

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.

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.

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.

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.

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

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 an 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.

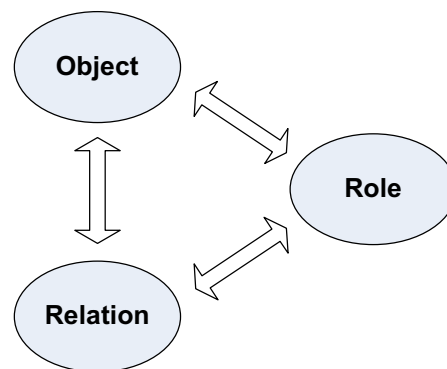


Figure 15.2: The three fundamental concepts in design.

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.

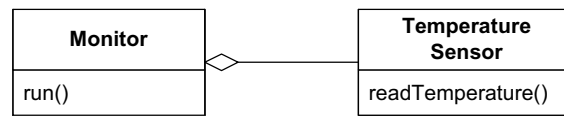


Figure 15.3: Model perspective design.

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 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<Apple> {
    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.

This is best illustrated by an example. The example is from MiniDraw, a two dimensional graphics framework that I will present in learning iteration 7, *Frameworks*. MiniDraw can be used to make drawing programs, like diagram editors for UML or box-and-line schematics. 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 role¹ 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 of the 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.
- Clear a selection.

Thus the **Drawing** interface can actually be defined as composition of these two roles:

¹The 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 {  
    ...  
}
```

This allows the parts of the MiniDraw drawing program that only deals with adding/removing figures to interact with the Drawing object solely via the FigureCollection interface, and similarly the parts that handles selecting figures only to interact via the SelectionHandler interface. Metaphorically, using the example from Section 15.5.3, it is similar to that my students interact with me using the *teacher* role, while my children interact with me using my *father* role. And these two roles, of course, have very different “methods”: I would be puzzled if a student asked me to help fixing a flat tire on his bike—that belongs to the “father” role, certainly not to the “teacher” role.

You may wonder if an object playing several roles may easily end as a *blob* class? That is, a long and complex implementation, as the object has to have methods for all the different roles it has. Indeed, this is a trap, but it can usually be avoided by obeying to the composition principles, ② *Favor object composition over class inheritance*, that I will return to in Section 16.3.

The point is, that often the different roles are rather disjoint functionally, and therefore you can develop individual implementations of each role, and then use composition and delegation.

Returning to MiniDraw for illustration, the FigureCollection and SelectionHandler roles are rather self-contained roles 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.

The CompositionalDrawing code thus looks like this:

```
public class CompositionalDrawing implements Drawing {  
    private FigureCollection figureCollection;  
    private SelectionHandler selectionHandler;  
    [...]  
    public CompositionalDrawing() {  
        figureCollection = new StandardFigureCollection();  
        selectionHandler = new StandardSelectionHandler();  
        [...]  
    }  
    @Override  
    public Figure add(Figure figure) {  
        return figureCollection.add(figure);  
    }  
    [...]  
}
```

Thus, delegation to these smaller role implementations reduces the size and complexity of the CompositionalDrawing substantially. Indeed, the implementation in the current MiniDraw release is less than 130 lines of code.

☞ This is an example where the focus on roles leads to inventing objects, like `StandardSelectionHandler`, 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.

15.8 Interface Segregation Principle

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 and tested in isolation.

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.*

That is, complex roles are segregated into small ones. This is commonly known as the **Interface Segregation Principle**. The definition 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.

15.8.1 Role Interfaces

This “define small and cohesive roles” point is so important that it has led to a specific term for interfaces that express a specific role (Fowler 2006), not surprisingly named:

Definition: Role Interface

A *role interface* is defined by looking at a specific interaction between suppliers and consumers. A supplier component will usually implement several role interfaces, one for each of these patterns of interaction.

Thus, the **FigureCollection** and **SelectionHandler** from Section 15.7.2 are fine examples of role interfaces; they express a “specific interaction”—the **FigureCollection** expresses the specific interaction a 2D graphical editor (the consumer) needs in order to store and remove figures in the drawing (the supplier), and nothing else.

15.8.2 Private Interface

Often, role interfaces are defined for a specific purpose relating to controlling access to object, that is, enforcing proper *encapsulation*. A language like Java has modifiers like *public* and *private* to control access to a method, but often this is too restricted: What if an class wants to express that one specific other class may call a specific method, while all others may not?

We can use role interfaces for that in the form of **Private Interface**. It was first formulated 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.

Private 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.

15.8.3 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!*

15.8.4 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 (Section 16.2). 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.

15.8.5 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 private 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.

15.9 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—the actual algorithm implemented. 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, also termed *role interfaces*. A special kind of role interface, is the *private interface* whose purpose is to provide some specific collaborating class special access to a selected set of methods.

To avoid that a class that implements many role interfaces becomes a blob, the ② *Favor object composition over class inheritance* principle should be used: let it delegate to implemenations of the individual role interfaces.

15.10 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.

Explain what *role interface* is? Explain what *private interface* is? Give examples of both.