SelectionHandler` is a self-contained role and I can therefore make default implementations of both roles in `MiniDraw`: `StandardFigureCollection` and `StandardSelectionHandler`. Thus the default `Drawing` implementation, `CompositionalDrawing`, simply creates instances of both these implementations and delegates all figure collection and selection handling to these two delegates. It also means I can reuse these default collection and selection handling implementations in specialized drawing implementations for the projects later in the book.

 This is actually an example where the focus on roles leads to inventing objects, the standard selection handler, that would probably not have been identified by a model-perspective.

Another common usage is to integrate two frameworks. An example of this is again `MiniDraw`, that needs to integrate with a concrete Java GUI framework. `MiniDraw` solves this problem by defining classes that simply play roles in both the `MiniDraw` framework as well as the `Swing` GUI framework. An example is the drawing window role that is defined by the `DrawingView` interface in `MiniDraw`. To integrate it with `Swing`, `MiniDraw` defines the `StandardDrawingView` class:

```
public class StandardDrawingView
    extends JPanel
    implements DrawingView ,
               MouseListener ,
               MouseMotionListener ,
               KeyListener {
```

This way the concrete `Swing` drawing canvas `JPanel` and the object playing the `MiniDraw` canvas role, defined by the `DrawingView` interface, are directly coupled within the `StandardDrawingView` object. As it also plays the roles of mouse and mouse motion listener it can receive all types of mouse events.

### 15.7.3 Analysis

Both of the discussions above actually pull in the same direction: towards defining small and cohesive roles that can be expressed as interfaces. As a role is not one-to-one related to an object I still have the freedom to implement the roles in a number of different ways. And smaller roles increase the likelihood that objects implementing them can be reused.

#### Key Point: Define small and cohesive roles

*Roles should not cover too many responsibilities but stay small and cohesive. Complex roles may then be defined in terms of simpler ones.*

This keypoint is similar to the *interface segregation principle*<sup>16.7</sup> discussed in the next chapter.



## 15.8 Summary of Key Concepts

Object-oriented design can be seen from different perspectives. In the *language-centric* perspective, objects are simply containers of data and methods. The *model-centric* perspective perceives objects as model elements, mirroring real world (or imaginary) phenomena. The *responsibility-centric* perspective views objects as interacting agents playing a role in an object community.

The responsibility-centric perspective puts emphasis on the behavior of software systems and the concepts used to do design are *behavior, responsibilities, roles, and protocol*. Behavior is the lowest, concrete, level of actual acting. Responsibility is an abstraction, being dependable to answer a request, but without committing to a specific set of behaviors. Role is a set of responsibilities with associated protocol. And protocol is the convention detailing the interaction expected by a set of roles.

The role concept is important because it allows designers to design system functionality in terms of responsibilities and roles and only later assign roles to objects. This *what/who* process is the opposite of the traditional modeling approach that focus on *who/what*, i.e. finding the objects first and next assigning behavior to these. The result is smaller and more cohesive objects but more complex protocols. It often also results in objects that have no real world concept counterpart.

The relation between role and object is a many-to-many relation. An object may implement several roles due to the multiple interface inheritance ability of Java and C#. And a role may be played by many different objects, as multiple concrete classes may implement the same interface. Therefore, it is advisable to define small and cohesive roles and express them as interfaces.

## 15.9 Review Questions

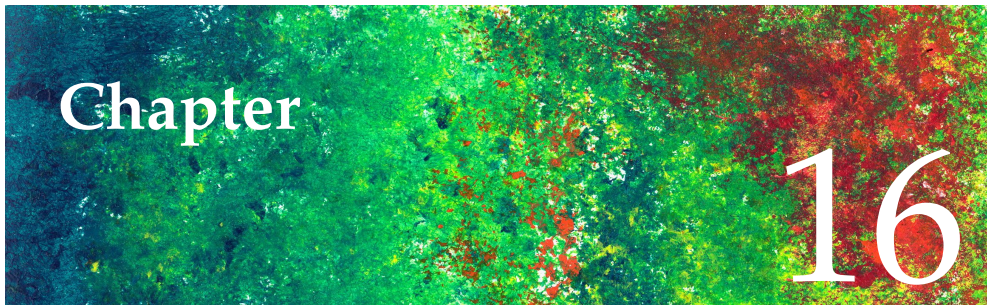
List the three different perspectives on what an object is, and outline the main ideas of each.

Define the concepts *behavior, responsibility, role, and protocol*. What do they mean? Provide examples of each.

Explain the difference between the *who/what* and *what/who* approach to design.

Give examples of a single role that is played by many types of objects. Give examples of a single object that plays several roles.





# Compositional Design Principles

## Learning Objectives

In this chapter, I will more formally introduce the three principles that form the ③-①-② process. The learning focus is understanding how these principles manifest themselves in our design and in our concrete implementation, and how they work in favor of increasing the maintainability and flexibility qualities in our software.

## 16.1 The Three Principles

The original design pattern book (Gamma et al. 1995) is organized as a catalogue of 23 design patterns. It provides, however, also an introduction that discusses various aspects of writing reusable and flexible software. In this introduction they state some principles for reusable object-oriented design.

### Compositional Design Principles:

- ① *Program to an interface, not an implementation.*
- ② *Favor object composition over class inheritance.*
- ③ *Consider what should be variable in your design.*  
(or: *Encapsulate the behavior that varies.*)

The authors themselves state the first two directly as principles (p. 18 and 20) whereas the third principle is mentioned as part of the process of selecting a design pattern (p. 29) and thus not given the status of a principle. Shalloway and Trott (2004) later highlight the statement *Consider what should be variable* as a third principle. Grand (1998) restated the two first principles as *fundamental patterns*, named INTERFACE and DELEGATION.

These three principles work together nicely and form the mindset that defines the structure and responsibility distribution that is at the core of most design patterns. You have already seen these three principles in action many times. They are the ③-①-② process that I have been using over and over again in many of the preceding chapters.

- |   |   |   |
|---|---|---|
| ③ I identified some behavior that was likely to change...   | = | ③ <i>Consider what should be variable in your design.</i> |
| ① I stated a well-defined responsibility that covers this behavior and expressed it in an interface...  | = | ① <i>Program to an interface, not an implementation.</i>  |
| ② Instead of implementing the behavior ourselves I delegated to an object implementing the interface... | = | ② <i>Favor object composition over class inheritance.</i> |

## 16.2 First Principle

① *Program to an interface, not an implementation.*



A central idea in modern software design is *abstraction*. There are aspects that you want to consider and a lot of details that you do not wish to be bothered with: *details are abstracted away*. We humans have limited memory capabilities and we must thus carefully select just those aspects that are relevant and not overburden ourselves with those that are not.

In object-oriented languages, the *class* is a central building block that encapsulates a lot of irrelevant implementation details while only exposing a small set of public methods. The methods abstract away the implementation details like data structures and algorithms.

Encapsulation and abstraction lead to a powerful mindset for programming: that of a *contract* between a *user* of functionality and a *provider* of functionality. I will use the term *client* for the user class which is common for descriptions of design patterns—do not confuse it with the “client” term used in internet and distributed computing. The providing class I will term the *service*. Thus there exists a contract between the client (“I agree to call your method with the proper parameters as you have specified...”) and the server (“if you agree to provide the behavior you have guaranteed”).

The question is then whether the class construct is the best way to define a contract between a server and its client. In many cases, the answer is “no!” It is better to *program to an interface* for the following reasons:

**Copyrighted Material. Do Not Distribute.**

**Clients are free to use *any* service provider class.** If I couple the client to concrete or abstract classes I have severely delimited the set of objects that can be used by the client: an object used by the client *has* to be a subclass! Thus the client has high coupling to a specific class hierarchy.

Consider the two situations a) and b) in Figure 16.1 that reflect a development over time. In situation a) the developers have applied the *program to an interface* principle

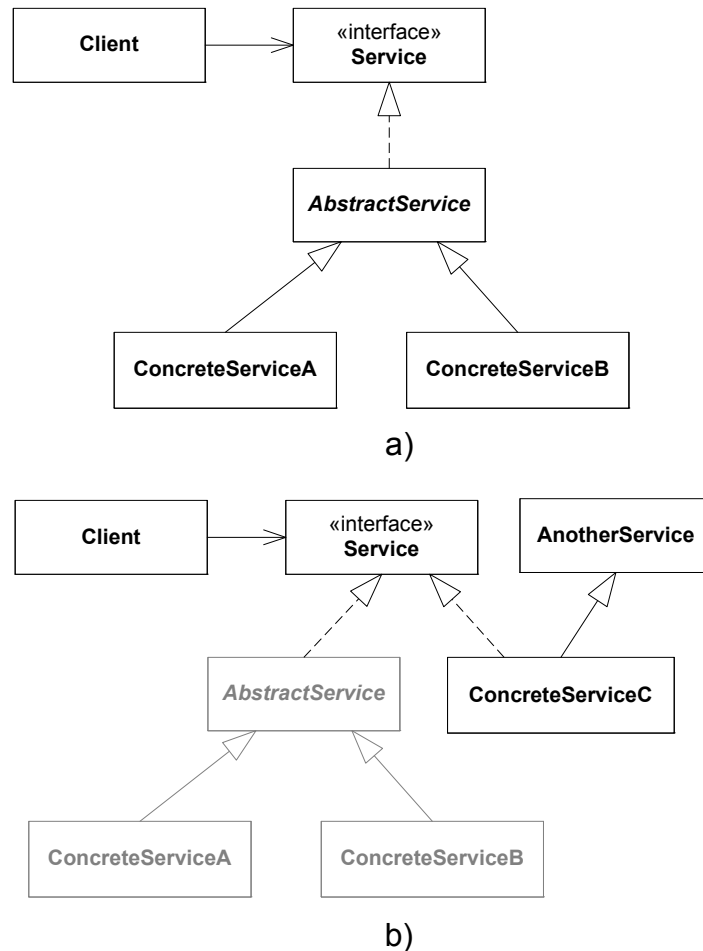


Figure 16.1: a) Original design b) Later design.

and provided a hierarchy of classes that the client uses. Later, however, it becomes beneficial to let the client collaborate with an instance of `AnotherService`. The interface decouples the client from the concrete implementation hierarchy.

**Exercise 16.1:** Describe what would happen if in situation a) the client was directly coupled to the `AbstractService` and you had to make the `ConcreteServiceC`. How would you do that?

**Interfaces allow more fine-grained behavioral abstractions.** By nature, a class must address all aspects of the concept that it is intended to represent. Interfaces,

however, are not coupled to concepts but only to behavioral aspects. Thus they can express much more fine-grained aspects. The classic example is the `Comparable` interface, that only expresses the ability of an object to compare itself to another. Another example is the `FigureCollection` and `SelectionHandler` interface, explained in Section 15.7.2, that showed how introducing fine-grained abstractions can lead to reuse.

Often interfaces are used as **Private Interfaces** (Newkirk 1997) or **Role Interfaces** (Fowler 2006), that is, fine-grained and specialized “mini-roles” that allows specialized access or mutation of an object, only available to specific collaborators. I will discuss this concept in more detail later in Section 16.6

**Interfaces better express roles.** The above discussion can be rephrased in terms of the role concept. Better designs result when you think of the *role* a given server object will play for a client object. This mind set leads to making interfaces more focused and thus smaller.

Again, consider the `Comparable` interface in the java collection library. To the sorting algorithms, the only interesting aspect is that the objects can play the comparable role—just as a Hamlet play is only interested in a person’s ability to play Hamlet. All other aspects are irrelevant.

**Classes define implementation as well as interface.** Imagine that the client does not program to an interface, but to a concrete service class. As a service class defines implementation, there is a risk that the client class will become coupled to its concrete behavior. The obvious example is accidentally accessing public instance variables which creates high coupling. A more subtle coupling may appear if the service implementation actually deviates from the intended contract as stated by class and method comments. Some examples are for methods to have undocumented side effects, or even have defects. In that case the client code may drift into assuming the side effects, or be coded to circumvent the defective behavior. Now the coupling has become tight between the two as another service implementation cannot be substituted.

## 16.3 Second Principle

② *Favor object composition over class inheritance.*

This statement deals with the two fundamental ways of reusing behavior in object-oriented software as outlined in Figure 16.2.

*Class inheritance* is the mechanism whereby I can take an existing class that has some desirable behavior and subclass it to change and add behavior. That is, I get complex behavior by reusing the behavior of the superclass. *Object composition* is, in contrast, the mechanism whereby I achieve complex behavior by *composing* the behavior of a set of objects.

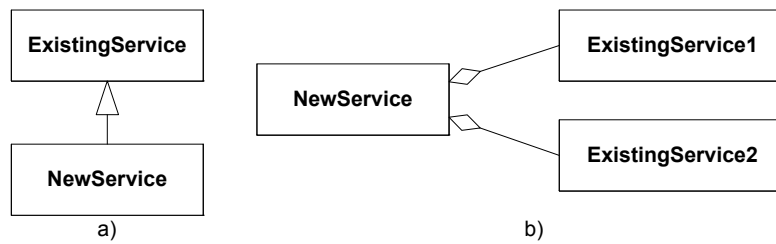


Figure 16.2: Class inheritance (a) and object composition (b).

You saw these two techniques discussed in depth in Chapter 7, *Deriving Strategy Pattern*, and in many of the following chapters concerning design patterns. The polymorphic proposal suggested using class inheritance to introduce a new rate structure algorithm; the compositional proposal suggested using object compositions to do the same.

Class inheritance has a number of advantages. It is straightforward and supported directly by the programming language. The language support ensures that you write very little extra code in order to reuse the behavior of the superclass. Below, I will describe liabilities of class inheritance, followed by a few for object composition.

**Encapsulation.** It is a fact that “inheritance breaks encapsulation” (Snyder 1986). A subclass has access to instance variables, data structures, and methods in all classes up the superclass chain (unless declared `private`.) Thus superclass(es) expose implementation details to be exploited in the subclass: the coupling is high indeed. This has the consequence that implementation changes in a superclass are costly as all subclasses have to be inspected, potentially refactored, and tested to avoid defects.

Object composition, in contrast, depends upon objects interacting via their interfaces and encapsulation is ensured: objects collaborating via the interfaces do not depend on instance variables and implementation details. The coupling is lower and each abstraction may be modified without affecting the others (unless the contract/interface is changed of course).

**You can only add responsibilities, not remove them.** Inheriting from a superclass means “you buy the full package.” You get *all* methods and *all* data structures when you subclass, even if they are unusable or directly in conflict with the responsibilities defined by the subclass. You may override a method to do nothing in order to remove its behavior, or indicate that it is no longer a valid method to invoke, usually by throwing an exception like `UnsupportedOperationException`. Subclasses can only *add*, never *remove* methods and data structures inherited. A classic example is `java.util.Stack`. A stack, by definition, only supports adding and removing elements by `push()` and `pop`. However, to reuse the element storage implementation, the developers have made `Stack` a subclass of `Vector`, which is a linear list collection. That is, an instance of stack also allows elements to be inserted at a specific position, `stack.add(7, item)`, which is forbidden by a stack’s contract!

Composing behavior, in contrast, leads to more fine-grained abstractions. Each abstraction can be highly focused on a single task. Thus cohesion is high as there is a clear division of responsibilities.

**Exercise 16.2:** Apply the ② principle to the stack example above so clients cannot invoke methods that are not part of a stack’s contract but the stack abstract still reuses the vector’s implementation.

**Compile-time versus run-time binding.** Class inheritance defines a compile-time coupling between a subclass and its superclass. Once an object of this class has been instantiated its behavior is defined once and for all throughout its lifetime. In contrast, an object that provides behavior by delegating partial behavior to delegate objects *can* change behavior over its lifetime, simply by changing the set of delegate objects it uses. For instance, you can reconfigure a Alphatown pay station to become a Betatown pay station even at run-time simply by changing what rate strategy and what factory it uses.

**Exercise 16.3:** Extend the pay station so it can be reconfigured at run-time by providing it with a new factory object. You will have to introduce a new method in the PayStation interface, for instance

```
public void reconfigure(PayStationFactory factory);
```

**Recurring modifications in the class hierarchy.** A force I have often seen in practice is that classes in a hierarchy have a tendency to be modified often, as new subclasses are added. As an example, consider a service that fulfills its contract nicely using a simple `ArrayList` data structure, see a) in Figure 16.3. Later I need a better performing service implementation but if I simply subclass the original service class I have to override all methods to use a better performing data structure, and instantiated objects will contain both structures. The logical consequence is to modify the class hierarchy by adding a common, abstract, class, as shown in pane b) of the figure. While the modification is sensible, it does mean that three classes are now modified

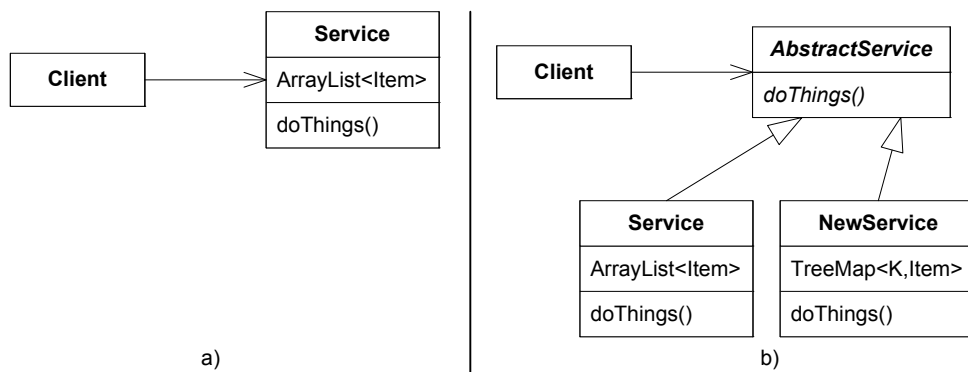


Figure 16.3: Modifications in hierarchy.

and have to be reviewed and tested to ensure the former reliability. Often, each new subclass added provides opportunities for reworking the class hierarchy and as a consequence the stability quality suffers. In Chapter 11, *Deriving State Pattern*, I discussed the tendency for subclass specific behavior “bubbling” up the hierarchy to end in the abstract superclass that becomes bigger and less cohesive over time.

**Copyrighted Material. Do Not Distribute.**



In a compositional design, the array list based and tree map implementations would be separate implementations without overlap. However, this benefit may turn into a liability if the `AbstractService` can make concrete implementations of many of the methods. In that case, a compositional proposal would end up with duplicated code or be required to do further decomposition to avoid it. Thus, in this case one should carefully consider benefits and liabilities before committing to either solution. This particular case is explored in the exercise below.

**Exercise 16.4:** To make Figure 16.3 concrete, consider that the service is a list storing integers. A demonstration implementation may look like

Fragment: exercise/compositional-principles/InitialImplementation.java

```
class IntegerList {
    private int contents[]; int index;
    public IntegerList() { contents = new int[3]; index = 0; }
    public int size() { return index; }
    public boolean add(int e) {
        contents[index++] = e;
        return true;
    }
    public int get(int position) { return contents[position]; }
    // following methods are data structure independent
    public boolean isEmpty() { return size() == 0; }
    public String contentsAsString() {
        String result = "[";
        for ( int i = 0; i < size()-1; i++ ) {
            result += get(i)+", ";
        }
        return result + get(size()-1)+"]";
    }
}
```

Note that the two last methods are implemented using only methods in the class' interface, thus they can be implemented once and for all in an abstract class.

Take the above source code and implement two variants of an integer list: one in which you subclass and one in which you compose behavior. Evaluate benefits and liabilities. How can you make a compositional approach that has no code duplication and does not use an abstract class?

**Separate testing.** Objects that handle a single task with a clearly defined responsibility may often be tested isolated from the more complex behavioral abstraction they are part of. This works in favor of higher reliability. Dependencies to *depended-on units* may be handled by test stubs. The separate testing of rate strategies, outlined in Chapter 8, is an example showing this.

**Increased possibility of reuse.** Small abstractions are easier to reuse as they (usually) have fewer dependencies and comes with less behavior that may not be suitable in a reusing context. The selection handler abstraction in MiniDraw, described in the previous chapter, is an example of this.

**Increased number of objects, classes, and interfaces.** Having two, three, or several objects doing complex behavior instead of a single object doing it all by itself naturally leads to an increase in the number of objects existing at run-time; and an increase in the number of classes and interfaces I as a developer have to overview at compile-time. If I cannot maintain this overview or I do not understand the interactions then defects will result. It is therefore vital that developers *do* have a roadmap to this web of objects and interfaces in order to overview and maintain the code. How to maintain this overview is the topic of Chapter 18.

**Delegation requires more boilerplate code.** A final liability is that delegation requires more “boilerplate” code. If I inherit a superclass, you only have to write Class B **extends** A and all methods are automatically available to any B object without further typing. In a compositional design, I have potentially a lot of typing to do: create an object reference to A, and type in all “reused” methods and write the delegation code:

```
void foo() { a.foo(); }  
int bar() { return a.bar(); }
```

## 16.4 Third Principle

③ *Consider what should be variable in your design.*

This is the most vague of the three principles (perhaps the reason that Gamma et al. did not themselves state it as a principle). Instead of considering what might force a design change you must focus on the aspects that you want to vary—and then design your software in such a way that it can vary *without* changing the design. This is why it could be reformulated as *Encapsulate what varies in your design*: use the first two principles to express the variability as an interface, and then delegate to an object implementing it.

This principle is a recurring theme of many design patterns: some aspect is identified as the variable (like “business rule/algorithm” in STRATEGY) and the pattern provides a design that allows this aspect to vary without changes to the design but by changes in the configuration of objects collaborating.

## 16.5 The Principles in Action

The principles can be used by themselves but as I have pointed out throughout this book they often work nicely in concert: the ③-①-② process.

③-**Consider what should be variable.** I identify some behavior in an abstraction that must be variable, perhaps across product lines (Alphatown, Betatown, etc.), perhaps across computing environments (Oracle database, MySQL database, etc.), perhaps across development situations (with and without hardware sensors attached, under and outside testing control, etc.) as shown in Figure 16.4.

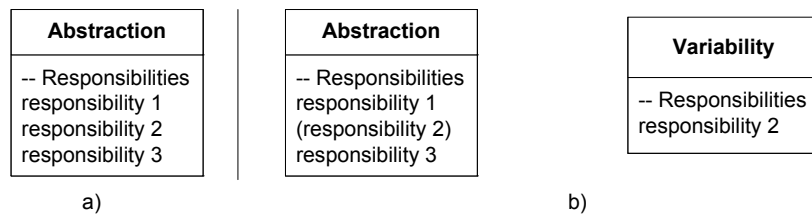


Figure 16.4: A responsibility (a) is factored out (b).

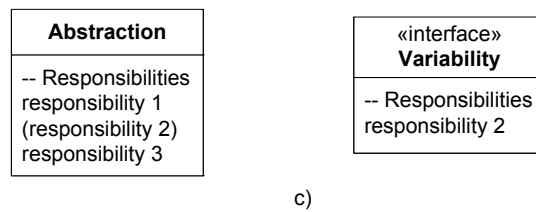


Figure 16.5: Expressing it as an interface (c).

①—**Program to an interface, not an implementation.** I express that responsibility that must be variable in a new interface, see Figure 16.5.

②—**Favor object composition over class inheritance.** And I define the full, complex, behavior by letting the client delegate behavior to the subordinate object: *let someone else do the dirty job*, as seen in Figure 16.6.

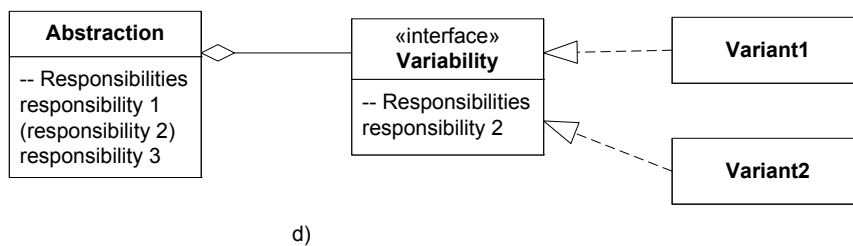


Figure 16.6: Composing full behavior by delegating (d).

Remember, however, that the ③-①-② is not a process to use mechanically. As was apparent in the discussion of the abstract factory you have to carefully evaluate your options to achieve a good design with low coupling and high cohesion. Note also that they are *principles*, not *laws*. Using these principles blindly on any problem you encounter may “over-engineer” your software. If you are not in a position where you can utilize the benefits then there is little point in applying the principles. Remember the TDD value: *Simplicity*: You should build or refactor for flexibility when need arises, not in anticipation of a need. Often when I have tried to build in flexibility in anticipation of a need, I have found myself guessing wrong and the code serving the flexibility actually gets in the way of a better solution.

## 16.6 Role Interfaces

Role interfaces is an interface which specifies a highly specific and often partial role of an object. It was first formulated by the name Private interface by James Newkirk (Newkirk 1997) as a pattern whose intent is

### Definition: Private Interface

Provide a mechanism that allows specific classes to use a non-public subset of a class interface without inadvertently increasing the visibility of any hidden member variables or member functions.

Later Martin Fowler (Fowler 2006) expressed much the same by the term Role Interface which I will prefer, as the term is more in line with the idea of objects having different roles in different contexts.

Role interfaces are an excellent way to model situations in which two delegates need to collaborate more closely without breaking encapsulation.

To illustrate this point, let me demonstrate by a (contrived) example of a new requirement to the PayStation from OmegaTown. OmegaTown has asked for a **RateStrategy** defined as

- One cent of payment gives two minutes parking.
- When exactly 100 cents have been entered, a 25 cent bonus is awarded.

Specifically, the last requirement should be handled by increasing the pay station's internal state so 25 cents is added to the internal integer holding the inserted payment.

This is a really weird and contrived requirement, but the point is that our current design of the rate strategy

Listing: chapter/refactor/iteration-6/src/main/java/paystation/domain/RateStrategy.java

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

is not sufficient, as the implementation has no way of interacting with the pay station nor modifying its internal state: it is only provided an integer value when the calculateTime() method is called.

This is actually a difficult requirement to satisfy, given our current design. What are my options?

In the next sections, I will treat a number of options that comes to my mind.

Copyrighted Material. Do Not Distribute.

## 16.6.1 Pass PayStation + Augment Interface

The most direct route is to refactor the interface `RateStrategy`'s `calculateTime()` method to accept a reference to the paystation:

```
public interface RateStrategy {
    int calculateTime(PayStation ps);
}
```

And refactor the call site in `addPayment()`:

```
public void addPayment( int coinValue )
    throws IllegalArgumentException {
    ...
    insertedSoFar += coinValue;
    timeBought = rateStrategy.calculateTime( this );
}
```

Now, the implementation of `OmegaTown`'s rate strategy can interact directly with the pay station it is associated with. However, for all the previous towns, we face an issue, namely that the `PayStation` interface has not accessor method to retrieve the "insertedSoFar" value!

Thus, to make all the old variants work, we are forced to introduce another method to the pay station interface, and thus another responsibility to the role.

```
public interface PayStation {
    [old methods]
    int getInsertedSoFar ();
}
```

This is not ideal as we are now "blobbing" the interface with an accessor to an internal value which was meant to be encapsulated. Even worse, the `OmegaTown`'s requirement force us to even add another method to the interface:

```
public interface PayStation {
    [old methods]
    int getInsertedSoFar ();
    void adjustInsertedSoFar( int newValue );
}
```

Now the `OmegaTown`'s rate strategy can implement the requirement, however, all other clients are now *also* allowed to fiddle with the internal state of the pay station.

Conclusion: This approach breaks encapsulation and bloats the interface. Maintainability and notably Stability (Chapter 3) suffer. It is a terrible solution. *Do Over!*

## 16.6.2 Pass PayStationImpl

The main issue of the previous approach was that it exposed internal details in the public interface. One way to avoid this is simply just to expose methods only at the implementation level—in `PayStationImpl` only. Thus I may change the rate strategy to:

```
public interface RateStrategy {
    int calculateTime(PayStationImpl ps);
}
```

This helps a lot because the needed methods can then *only* be defined in the implementing class:

```
public class PayStationImpl implements PayStation {
    [old methods]
    int getInsertedSoFar() { return insertedSoFar; }
    void adjustInsertedSoFar(int newValue) {
        insertedSoFar = newValue;
    }
}
```

Now, any client that only refers to instances of the interface, `PayStation`, like the GUI does, cannot read or modify the internal “insertedSoFar” in the pay station.

A variant of this solution, is to keep the original `RateStrategy` interface from the previous section (that is, having a `PayStation` parameter), and then do a cast in the concrete rate strategy ala

```
PayStationImpl psImpl = (PayStationImpl) ps;
int insertedSoFar = psImpl.getInsertedSoFar();
...
```

So, this is a far better solution. But it has its share of liabilities.

First, it does not adhere to the ① *Program to an interface* principle. Suddenly the rate strategy has tight coupling with the implementing class, and I can never substitute another implementation than the `PayStationImpl` class in my system. Either I will get a compile error, or an error in the cast in the second variant.

Second, the `RateStrategy` has full access to all aspects of the `PayStation`. It can call the `buy()` method! Or `cancel()`. Of course, this does not make sense at all to do so—and I will definitely find out during testing. However, the point is our rate strategy has too much access and our Java compiler cannot warn us about that fact.

Conclusion: Better but not good.

### 16.6.3 ModifiablePayStation Role

Again, let us attack the main issue of the previous solution: That the rate strategy has too much access of the pay station provided.

I can solve that by introducing a role interface, just for the purpose of giving any rate strategy access to required actions:

```
public interface ModifiablePayStation {
    int getInsertedSoFar();
    void adjustInsertedSoFar(int newValue);
}
```

I then refactor the rate strategy to become:

```
public interface RateStrategy {
    int calculateTime(ModifiablePayStation ps);
}
```

Now, our implementation of the pay station implements both the usual `PayStation` interface and in addition, this new interface:

```
public class PayStationImpl implements PayStation ,
    ModifiablePayStation {

    [old methods here ...]

    // Implementing the private interface methods
    @Override
    public int getInsertedSoFar() {
        return insertedSoFar;
    }

    @Override
    public void adjustInsertedSoFar(int newValue) {
        insertedSoFar = newValue;
    }
}
```

Now, again we pass the `this` reference at the call site:

```
public void addPayment( int coinValue )
    throws IllegalCoinException {
    ...
    insertedSoFar += coinValue;
    timeBought = rateStrategy.calculateTime( this );
}
```

which is perfectly correct, as ‘`this`’ is of course an `ModifiablePayStation`.

This solves all issues. A rate strategy is passed a reference to a `ModifiablePayStation`, and can thus only access and mutate via the two methods it that interface. Thus, it is not possible for it to call `cancel()` or `buy()` as was the case of the previous solution.

Note that this is an example of the ability of interfaces to “express fine-grained responsibilities” as described in Section 16.2.

## 16.7 Interface Segregation Principle

The **Interface Segregation Principle** is according to Wikipedia:

### Definition: Interface Segregation Principle

In the field of software engineering, the interface segregation principle (ISP) states that no code should be forced to depend on methods it does not use. ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces.

So, as also evident, this principle is closely tied to Role Interfaces, and you have already seen the principle in action in the previous section: The `PayStationImpl` needs to satisfy two roles—the original `PayStation` interface towards the user interface; and the much more specialised (private) role of allowing internal strategies to adjust its

internal state via the `ModifiablePayStation`. Thus the full set of responsibilities are segregated into two more specialized role interfaces.

Another good example is drawn from the game project in the last part of the book. In the `HotStone` project, a card game, the game must ensure that the attributes of the playing card are exposed so a user interface can render graphics that show cards and their values to the user on one hand, but at the same time ensure that no external object modifies the card's internal state. We do not want a clever user to increase the value of a card and thus break the rules of the game.

We can solve this by the ISP: The card interface must be split into role interfaces. The one exposed to the user interface only contains accessor methods (so no modification can be made) while another only exposed internally allows card attributes to be modified.

## 16.8 SOLID

SOLID is an mnemonic acronym that is often used, which basically covers the principles stressed throughout this book. The principles were promoted by Robert C. Martin (Martin 2000) while Michael Feathers later introduced the acronym itself (Feathers 2017).

The principles are:

- S The single-responsibility principle: "There should never be more than one reason for a class to change." That is, encapsulate behavior in well-defined and fine-grained roles; encapsulate what varies.
- O The open-closed principle: "Software entities . . . should be open for extension, but closed for modification." That is, favor change by addition.
- L The Liskov substitution principle: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." That is, program to an interface.
- I The interface segregation principle: "Many client-specific interfaces are better than one general-purpose interface." That is, express behavior using fine-grained roles.
- D The dependency inversion principle: "Depend upon abstractions, [not] concretions." That is, program to an interface, and favor object composition by dependency injection.

## 16.9 Summary of Key Concepts

Three principles are central for designing compositional designs. These are:

**Copyrighted Material. Do Not Distribute.**



### Compositional Design Principles:

- ① *Program to an interface, not an implementation.*
- ② *Favor object composition over class inheritance.*
- ③ *Consider what should be variable in your design.*  
(or: *Encapsulate the behavior that varies.*)

Generally, applying these patterns makes your design more flexible and maintainable as abstractions are more loosely coupled (first principle), bindings are run-time (second principle), and abstractions tend to become smaller and more cohesive. The third principle is a keystone in many design patterns that strive to handle variability by encapsulation, using the first two principles.

## 16.10 Selected Solutions

### Discussion of Exercise 16.1:

One possible way would be to create a subclass `AbstractServiceD` and let it create an instance of `AbstractServiceC`. All methods in `AbstractServiceD` (which are the ones the client invokes) are overridden to call appropriate methods in `AbstractServiceC`. However, the construction is odd, as `AbstractServiceD` of course inherits algorithms and data structures from `AbstractService` that are not used at all.

This proposal resembles the ADAPTER pattern, however adapter is fully compositional.

### Discussion of Exercise 16.4:

You can find solutions to the exercise in folder *solution/compositional-principles*. Basically, you can do the same thing with a compositional design as with an abstract class: you factor out common code into a special role, `CommonCollectionResponsibilities`, implement it, and delegate from the implementations of the integer list. However, due to the delegation code and extra interfaces, the implementation becomes longer (100 lines of code versus 85) and more complex.

## 16.11 Review Questions

What are the three principles of flexible software design? How are they formulated? Describe and argue for their benefits and liabilities.

How do these principles relate to patterns like STRATEGY, ABSTRACT FACTORY and others that you have come across?

What are the alternative implementations that arise when these principles are not followed?

## 16.12 Further Exercises

### Exercise 16.5:

Many introductory books on object-oriented programming demonstrate generalization/specialization hierarchies and inheritance by a classification hierarchy rooted in the concept *person* as shown in Figure 16.7. For instance a person can be a teacher or a student and by inheriting from the `Person` class all methods are inherited “for free”: `getName()`, `getAge()`, etc.

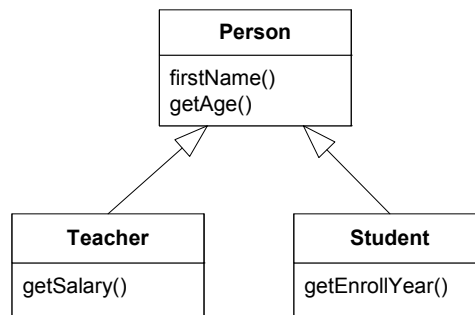


Figure 16.7: A polymorphic design for teachers and students.

This design may suffice for simple systems but there are several problems with this design that you should study in this exercise. You should analyze the problems stated in the context of an object-oriented university management system that handles all the university’s associated persons (that is both teachers and students).

**Life-cycle problem.** Describe how to handle that a student graduates and gets employed as a teacher?

**Context problem.** Describe how the system must handle that a teacher enrolls as student on a course? How must the above class diagram be changed in order to fully model that a person can be both a student as well as a teacher?

**Consistency problem.** Describe how the system must handle a situation where a teacher changes his name while enrolled in a course.

Based on your understanding of roles and by applying the principles for flexible design propose a new design that better handles these problems. Note: It is not so much the ③-①-② process that should be used here as it is the individual principles in their own right.

As a concrete step, consider the following code fragment that describes the consistency problem above:

```

Teacher t = [get teacher 'Henrik Christensen']
Student s = [get student 'Henrik Christensen']

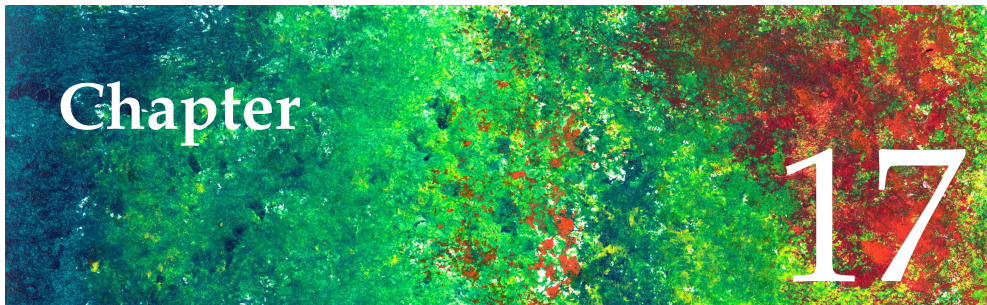
assertEquals('Henrik', t.firstName());
assertEquals('Henrik', s.firstName());
  
```

```
[Person Henrik renamed to Thomas]
assertEquals( 'Thomas' , t.firstName() );
assertEquals( 'Thomas' , s.firstName() );
```

The point is that the name change should be a single operation at the *person* level (as name changes conceptually has nothing to do with neither Henrik's teacher nor student association).

How would you make the person-teacher-student design so that this test case passes without making changes to multiple objects?





# 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**:

Product	Variability points		
	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.

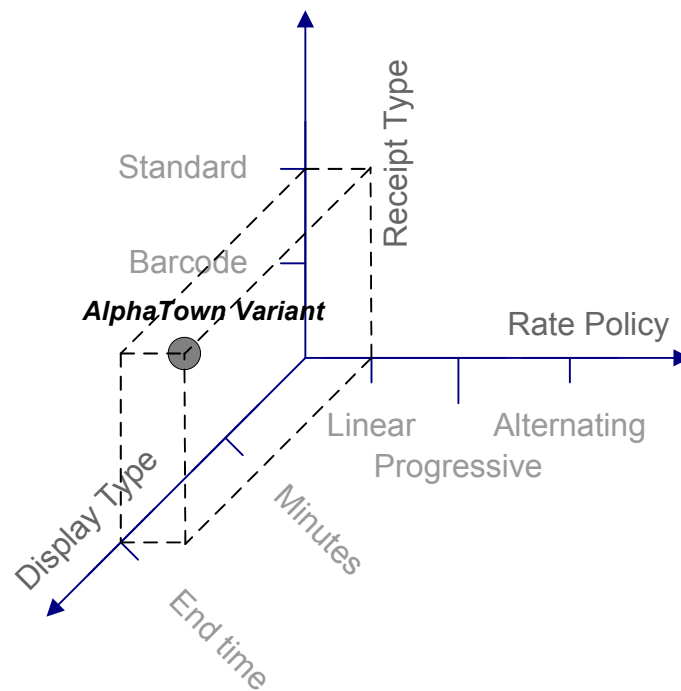


Figure 17.2: The Alphetown variant plotted in variability space.

## 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 Alphetown, Betatown, and Gamma-town 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 `PayStationAlternatingRateBarcodeReceiptEnd-TimeDisplay`. 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.

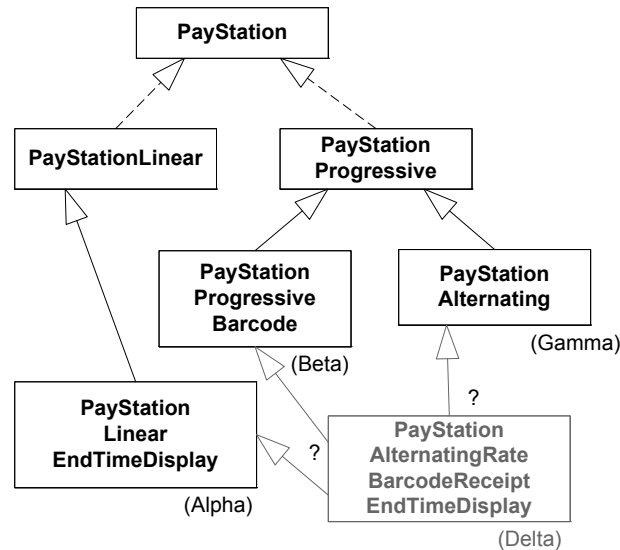


Figure 17.3: A combination of variability handled by inheritance.

**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 Delta-town variants in Figure 17.3.

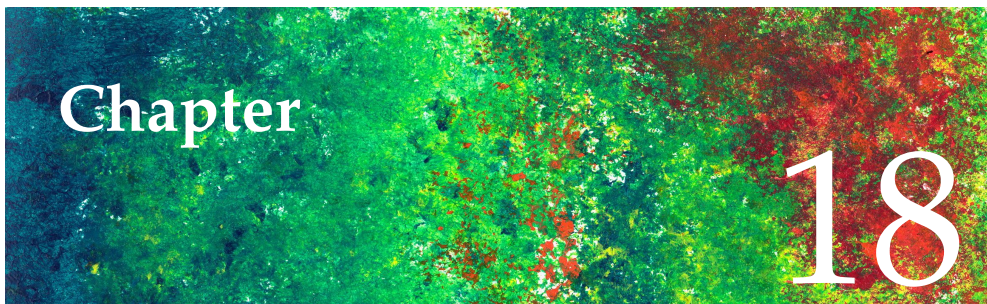
### Exercise 17.4. Source code directory:

```
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 its state (the amount of payment entered so far) during reconfiguration.



# Design Patterns – Part II

## Learning Objectives

In Chapter 9, *Design Patterns – Part I*, I presented two definitions of design patterns, one by Gamma et al. and one by Beck et al. The learning objective of this chapter is to present two additional definitions of design patterns in order to convey an even deeper understanding of the concept. The first definition recasts patterns in the responsibility-centric perspective, and the second looks at how patterns are key concepts for understanding and keeping overview of complex, compositional, designs.

## 18.1 Patterns as Roles

The structure of design patterns are usually documented using UML class diagrams. The diagrams for STRATEGY and STATE, reproduced in Figure 18.1 and Figure 18.2, consist of interface, classes, and relationships.

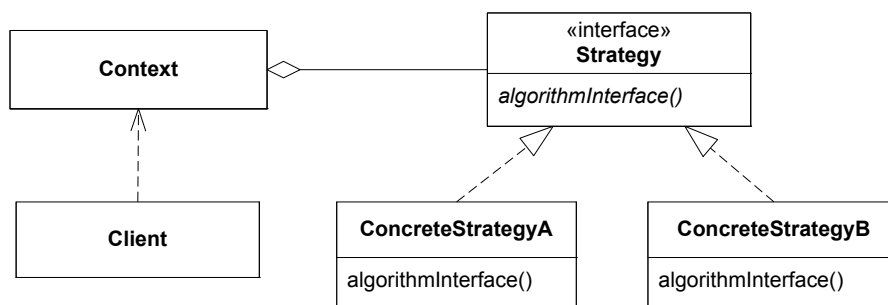


Figure 18.1: STRATEGY pattern structure in UML.

Now, take a moment to compare the design pattern structure diagrams with the design of the special Gammatown rate strategy, reproduced in Figure 18.3.

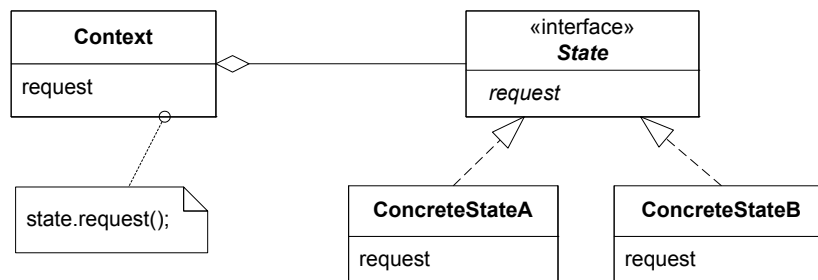


Figure 18.2: STATE pattern structure in UML.

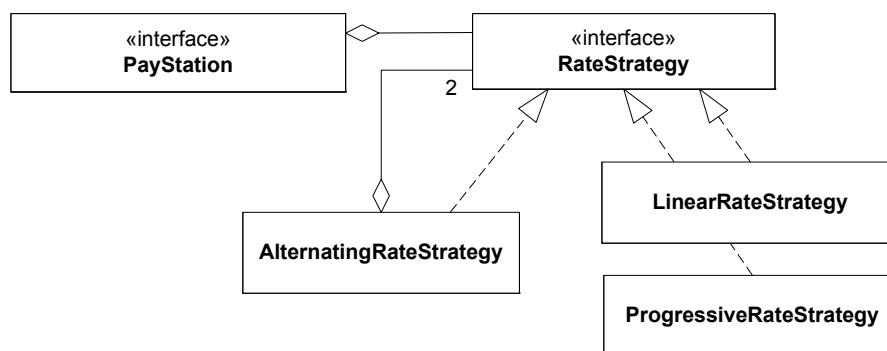


Figure 18.3: The combination of STRATEGY and STATE.

Do you see a problem? If I concentrate on the two patterns STRATEGY and STATE then the UML class diagrams for these patterns state that there is an interface called **Strategy** and an interface called **State** in the respective patterns. Looking at Figure 18.3 though, I can find the strategy interface (**RateStrategy**)—but where is the interface defining the **State** role that appears in the UML for the STATE pattern? Apparently—it is not there. The question is then: Are the pattern diagrams wrong? Or is it my pay station design that is wrong?

The answer lies in understanding that design patterns are *not* a fixed set of classes and interfaces. Interfaces and classes are the tools available in UML but what design patterns really express are roles.

### Definition: Design pattern (Role view)

A design pattern is defined by a set of roles, each role having a specific set of responsibilities, and by a well-defined protocol between these roles.

The point is that when you see the **Strategy** interface in the STRATEGY pattern you must think of it as a *role*. In the pay station design the **Strategy** role is defined by the **RateStrategy** interface. And—the **State** role from the STATE pattern is also present—it is defined by the very same **RateStrategy** interface: it defines the common interface for all **ConcreteState** instances.

Thus I can annotate the abstractions for this part of the pay station design with the corresponding roles from the design patterns as done in Figure 18.4. I will call this type of diagram a role diagram.

**Definition: Role diagram**  
 A role diagram is a UML class diagram in which gray boxes either above or below each interface or class describe the abstraction’s role in a particular pattern. The role is described by **pattern-name:role-name**.

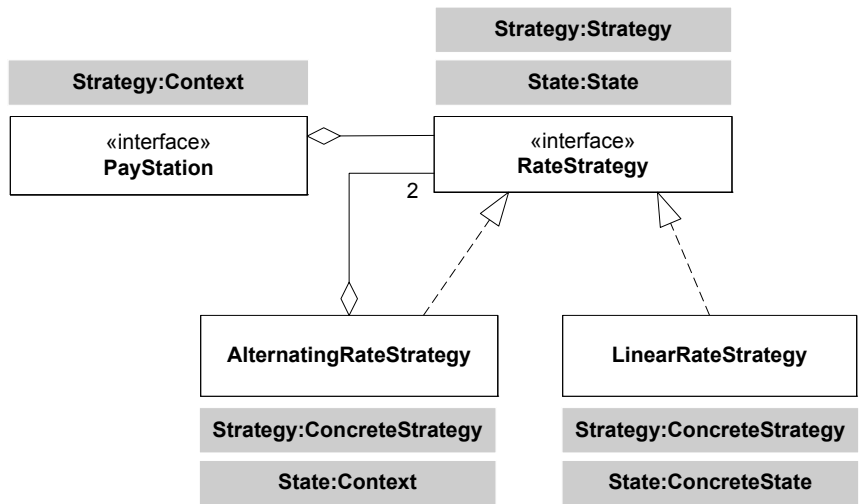


Figure 18.4: Partial pay station design annotated with roles.

Thus the RateStrategy interface serves both the strategy role in the Strategy pattern as well as the State role in the State pattern. It is even more evident at the object level where a concrete instance of LinearRateStrategy may play both the **Strategy:ConcreteStrategy** role as well as the **State:ConcreteState** role.

**Exercise 18.1:** Consider class AlternatingRateStrategy in role diagram 18.4. Explain why it can play the same role as LinearRateStrategy in STRATEGY but a different role in State? What consequences does it have on the coding? You may review the role section of respective design pattern boxes to answer this question.

**Exercise 18.2:** STRATEGY also defines a **Client** role. It is not shown on Figure 18.4. Which abstraction plays this role in the pay station system?

## 18.2 Maintaining Compositional Designs

I have argued in favor of compositional design, using design patterns, and stressed that my pay station production code has become flexible. The question is whether it is also *maintainable*? Remember that the definition of maintainability is:

Copyrighted Material. Do Not Distribute.

### Definition: Maintainability (ISO 9126)

The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

Maintainability is closely related to the architects' and developers' ability to read, understand, and reason about the system's behavior. If you consider the relatively small functional requirements of the pay station it may come as a surprise to see the full class diagram in Figure 18.5.

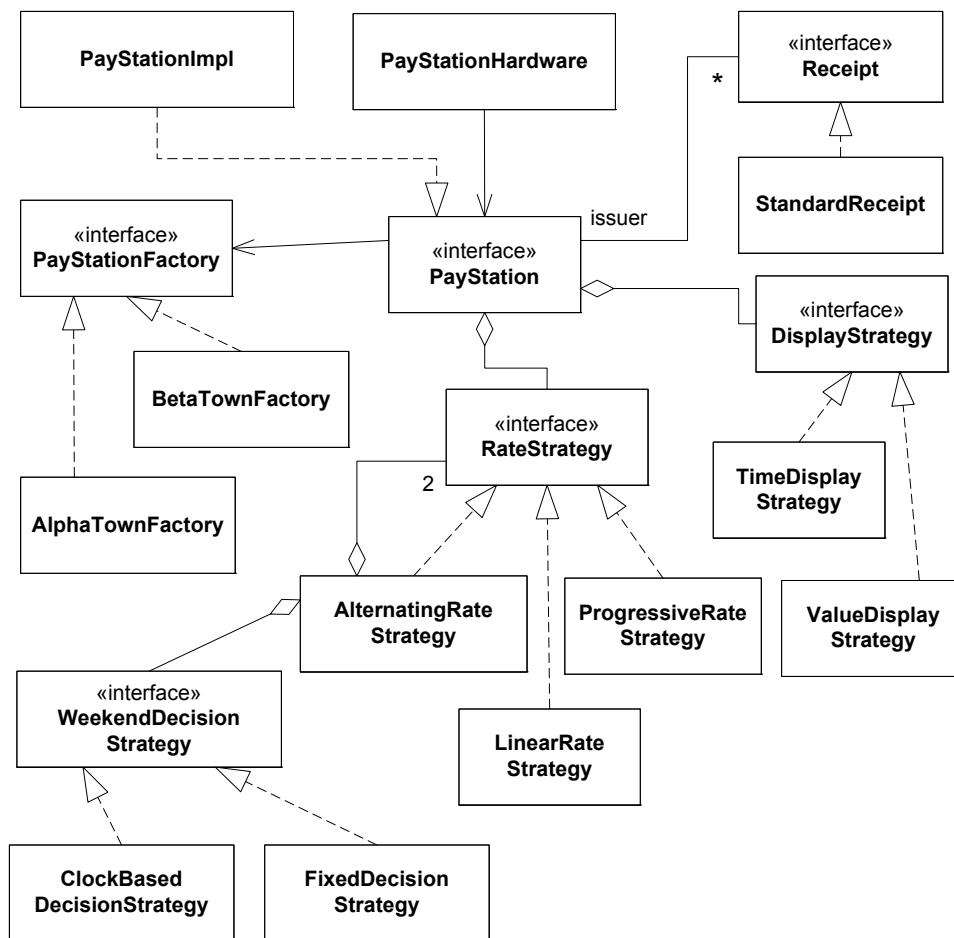


Figure 18.5: The pay station class diagram.

**Exercise 18.3:** Even though the class diagram is pretty large, it is actually a simplification compared to the real system. Identify some classes and relations that have been omitted in the diagram.

After all, the pay station is a behaviorally simple system, but apparently I have turned the class diagram into a big mess. OK, it is flexible and can exist in many different

variants, but nonetheless the resulting design *is* complex. So, the really good question is: how do I overview and understand this big mess of classes? And even more challenging: communicate it with my fellow developers and architects? I could, as I have indeed done, write a several hundred pages long book about the design but this is of course absurd in real development where the business lies in the software produced, not in the teaching aspects nor the book produced.

This is indeed a major problem in compositional designs and frameworks that have a lot of delegation, lots of interfaces, and many aspects that may vary. The cure to the problem is to *know your patterns* and *document them in the design*. When I deeply understand the roles and protocols of the design patterns in my design, there are lots of details that I can either omit from my class diagrams or that I can read from the diagram without getting confused: all the jigsaw pieces form an understandable picture.

I have redrawn Figure 18.5 with emphasis on the central roles in the patterns used—and grayed those roles that are “merely” implementations of the roles, see Figure 18.6. For instance, once I understand that the `RateStrategy` interface in the diagram plays the **Strategy** role of the STRATEGY pattern I *know* that there must be several **ConcreteStrategy** objects in the diagram as well—and indeed there are: `LinearRateStrategy` and its siblings. They do not confuse, though. I know they have been there in order to provide concrete behavior to the strategy role.

Using the pattern’s name directly in naming interfaces and classes provides strong clues to the patterns that are in play in a design, here ABSTRACT FACTORY and STRATEGY. It is not always possible as for instance when a given abstraction plays a part in several patterns. An example is `AlternatingRateStrategy` that plays both the **Strategy:Strategy** role as well as the **State:Context** role. However, the Strategy role is always in play whereas it is only for the Gammatown pay station that it acts as context for state. However, pointing towards the state pattern turned too odd in the naming. Maybe `AlternatingRateStrategyStateContext` but it becomes too long for my taste.

The documentation (JavaDoc or the like) of the interfaces and classes should describe the patterns they are part of as well as the roles. This is more important for the central roles and less so for the concrete implementations.

In the diagram, note also that there are quite a lot of relations that I have not drawn at all. I have omitted the relations between the ABSTRACT FACTORY and the strategies they instantiate. The reason is that the diagram would become cluttered with association lines. This would make it harder to read even though it would be more correct. I consider class diagrams to be *roadmaps* that should emphasize overview more than correctness. As I know my ABSTRACT FACTORY pattern well I know that each factory has relations to a set of **ConcreteProduct** roles. If I am interested in finding the true configuration for, say `AlphaTownFactory`, then I must go to the production code (or a table in the documentation, perhaps) to extract this information. The class diagram provides the roadmap while the production code provides the “truth” about the details of the road.

Thus, it is possible to provide yet another classification of what design patterns are:

### Definition: Design pattern (Roadmap view)

Design patterns structure, document, and provide overview of the roles and protocols in complex, compositional, designs. A design pattern serves as a roadmap of a part of the design.

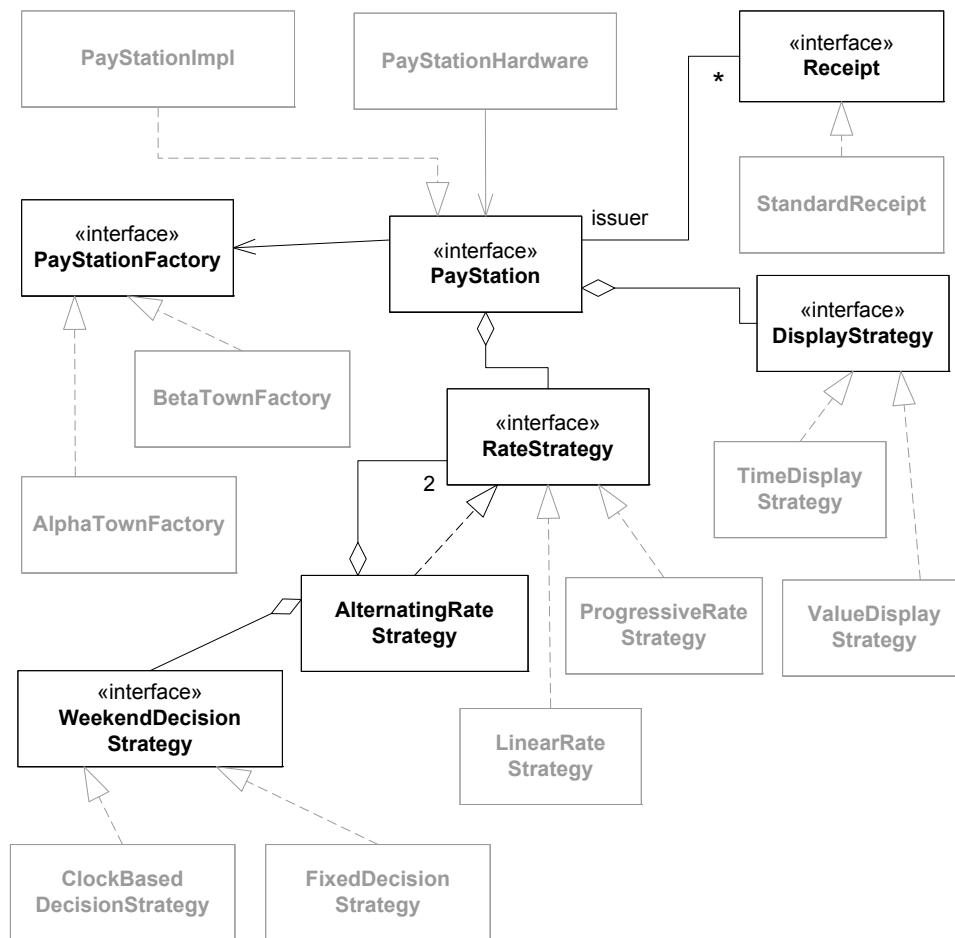


Figure 18.6: The central roles in the class diagram.

The argumentation above can also be reversed. What happens if a developer that has never received any training in design patterns and compositional design processes is requested to make modifications or implement new requirements in the design? The answer is that he *will* just see one big mess of interfaces and classes that interact in, to him, overly complex ways. I once participated in a research project where this happened (Christensen and Røn 2000) and the conclusion was that “The newcomers were unable to produce code that followed the intent of the patterns and much code therefore had to be rewritten.”

**Key Point: Maintaining compositional designs requires appropriate training**

*Compositional designs are complex in terms of the protocols and number of roles and objects involved. Software designers and developers must therefore have a deep understanding of compositional design principles and design patterns in order to successfully maintain such designs.*



**Exercise 18.4:** You now have four different definitions of design patterns, two from Chapter 9 and two from this chapter. Argue why it is possible to have different definitions without them being in conflict.

## 18.3 Summary of Key Concepts

The structure of design patterns are normally documented by UML class diagrams, but patterns are not a fixed set of interfaces and classes. Instead *patterns are a set of roles and their protocols* which can be mapped to concrete interfaces and classes in many different ways. A *role diagram* annotate each class or interface in a class diagram with the respective roles they play in a set of patterns.

Compositional designs are complex because of the large number of interfaces and classes that interact in complex ways. Therefore, they present a challenge to software architects and developers maintaining these systems because they need to overview the structure, reason about its behavior, and avoid introducing code that breaks the conventions. A deep knowledge of the principles of flexible design and design patterns is vital in order to cope with this complexity. Design patterns define the central roles that each object will ultimately play and the protocols that govern how it will interact. Thus, one can define *design patterns as a roadmap of a design* that allows developers to maintain overview.

## 18.4 Selected Solutions

### Discussion of Exercise 18.1:

Class `AlternatingRateStrategy` as well as `LinearRateStrategy` serve the **ConcreteStrategy** role that is described in design pattern STRATEGY as *defines concrete behavior fulfilling the algorithmic responsibility*. As both classes do the same thing, namely perform the rate policy calculation, they obviously serve this role.

In the STATE pattern, however, the **Context** object delegate to its current state object while the **ConcreteState** objects define the specific behavior associated with each specific state. And here the difference appears also at the coding level. The `AlternatingRateStrategy` contains the state object decision code as well as the delegation and therefore conforms to the **Context** role. The `LinearRateStrategy` only contains algorithmic code, that is, the state specific behavior.

### Discussion of Exercise 18.2:

The **Strategy:Client** role is played by the pay station hardware (or the GUI or the JUnit integration test cases).

### Discussion of Exercise 18.3:

First of all, the `GammaTownFactory` class is missing (as is the “helper” class `IllegalCoinException` which it purposely has been on all diagrams). Also the special test rate strategy `One2OneRateStrategy` is not shown. And a final point, all the relations between the factories to the concrete products are missing. If they had been

included, the diagram would have been heavily cluttered with association lines that would have made it difficult to read.

#### Discussion of Exercise 18.4:

The definitions are not in conflict. The reason that it makes sense to have several different definitions of the same concept is that the concept is deep and has many facets. The four different definitions represent different perspectives or views upon the same concept and as such they complement and extend each other.

## 18.5 Review Questions

What is the definition of design patterns from the role view? Argue why design patterns are not a fixed set of interfaces and classes but must be understood as a set of roles and protocols.

What does a role diagram show?

What is the definition of design patterns from the roadmap view? How can design patterns provide overview and understanding in complex designs made by a compositional design approach?

What are the rules for naming interfaces and classes in complex designs using design patterns? What is the justification for these rules?

## 18.6 Further Exercises

#### Exercise 18.5:

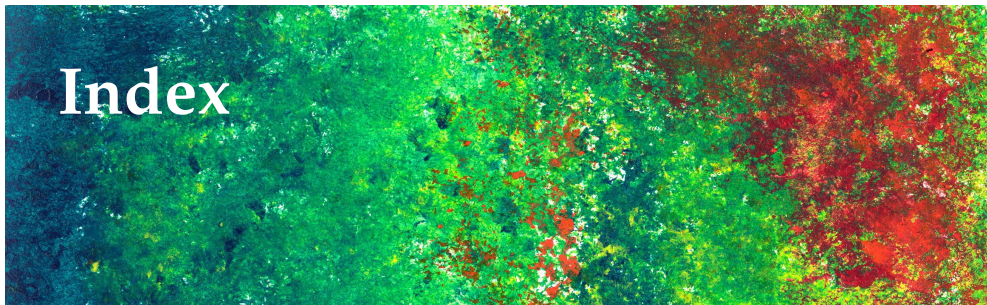
Draw one or several role diagrams that outline all roles for all interfaces/classes in the pay station design.



# Bibliography

- Barnes, D. J. and M. Kolling (2005). *Objects First with Java: A Practical Introduction Using BlueJ, 2nd ed.* Prentice Hall.
- Beck, K. and W. Cunningham (1989). A Laboratory for Teaching Object-Oriented Thinking. In *Proceedings of SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Volume 24 of SIGPLAN Notices, pp. 1–7.
- Budd, T. (2002). *An Introduction to Object-Oriented Programming.* Addison-Wesley.
- Christensen, H. B. (2005, June). Implications of Perspective in Teaching Objects First and Object Design. In *Proceedings of 10th Annual Conference on Innovation and Technology in Computer Science Education*, Lisbon, Portugal.
- Christensen, H. B. and H. Røn (2000, November). A Case Study of Framework Design for Horizontal Reuse. In *Proceedings of 37th International Conference on Technology of Object-Oriented Languages and Systems*, Sydney, pp. 278–289. IEEE Computer Society Press.
- Feathers, M. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Prentice Hall.
- Fowler, M. (2006). Role interface. <https://www.martinfowler.com/bliki/RoleInterface.html>.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.
- Grand, M. (1998). *Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns.* John Wiley and Sons, Inc.
- Kay, A. (1977). Microelectronics and the Personal Computer. *Scientific American* 237(3), 230–244.
- Madsen, O. L., B. Møller-Pedersen, and K. Nygaard (1993). *Object-Oriented Programming in the BETA Programming Language.* Addison Wesley.
- Martin, R. C. (2000). Principles of ood. <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>.
- Newkirk, J. (1997). Private interface. In *Proceedings of Pattern Language of Programs.*
- Savitch, W. (2001). *Java: An Introduction to Computer Science and Programming.* Prentice Hall.
- Shalloway, A. and J. R. Trott (2004). *Design Patterns Explained: A New Perspective on Object-Oriented Design, 2nd ed.* Addison-Wesley.

- Snyder, A. (1986). Encapsulation and Inheritance in Object-Oriented Languages. In *Proceedings of SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, pp. 38–45. ACM Press.
- Wirfs-Brock, R. and A. McKean (2003). *Object Design – Roles, Responsibilities, and Collaborations*. Addison-Wesley.



- ③-①-② process, **21–22**
- behavior, **9**
- blob, **17**
- boilerplate code, **28**
- combinatorial explosion, **40**
- configuration table, **40**
- contract, **22**
- CRC card, **8, 11, 14**
- design pattern
  - roadmap view, **49**
  - role view, **46**
- encapsulate variable behavior, **21, 28**
- explosion, combinatorial, **40**
- favor object composition, **21, 24–28**
- flyer (role), **13**
- Interface Segregation Principle, **33, 33**
- maintainability (ISO), **48**
- multi-dimensional variability, **40**
- object-orientation
  - language perspective, **6**
  - model perspective, **7**
  - responsibility perspective, **9**
- principle
  - compositional design, **21**
- Private Interface, **24, 30**
- private interface, **33**
- program to an interface, **21–24**
- protocol, **13**
- responsibility, **10**
- role
  - general, **12**
  - in software, **13**
  - role box, **14**
  - role description box, *see* role box
  - role diagram, **47**
  - Role Interface, **24**
  - role interface, **32**
  - variability
    - multi-dimensional, **40**
  - variability space, **40**
  - what/who cycle, **15**
  - who/what cycle, **15**





## Index of Sidebars/Key Points

- Define small and cohesive roles, 18
- Do not use inheritance to handle multi-dimensional variation, 42
- Maintaining compositional designs requires appropriate training, 50
- The relation between role and object is a many-to-many relation, 12