Chapter 5
Decoupling through Interfaces

Pedagogic Motivations

- decoupling
- implementation

Decoupling

Before we go too much further with *building* software systems, we should think a little bit about *organizing* software systems. It is generally good to write software which allows its constituent components to vary independently of each other; such software is *flexible* and *adaptable* to new situations. The question is how components of a software system should be organized to achieve flexibility. An important technique for achieving flexibility in software is called *decoupling*. To *couple* means to join together or connect; to *decouple* means to separate. Components which are tightly coupled together cannot change independently of each other, whereas those which are not can. The process of discovering where in a system components can reasonably be decoupled is sometimes referred to as *separation of concerns*. Components that are doing fundamentally different tasks (or, have fundamentally different concerns) can be decoupled from one another.

Abstraction plays an important role in decoupling. Abstraction allows us to look past the specific details of one problem and look to the essential characteristics of a whole class of similar problems. It is often much clearer where decoupling can and should occur when looking at a problem in a general form than it is when looking at a specific problem.

In this chapter we are particularly interested in exploring how to decouple the specification of *what* messages an object must respond to from the definition of *how* an object should respond to those messages.[1] This chapter therefore focuses on the *specification* of capabilities (saying *what* capabilities an object must have). The next chapter discusses the *definition* of capabilities (saying *how* an object should react when one of its capabilities is invoked).

In previous chapters we have seen how to define a class.[2] When it comes to building flexible software systems, it turns out that there is a drawback to using *only* classes, because they define the capabilities of their instances. This may strike you as an odd statement to make, since this is the point of classes. Consider, though, the following example.

---

[1] "What" information is sometimes called *declarative* information. "How" information is sometimes called *procedural* information.

[2] There is quite a bit more we have to learn about defining classes, but we made a start.

Example 1

We want to travel to Aarhus, Denmark from Buffalo, NY. Since we only have a week to complete our travels, we want to travel by airplane. An airplane provides as a core functionality the ability to travel by air. It is self-propelled (to distinguish it from a glider), it remains airborne due to lift generated by stationary wings (to distinguish it from a helicopter), and its takes off and lands on a runway (to distinguish it from a rocket). If we book tickets on an airplane to take us from Buffalo, NY to Aarhus, Denmark, we do not much care whether we travel on a Boeing 747 or an Airbus A320 – they are both airplanes, and satisfy our "airplane" requirement. However, we do not expect to end up on a train, or a boat, since these do not satisfy the "airplane" requirement.

If we model this system using only classes then it seems we would have to model the requirement that we want to travel by air in a very specific way: we would have to say either, we want to travel on a Boeing 747, or we want to travel on an Airbus A320. But this results in a very tight coupling between my requirements and a specific class which satisfies those requirements. It also misses a very clear generalization: we do not care which type of aircraft we travel on; we just care that we travel by plane. Said another way, how can we decouple our travel requirements from any specific class which satisfies them? How can we decouple our travel requirements from any specific implementation of them?

For example, the question boils down to this: if we do not care about the difference between the 747 and the A320, how can we express that? How can we express that we want to travel on an *airplane*, a concept which defines none of the capabilities which are specified for it? We will find out in the next section.

To recapitulate, abstraction and decoupling simplify things: if we have managed to decouple two concepts then we can deal with them one at a time, without worrying about both of them together. In other words, we do not have to worry about interactions between the two, a very important step in managing complexity. This in turn allows us to vary the two independently of each other. This is a very important benefit of decoupling.

Examples of decoupling

It turns out that decoupling is a fundamental technique used throughout computer science. Here are some examples which show a separation of concerns; we do not expect all of these to be familiar to you, but we do hope that you recognize and can relate to at least one:

Example 2

Computer hardware is decoupled from computer software. This means that the hardware and the software of a computer system can (within limits) vary independently of each other. The same software program can be used on computers with different hardware (for example, I can run an IDE called Eclipse on my laptop computer and on my desktop computer, even though the hardware inside those machines is quite different). I can of course also vary the software that I run on a given piece of hardware: on my laptop I can run not only my IDE, but a web browser, a word processor, a spreadsheet, and so on. In fact, I can even add new software programs to run on my machine at any time. This is possible because hardware and software are decoupled.

Example 3

Cascading Style Sheets (CSS) enable the decoupling of the content of a web page from the formatting of the web page. If you have played around with making web pages you have probably come across CSS. CSS is a mechanism whereby you can specify the formatting of a web page (things like background color, font style and size, indentation, etc.) independently of the content of the page. This makes it very easy to change one without affecting the other: the whole look of a web page can be modified by swapping one style sheet for another. The same style sheet can of course also be used with many different content pages. In other words, content and presentation are decoupled and can therefore vary independently of each other.

Example 4

Not only is computer software decoupled from computer hardware, it is also decoupled from the data which it processes. For example, your word processor is decoupled from the documents you write with it: you can write many different documents with your word processor, and, within limits, you can use different word processors to edit a given document. The same scenario plays out with your MP3 player. It is decoupled from the music files you can play with it.

Example 5

A less concrete example, but one which is very important in computer science, is the notion that computer algorithms (sequences of steps to be followed in solving a particular type of problem) are decoupled from the data (stored in something called a data structure) over which they operate. This leads to the same sort of flexibility as we have seen in our other examples: the algorithm and the data can vary independently.

Example 6

Finally, a more familiar example is that user interfaces are decoupled from the underlying functionality which the interface accesses. For instance, if you have a bank account you can probably access it from an automated banking machine, from a web page, and perhaps also by telephone. The interface provides you (in your role as a client) with a set of capabilities (such as *get balance* and *pay bill*) which you can ask your bank account to perform. The user interface and the account can vary independently of each other: you can access your account using many different interfaces (web based or banking machine based), and each user interface can interact with many different bank accounts (your accounts, your neighbors' accounts, even Bill Gates' accounts).

The common thread running through these examples is decoupling of components. To truly appreciate the important role that decoupling plays in these examples, try to imagine what would happen if the systems were not decoupled. How useful is an MP3 player than can play only one song? Or an ATM that can only withdraw money from your neighbor's account? Or a web browser that can only display one web page?

Exercise

Come up with at least three more examples of decoupling that you are familiar with. These need not be computer-related examples, although they may be. In each case clearly spell out what the components are that are being decoupled, and what benefits the decoupling brings about.

Interfaces

We now focus on how to decouple two basic aspects of an object oriented software system: the set of messages which an object must be able to respond to, and the responses that a given object associates with those messages. An object's response to a message is defined in a method. We specify a set of methods, without a commitment to a specific response, by defining an *interface*. An interface provides a specification, which in programming-language terms means a set of capabilities that one component expects another component to supply. Some people call interfaces contracts, because like contracts, interfaces give the terms (specifications) that a class must adhere to.

While interfaces and classes are different in many ways, they are also similar in some ways. One way in which they are different is that interfaces only *specify* capabilities, while

classes *define* capabilities. A way in which they are the same is that both interfaces and classes define *types*. We will think of a type as a category of objects. We sometimes refer to types defined by interfaces as *interface types* and types defined by classes as *class types*. Wherever the syntax of Java requires a type we may use either an interface type or a class type.

Decoupling of components is achieved through the use of an interface. If one component requires a specific other component to work with, then those two components are tightly coupled: they are highly dependent on one another. On the other hand, if one component requires only another component as long as that other component satisfies some interface, then we can supply any such component and get the job done.

Example 7

Returning to our travel plans, using an interface we can model the domain the way we want to: we can model the abstract notion "airplane" without having define an airplane class. The interface lets us describe the capabilities an airplane must have, while also allowing us to define specific types of airplanes in distinct ways.

Example 8

If I want to cut my grass, I need to use a lawnmower. If I have my heart set on an environmentally friendly walk-behind reel mower, then that is all that I will be willing to work with: I am tightly coupled with the reel mower. If all I care about is getting the grass cut, then any lawnmower will do: an environmentally friendly walk-behind reel mower, a electric push mower, a gas-powered push mower or a gas-powered riding mower. If express what I need in this way I have specified required capabilities, not specific behaviors. This leads to a more flexible system.

Specifying requirements using a specific class name makes a system less flexible, less able to adapt to change, and therefore more brittle: if a required component changes, the system will no longer work, and there will be no suitable replacements (since a specific class was called for). Specifying requirements using interfaces focuses on function rather than form.

Example 9

Consider the power grid. It provides an interface between electrical appliances and power stations. A typical electrical appliance needs 110V of 60Hz AC power. It does not care how the power was generated: whether by a coal-fired plant, a nuclear plant, a wind-generator, a hydroelectric dam or a solar cell. Since appliances do not care about how the power is generated (procedural information) only that the power is generated (declarative information) we can provide the required power using many different types of power producing techniques. This makes the power system as a whole more robust because it is not dependent on any one type of plant, or any one type of fuel.

We see here again the benefit of decoupling components of a large system: flexibility. The power grid abstracts away from the mechanism used to generate the power. In the same way we want to build software systems so that components care about what functionality other components provide, not how they provide them.

Defining an Interface

How does one define an interface? In Java an interface definition consists of (surprise) a header and a body.

The Interface Header

The header of an interface is similar to the header for a class. One significant difference is that the keyword `interface` is used rather than `class` to identify the sort of element that is being defined:

```
public interface identifier
```

If we want to define an interface for the category of things which can be a host, we might define an interface named `IHost`. This demonstrates our common naming convention for interfaces: interface names start with the capital letter I. In our case we put the letter I in front of the name Host to form an appropriate interface name. The name `IHost` communicates that this is the interface for a Host. Aside from this special naming convention, the naming of interfaces follows the same naming conventions as for classes: the first letter is capitalized and subsequent words are capitalized. The header for the `IHost` interface therefore looks like this:

```
public interface IHost
```

Interface body

The body of an interface consists of specifications of capabilities. Since the point of an interface is to decouple specification from implementation, no capabilities are *defined* in an interface; they are only specified (or declared). The body of an interface therefore consists not of full method definitions, but of *method declarations* (sometimes called *method specifications*). A method declaration consists of a method header followed by a semicolon '**;**'. In order for a class to conform to the specification given by the interface, one must be able to invoke all of the methods given in the interface on an instance of the class. If this is the case, we say that the class *implements* the interface.

Returning to the idea of a "host" interface, capabilities which a host should be able to perform include: seating patrons and notifying a server that a table in his/her area has been seated. Just as with classes, an interface must be declared to be inside a particular package. Here is an example:

```
package chapter3.restaurant;

/**
 * An interface describing the capabilities
 * that a host must possess.
 *
 * @author alphonce
 *
 */
public interface IHost {
	/**
	 * This method, when invoked, should cause the patron to become
	 * seated at an available table, chosen by the host.
	 *
	 * @param patron
	 */
	public void seatPatron(IPatron patron);

	/**
	 * This method, when invoked, should communicate to the server at
	 * whose table a new patron was seated, that the table now needs to
	 * be waited upon.
	 *
	 * @param table
	 */
	public void notifyServerTableIsSeated(ITable table);
}
```

Any class which implements this interface must provide definitions for both of these methods.

An Interface as a Contract

An interface is spelling out a *contract* by which two or more components in a software system will communicate with each other. The use of a contract to ensure interchangeability of components is quite common. We see this in electrical outlets. In the early days of electricity lighting was the main application.[3] Electrical fixtures were hard-wired into homes, but light bulbs were changeable. When it was discovered that electricity could be used to power all manner of nifty appliances, their power came from connections made into light fixtures. This proved to be inconvenient. The electrical outlet provided an easy way to decouple the power source from the power consumer, without having to route power through a light fixture.

Today it is common to see different sorts of power outlets in homes, each kind accepting a different power plug. For example, a 110-volt power outlet will accept a two or three pronged 110-volt power plug, but will not accept a 220-volt power plug. Likewise, a 220-volt power outlet will only accept a 220-volt power plug, not a 110-volt power plug. Each plug provides a different interface. As long as the plug conforms to the interface it can be connected to the power source provided by the outlet. This standardization ensures that appliances are not damaged by plugging to an outlet providing power with characteristics different from that is expected.

The rationale for using interfaces in software is much the same. An interface ensures that communication between software components will be successful. Interface definitions provide method headers, which provide all the information necessary to ensure that the corresponding methods can be called: the name of the method, the arguments and their types, and the return type specification.

Implementing an Interface

The preceding chapters have laid the foundations for building executable models. We should now have an understanding of what an object is, how it is created (instantiated) from a class, and the rudiments of how it communicates with other objects via method calls. We have also explored how interfaces allow us to specify required capabilities. In a sense interfaces allow us to express expectations of capabilities – the capabilities which one object requires of other objects in order to be able to interact with them.

In our discussion of interfaces we learned how to write a method specification. A method header is the part of a method definition which tells us how we can interact with the method: it gives the name of the method, the arguments it expects when it is called, and what type of value (if any) that it returns to the caller. In this chapter we will learn how define a class which implements an interface by providing complete *method definitions* for the methods specified in the interface.

---

[3] For more information, see http://en.wikipedia.org/wiki/Electrical_outlet.

Building an executable model

Using an interface, we can specify the methods which are required of a component in our system. How do we actually build a component which satisfies the requirements expressed in the interface? We define a class which gives definitions all of the methods that are specified in the interface.

When we write a class which defines all the methods of an interface, we say that the class *implements* the interface. We can also say that the class *realizes* the interface. We indicate our intent to do this in the code by adding an "implements clause" to the header of the class which implements the interface. An implements clause consists of the `implements` keyword followed by an identifier:

```
implements identifier
```

Let us make this more concrete. Recall the `IHostess` interface from last chapter (shown below in the `chapter4.restaurant` package, but otherwise unchanged):

```
package chapter4.restaurant;

/**
 * An interface describing the capabilities
 * that a hostess must possess.
 */
public interface IHostess {
      /**
       * @param patron is the person to be seated
       */
      public void seatPatron(IPatron patron);

      /**
       * @param table is the table that is seated
       * @param waitress is to be notifed that table is seated,
       *        and ready to be served
       */
      public void notifyWaitressTableIsSeated(ITable table, IWaitress
waitress);
}
```

A class which implements this interface (let us call it `Hostess`) will have the following class header:

```
public class Hostess implements IHostess
```


Whenever a class is defined as implementing an interface, it must provide a full method definition for each method specified in the interface.[4] This means we need to write a method header and a method body for each method that appears in the interface. How do we know what the method header should be? The method specification in the interface gives us the required method header.[5] A method body consists of a (possibly empty) sequence of *statements* enclosed in curly braces, '`{`' and '`}`'. We will begin by defining methods with empty statement sequences. Such a minimal method body is simply:

```
{}
```

It is common to call a method with a minimal body a *method stub* or an *empty method*. When invoked, an empty method does nothing.

Since we can, at this point, only write empty methods, we are going to start by modeling a hostess which is quite lazy: a hostess which does nothing! We can define the methods `seatPatron` and `notifyWaitressTableIsSeated` each with a minimal method body as follows:

```
public void seatPatron(IPatron patron) {
}

public void notifyWaitressTableIsSeated(ITable table, IWaitress waitress) {
}
```

This is a commonly used approach to defining classes: we start by defining all the methods as stubs,[6] then iteratively refine the method definitions. At this point, the full class definition for our `Hostess` class is:

---

[4] We will learn later that this is not quite true; it is true for classes called *concrete* classes, but not for what are called *abstract* classes. For now all our classes are concrete.

[5] Recall that the body of an interface consists of a collection of method specifications, and that each such specification consists of a method header followed by a semicolon ';' The header of a method must be the same in an interface and in any class which implements it.

[6] This is called "stubbing out" a class.

```
package chapter4.restaurant;

/**
 * An implementation of the IHostess interface.
 * As it stands, it models a lazy hostess, who does not do anything.
 */
public class Hostess implements IHostess {

	/**
	 * A default constructor.
	 */
	public Hostess() {
	}

	/**
	 * @see IHostess#seatPatron(IPatron)
	 */
	public void seatPatron(IPatron patron) {
	}

	/**
	 * @see IHostess#notifyWaitressTableIsSeated(ITable, IWaitress)
	 */
	public void notifyWaitressTableIsSeated(ITable table, IWaitress
waitress) {
	}
}
```

To stub out a method with a non-void return type specification, we must return a value of the appropriate type. To indicate what value is being returned from a method, we use the keyword `return` followed by the value that is to be returned. Now that we know about object references and `null`, we can stub out a method with a non-void return type specification by writing it so that it returns the `null` value. For a method which is specified as:

```
public Water getWater();
```

we can write an implementing method stub as follows:

```
public Water getWater() {
    return null;
}
```


Exercise

Define an interface which contains two method specifications, one for a method whose return type specification is `void`, whose name is `turnOn`, and which has an empty formal parameter list, and another for a method whose return type specification is `void`, whose name is `turnOff`, and whose formal parameter list is also empty. Name the interface `ISimpleDevice`.

Exercise

Define an class which implements the interface `ISimpleDevice`. Define the required methods as stubs.

The Realization Relationship: Types and subtypes

In chapter 3 we mentioned that when we define an interface or a class we are actually extending Java's type system with a brand new type. Any type defined by an interface or a class can be used wherever a type is needed, e.g. as the return type of a method or the type of a parameter.

The relationship that exists between a class and an interface when a class implements the interface is called a *realization relationship*: the class *realizes* the interface. Something else happens too: when a class implements an interface there is a special sort of relationship that is set up between their corresponding types. The type defined by the class is a *subtype* of the type defined by the interface. Alternatively we can say that the type defined by the interface is a *supertype* of the type defined by the class.

Example 10

A person can play multiple roles in their life. One role a person can play is that of *police officer*. A police officer has capabilities like the following: (i) write a ticket, (ii) read suspect their rights, and (iii) arrest suspect. Another role a person can play is that of *parent*. A parent has capabilities along these lines: (a) read bedtime story, (b) give milk and cookies, and (c) give hugs.

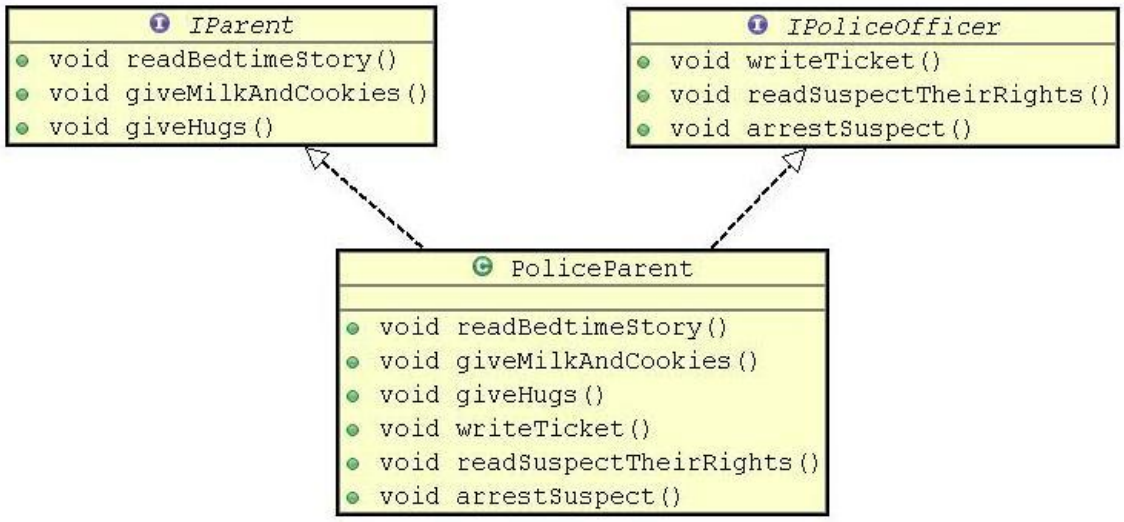
One and the same person can satisfy both roles, though typically they would do so at different times. Imagine what would happen otherwise – a bank robber might be given hugs to along with their milk and cookies, whereas the toddler might be read their rights before being given a ticket.

When we model a domain we want to capture those aspects which are relevant to solving a given problem. If the multiple roles that a person plays in life are relevant they must be included in a model. An interface is an appropriate tool to use in modeling roles.

Realization Relationship in UML

We can show the existence of a realization relationship between a class and an interface in a UML class diagram by drawing a relationship arc between the relevant class and interface. The realization relationship is shown by drawing a dashed line between the class and the interface, and by drawing a open arrowhead at the interface end of the line.

The following UML diagram shows two realization arcs. This diagram reflects what we said in the previous example:



There are two roles identified in the example; in the diagram we call them `IParent` and `IPoliceOfficer`. There is one class, `PoliceParent`, which implements each of these two interfaces. The diagram therefore shows two realization relationships, one between the class `PoliceParent` and the interface `IParent`, the other between the same class, `PoliceParent`, and the interface `IPoliceOfficer`.

Notice that it is possible for a class to implement more than one interface. In fact, it is permissible for a class to implement any number of interfaces, including zero.

Realization Relationship in Code

We now turn to the expression of the realization relationship in Java code. There are two basic things to remember when writing a realization relationship in Java code. The first is that the class which implements an interface must declare so in an implements clause in its header. The second is that the class must provide definitions for all the methods specified in the interface. For example, here is what a stubbed-out version of the PoliceParent class looks like:[7]

---

[7] Notice that there are many more comments in this example than we have seen before. There is really nothing new here, tough. We are still writing comments in comment-block form, its just that we've sprinkled comments in many more places. In fact, each method is annotated with a comment. The form of comment we're seeing here, in which the comment block starts with "/**" rather than just "/*", is a special kind of comment block

```
package chapter4.roles;

/**
 * @author alphonce
 *
 */
public class PoliceParent implements IParent, IPoliceOfficer {

	/** (non-Javadoc)
	 * @see chapter4.roles.IParent#readBedtimeStory()
	 */
	public void readBedtimeStory() {
	}

	/** (non-Javadoc)
	 * @see chapter4.roles.IParent#giveMilkAndCookies()
	 */
	public void giveMilkAndCookies() {
	}

	/** (non-Javadoc)
	 * @see chapter4.roles.IParent#giveHugs()
	 */
	public void giveHugs() {
	}

	/** (non-Javadoc)
	 * @see chapter4.roles.IPoliceOfficer#writeTicket()
	 */
	public void writeTicket() {
	}

	/** (non-Javadoc)
	 * @see chapter4.roles.IPoliceOfficer#readSuspectTheirRights()
	 */
	public void readSuspectTheirRights() {
	}

	/** (non-Javadoc)
	 * @see chapter4.roles.IPoliceOfficer#arrestSuspect()
	 */
	public void arrestSuspect() {
	}
}
```

What does it mean for one type to be a subtype of another type? It means that an object which belongs to the subtype also belong to the supertype, and can play the role of the supertype (since it is guaranteed to satisfy the specifications given in the supertype).

---

called Javadoc comment block. Within such a block we can write comments with special tags, like @author or @see. These tags are used by a special piece of software, called javadoc, to generate comments in a form that is easily readable on the web. We will discuss these kinds of comments in more detail later.

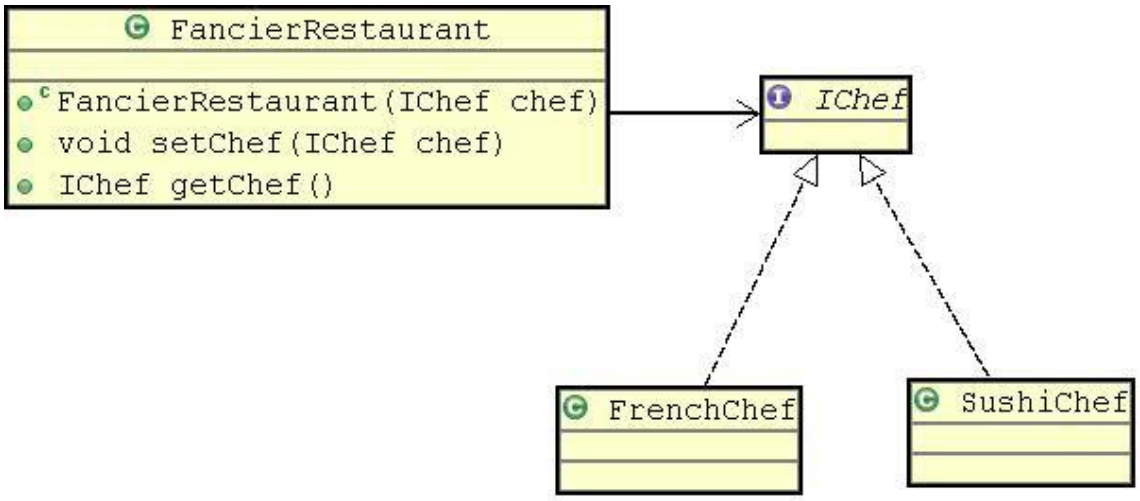
> Referring to our example, any object of type `PoliceParent` is also of the types `IParent` and `IPoliceOfficer`. This means that if a method requires an object of type `IPoliceOfficer` to be passed in, any instance of the `PoliceParent` type can be passed along in the method call.

This is quite an important idea, which we will return to frequently as we see some more ways in which the supertype-subtype relationship can be established.

Interfaces and flexibility

A final observation regarding the fancy restaurant. As it stands the `FancyRestaurant` class requires that its chef be of the type `Chef`. Imagine a scenario in which there are many different type of chefs, such a French chefs and sushi chefs. We might not necessarily want to make the `FancyRestaurant` class dependent on a specific type of chef. To make our model flexible we could define an *interface* for chefs, and model the specific types of chefs as implementations of the common interface.

We close this chapter by showing the UML class diagram and implementation of a more flexible system, one which makes use of an interface, `IChef`, to represent the notion of a generic chef. First the UML diagram:



Notice that the association arc is drawn between the `FancierRestaurant` class and the `IChef` interface, even though in a running program the `_chef` variable will refer to an instance of a class which implements `IChef`. The code for the `FancierRestaurant` class is shown below; it includes a composition relationship with the `Kitchen` class.[8]

[8] Remember that a UML class diagram shows only as much detail as is deemed necessary. In the diagram just shown in the text we wanted to draw your attention to the relationships that exist precisely amongst the three classes and one interface shown. Including the `Kitchen` class in the diagram we felt would be a distraction, so we chose not to.


```
package chapter4.restaurant;

/**
 * @author alphonce
 *
 * This class models that a FancierRestaurant not only must
 * be composed of a Kitchen, it must know a generic IChef; the
 * relationship with the IChef is modeled using an association
 * relationship.
 *
 * Since IChef is an interface, and interfaces cannot be
 * instantiated, the _chef variable must refer to an instance
 * of a class which implements IChef.
 */
public class FancierRestaurant {

	/**
	 * The _kitchen of a FancierRestaurant object.  The _kitchen is
	 * a **part** of a FancierRestaurant.
	 */
	private Kitchen _kitchen;

	/**
	 * The _chef of a FancierRestaurant object.  The _chef is known
	 * to a FancierRestaurant.
	 */
	private IChef _chef;

	/**
	 * @param chef (of the FancierRestaurant)
	 */
	public FancierRestaurant(IChef chef) {
		_kitchen = new Kitchen();
		_chef = chef;
	}

	/**
	 * Sets the chef of this FancierRestaurant object.
	 * @param chef - an instance of a class which implements the IChef
	 *               interface
	 */
	public void setChef(IChef chef) {
		_chef = chef;
	}

	/**
	 * @return the current chef of this FancierRestaurant object.
	 */
	public IChef getChef() {
		return _chef;
	}
}
```

Notice that the type of the parameter in the mutator method, as well as the return type of the accessor method, is written as `IChef`.

Summary

In software development, decoupling is a desirable feature. One mechanism available to us to assist in decoupling is an interface. An interface allows us to model a collection of capabilities that we would like an object to possess.

Capabilities are the actions we would like an object to perform. In a class definition, capabilities are formally defined by methods. A method consists of a method header and a method body. Inside a method header, we indicate the visibility, return type, name of the method, and the parameters the method takes. The return type tells us the type of information that will be returned when the method is called. The parameters tell us what additional information is needed to perform the activities of the method. The method body will give the sequence of steps that must be executed in order for the activities of the capability to be performed.

As we saw with classes, simply writing the definition of a method is not enough. The method must be called or invoked in order for the action defined by the method to be performed. Calling a method is achieved through the use of the dot operator.

Syntactically an interface is a collection of method headers. Interfaces, like classes, allow us to define types in our programs. In order to take full advantage of interfaces, classes must meet the specification laid out in an interface.

In this chapter we saw how to define a class which implements an interface. We also learned that one thing an interface can model is the role an object plays. We learned that in order to implement an interface a class must provide definitions for the methods in the interface, which are only specified (not defined) there. To begin with, we can implement methods by stubbing out their method bodies. Methods that have a void return type will have empty method bodies. Methods that have a non-void return type can return the value `null`.

The notions of supertype and subtype were also touched upon. Each class and each interface defines a type. The class type which implements an interface type is a subtype of that interface type. Conversely, the interface type is the supertype of that class type.

Finally, we discussed the flexibility which we can engineer into a software system by using interfaces rather than classes to specify requirements.

Case Study

Stay tuned.

Chapter Wrap-Up

Learning Objectives

At the end of this chapter students should be able to:

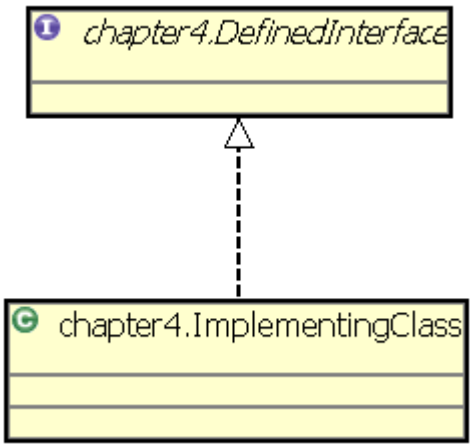
- Build an executable model from a given interface – i.e. define a class implementing the interface.
- Understand how an interface decouples declarative knowledge from procedural knowledge.
- Declare (or specify) a method, by giving its method header and an empty method body.

- Understand that both classes and interfaces define types.
- Define an interface consisting only of method declarations.

Relationships Covered

Realization

Informal Name: realizes, implements
First introduced in: Chapter 4 – page 12
UML representation:



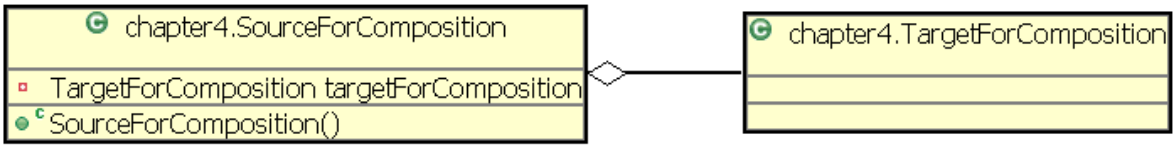
In code: (from above diagram)

```
package chapter4;

public class ImplementingClass implements DefinedInterface {

}
```

Note: There is no code inside the class `DefinedInterface` to indicate the relationship

Composition

Informal Name: *has-a*
First introduced in: Chapter 4 – page **Error! Bookmark not defined.**
UML representation:



In code: (from above diagram)

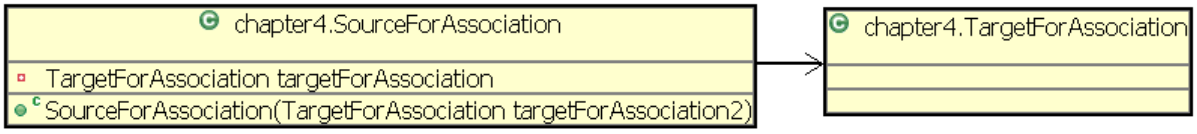
```
package chapter4;

public class SourceForComposition {
```

```
	private TargetForComposition _targetForComposition;
	public SourceForComposition() {
		_targetForComposition = new TargetForComposition();
	}
}
```

Note: There is no code inside the class `TargetForComposition` to indicate the relationship

Association

Informal Name: *knows-a*
First introduced in: Chapter 4 – page **Error! Bookmark not defined.**
UML representation:



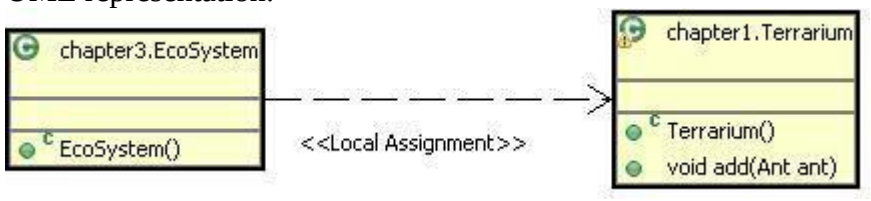
In code: (from above diagram)

```
package chapter4;

public class SourceForAssociation {
	private TargetForAssociation _targetForAssociation;

	public SourceForAssociation(TargetForAssociation tfa) {
		_targetForAssociation = tfa;
	}
}
```

Note: There is no code inside the class `TargetForAssocation` to indicate the relationship

Local Variable Dependency

Informal Name: *uses-a*
First introduced in: Chapter 3 – page **Error! Bookmark not defined.**
UML representation:



In code: (from above diagram)

```
package chapter3;
```

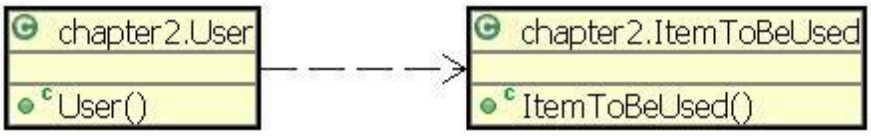
```
public class EcoSystem {
	public EcoSystem() {
		chapter1.Terrarium _terrarium = new chapter1.Terrarium();
	}
}
```

Note: There is no code inside the class `chapter1.Terrarium` to indicate the relationship.

Instantiation Dependency
Informal Name: *uses-a*

First introduced in: Chapter 2 – page **Error! Bookmark not defined.**

In UML representation:



In code: (from above diagram)

```
package chapter2;

public class User {
	public User() {
		new ItemToBeUsed();
	}
}
```

Note: There is no code inside the class `ItemToBeUsed` to indicate the relationship.

Keywords Covered

`implements` (page 9)

`interface` (page 6)

`return` (page 11)

`Previously:`

`class` (**Error! Bookmark not defined.**)

`new` (**Error! Bookmark not defined.**)

`null` (**Error! Bookmark not defined.**)

`package` (**Error! Bookmark not defined.**)

`private` (page **Error! Bookmark not defined.**)

`public` (**Error! Bookmark not defined.**)

`void` (page **Error! Bookmark not defined.**)

Naming Conventions Covered

Classes – Begin with an upper case letter. The first letter of each subsequent word in class name is capitalized.

Instance variables – Begin with an underscore character. First letter of each subsequent word in instance variable name is capitalized.

Interfaces – Begin with an upper case letter. The first letter of each subsequent word in interface name is capitalized. Interface names are usually preceded by the capital letter I.

Local variables – Begin with a lower case letter. First letter of each subsequent word in variable name is capitalized.

Methods – Begin with a lower case letter. The first letter of each subsequent word in method name is capitalized.

Packages – Are written entirely in lowercase. Package names in nested packages are separated by dots.

Vocabulary & Programming Terms Covered

New this Chapter:

implement an interface (9)

interface (4)

interface type (5)

method stub (10)

empty method (10)

subtype (12)

supertype (12)

Previously Covered:

abstraction (**Error! Bookmark not defined.**)

access control modifier (**Error! Bookmark not defined.**)

accessor (**Error! Bookmark not defined.**)

assignment operator (**Error! Bookmark not defined.**)

bytecode (**Error! Bookmark not defined.**)

calling [a constructor] (**Error! Bookmark not defined.**)

problem domain (**Error! Bookmark not defined.**)

process (**Error! Bookmark not defined.**)

property (**Error! Bookmark not defined.**)

property value (**Error! Bookmark not defined.**)

reference (**Error! Bookmark not defined.**)

relationship (**Error! Bookmark not defined.**, **Error! Bookmark not defined.**)

return type specification (**Error! Bookmark not defined.**)

scope (**Error! Bookmark not defined.**)

semantics (**Error! Bookmark not defined.**)

service (**Error! Bookmark not defined.**)

source code (**Error! Bookmark not defined.**)

state (of an object) (**Error! Bookmark not defined.**)

statement (**Error! Bookmark not defined.**)

syntax (**Error! Bookmark not defined.**)

top-level object (**Error! Bookmark not defined.**)

type(**Error! Bookmark not defined.**)

UML [Unified Modeling Language] (**Error! Bookmark not defined.**)

uncomputability (**Error! Bookmark not defined.**)

variable (**Error! Bookmark not defined.**)

void (**Error! Bookmark not defined.**)

Design Patterns Covered

We have not covered any design patterns so far in this text.

