Chapter 6
Polymorphism

Pedagogic Goals

- Introduce the notion that polymorphism allows us to model inherent differences in the implementation of identical capabilities.
- Introduce the notion that polymorphism is a form of selection, based on the type of an object.
- Present design patterns as structuring mechanisms for code which allow us to build software systems possessing certain desirable characteristics.

Introduction

In the real world we often come across systems with the same capabilities which nonetheless differ quite a bit in the realization of those capabilities.

Example 1

Four-legged mammals have two different forms of walk. Some four-legged mammals, like dogs, walk by moving their left-rear and right-front legs together, and right-rear and left-front legs together. Others, like giraffes, walk by moving both legs on one side together. Both dogs and giraffes have the capability to walk, but they do so in quite different ways.
http://www.earthlife.net/mammals/locomotion.html

Because dogs and giraffes both walk but they walk differently, we say that dogs and giraffes are *polymorphic* in their walking capability. In this chapter we will explore many different systems which exhibit polymorphic behavior naturally. We will also learn how to build object oriented models which exploit the natural polymorphism of systems to yield flexible and easily extensible software.

Polymorphism is natural

Polymorphism refers the ability of different types of things to react differently to the same stimuli. In our object oriented systems, this means that different types of objects can respond differently to the same method call.

Polymorphism is a very natural, everyday phenomenon. We unconsciously make use of polymorphism several times a day. Before we look more closely at what polymorphism looks like in an executable model, let us consider several more examples of polymorphic behavior.

Example 2

One of the authors is a bit of an old-timer. He has a cassette deck hooked up to his stereo. He also has a CD player hooked up. The other author, who is far more hip, has an MP3 player. All of these devices can play music. They all have a "play" functionality. However, how they play their music is quite different. Playing is polymorphic!

The tape deck uses a magnetic head, called the read head, to read information from a magnetic tape which is passing below the head. The tape is pulled along from one reel of the cassette tape to another reel in the cassette by a little motor.

The CD player uses an optical read head to read digitally encoded information from the pitted surface of a CD as a stream of bits. The CD is rotated about its center by a motor. When the surface of the CD passes under the read head, laser light is reflected off the surface of the CD – depending on how the light reflects off the surface the CD player is able to distinguish 0's from 1's and thereby read each bit of information from the CD.

The MP3 player has no moving parts. It reads similar kinds of digital information from its memory as the CD player extracts from the surface of a CD.

Even though tape decks, CD players and MP3 players interpret a "play" instruction very differently, they can all play the same music. This is polymorphism at work.

Example 3

Once upon a time printing terminals with dot matrix print heads were the latest thing. Nowadays inkjet printers and laser printers are common. A document can be printed on any of these kinds of printers, but the means of forming an image on a piece of paper is quite different.

The Digital Equipment Corporation DEC-Writer was a printing terminal with a single dot-matrix print head. A dot-matrix print head has several pins which can be pushed out against an inked ribbon, which is pressed against a piece of paper, thereby making an impression on the paper. Characters were made by different patterns of pin impressions, usually on a 5 by 7 grid.

The inkjet printer and the laser printer use two very different printing technologies. Printing with an inkjet printer works by spraying a small amount of ink from a nozzle onto the page. Printing with a laser printer works by charging points on a rotating drum; charged regions pick up toner as the drum rotates, which is then melted onto the paper by a high-temperature fuser.

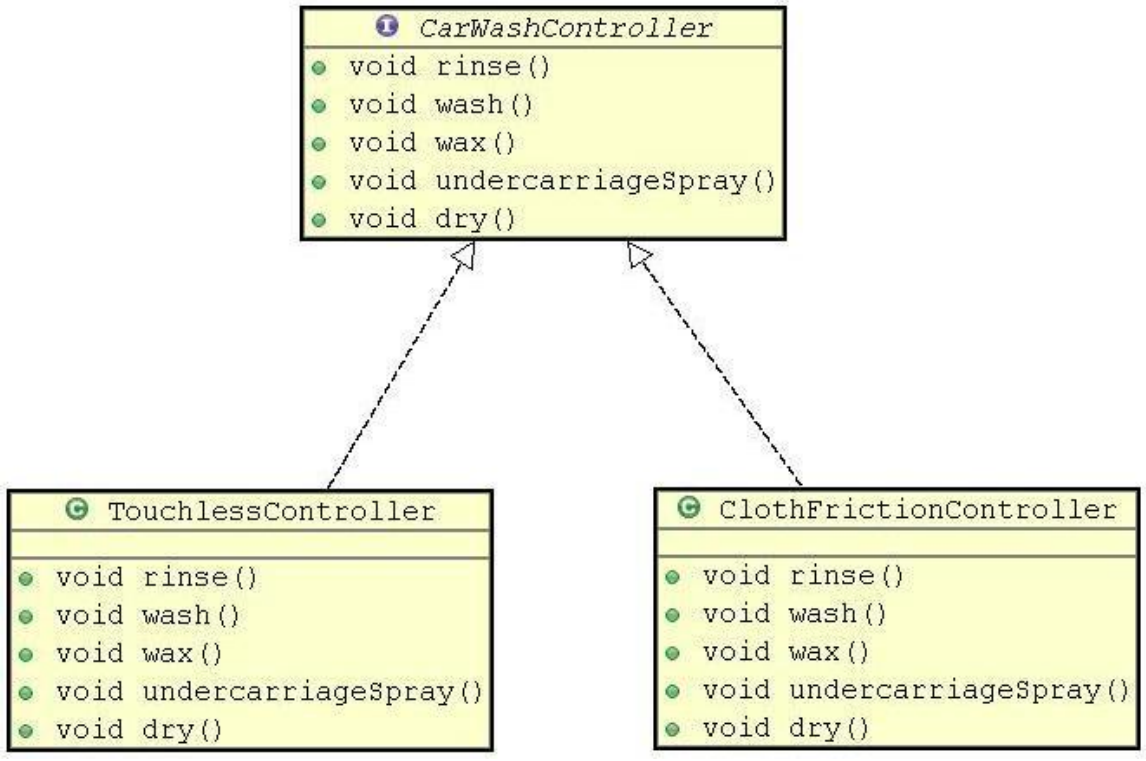
In all cases the printer "prints" a page of information, but in a decidedly different way. The software which is sending information to the printer need not know what printer the information is being sent to – the printer knows how to print in its own polymorphic way.

When we model systems in object oriented terms we want to be able to capture these aspects of systems. These are examples of *polymorphic* behaviors. In object oriented terms we model polymorphic behaviors by defining methods polymorphically. One way a method can be defined polymorphically occurs when different classes, all of which implement the same interface, provide distinct definitions for a method specified in the interface.

Example 4

There are two basic kinds of automatic car washes: mechanical (apparently called "cloth-friction") and touchless. A mechanical car wash uses brushes of some sort to scrub the outside of the car, while a touchless car wash uses powerful water sprays and strong chemicals to clean the car. Many of these systems are under computer control. As far as the client of the carwash is concerned the possible operations that the carwash can perform are the same, such as rinse, wash, wax, undercarriage spray and dry. The way the computer controls need to implement these operations is quite different. This is yet another example of polymorphic behavior.[1]

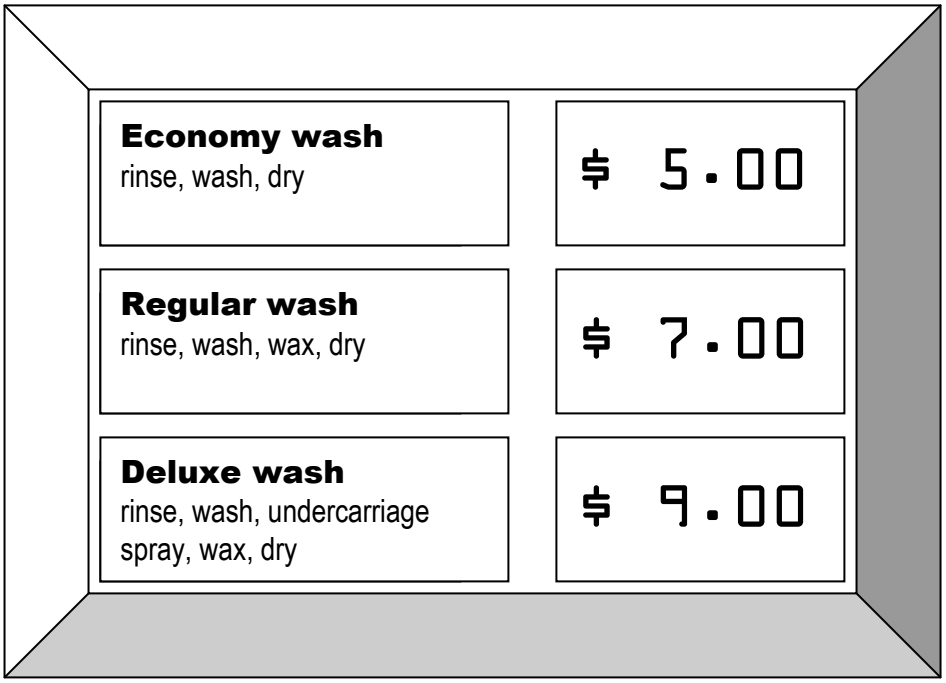
The basic framework for implementing the car wash control could be something like the following:



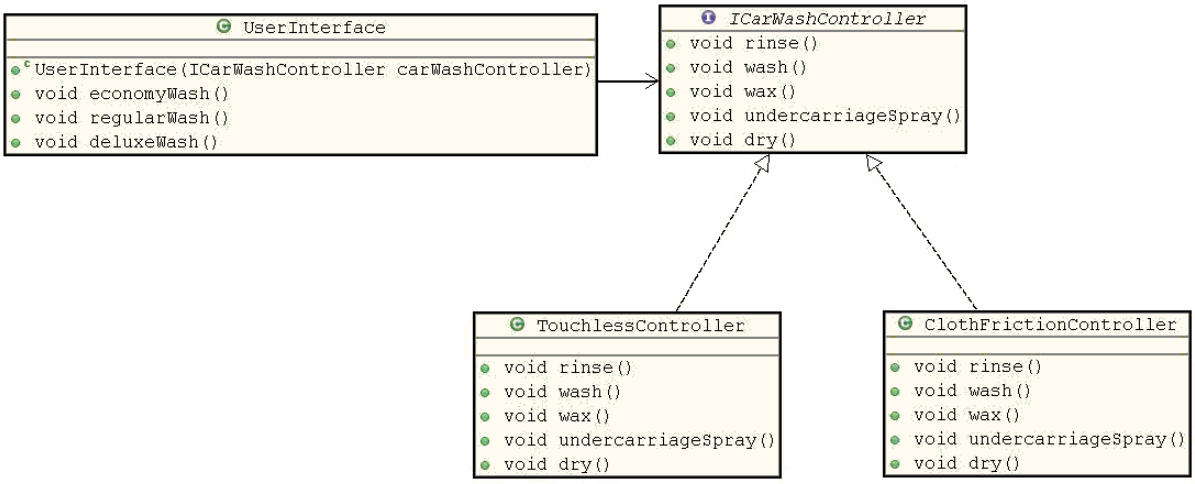
[1] For more information about car washes, see

- en.wikipedia.org/wiki/Carwash
- auto.howstuffworks.com/car-wash.htm

Let us suppose that in a typical car wash installation there is a user interface which lets a person select the type of wash that they want for their car. The user interface may have buttons like the ones shown below:



The software that runs the user interface records what kind of wash a person selects, and communicates this to the software that controls the running of the car wash. In order to enable this sort of communication the user interface must declare an instance variable of type `CarWashController`.



Notice that the `UserInterface` does not know (or care) what *actual* type of `CarWashController` it is communicating with. As long as the controller object conforms to the interface specifications, the user interface object can communicate with it just fine.

This means that the user interface manufacturer can build a single user interface that talks just as well to a `TouchlessController` as to a `ClothFrictionController`. In this way the `UserInterface` and the actual controller implementations are *decoupled*.

Imagine what the system would have to look like *without* the interface. In this case the `UserInterface` class could not take advantage of the interface type – its constructor would instead have to be defined to take an object of a *class type* rather than an *interface type*, as in:

```
public UserInterface(ClothFrictionController controller) {...}
```

Of course, if the `UserInterface` is defined this way it can only work with a `ClothFrictionController` and not any other type of controller. With this design, in order to accommodate two types of controller we would have to define another type of user interface class, whose constructor takes a parameter of the appropriate type.

Now follow this to its logical conclusion. Without an interface the class which provides the functionality of the user interface needs to be replicated for each type of controller we have in the system. Think about this: every time a new controller is added, *we also have to add a new user interface class!* Not even thinking about the fact that these classes will all do essentially the same thing, this means that instead of adding one new class, we have to add two!

If we use an interface, it allows us to *isolate* (decouple) the part of the system that needs to change when we add a new controller, making extension of the existing system much easier and less costly.[2] Moreover, when we use an interface we need to write the code for the `UserInterface` only once. If the `UserInterface` code needs to be changed at some point in the future this means that it only needs to be changed once, not once per controller that is in the system! In other words, employing an interface allows us to structure the overall system so that a small change in the requirements of the system results in a small change to the system. Without using an interface a small change in the requirements of the system results in a large change to the system.

There is an additional advantage to structuring software in this way: not only is it easy for the user interface object to communicate with either of the existing controller implementations, it can also communicate with any *future* car wash controller, as long as it implements the `CarWashController` interface! The beauty of decoupling in is that no matter what new car wash controllers are dreamt up in the future, `UserInterface` class will never need to be changed, as it is insulated from any changes in the software past the `CarWashController` interface.

Types and subtypes revisited

We know from discussions in previous chapters that the realization relationship between a class and an interface introduces a subtyping relationship between the type of the class and

[2] This sort of analysis, which essentially breaks a system down into the parts that can change and the parts that stay the same, is called "commonality-variability analysis" and also "variant-invariant decomposition". The basic idea is that we want to decouple variant parts of a system from invariant parts of a system, so that changing one does not necessitate changing the other.

the type of the interface.[3] It is worth emphasizing again that an object of a subtype can used whenever an object of the supertype is expected. Consider the following snippet of code.

```
public class UserInterface {
    private CarWashController _controller;

    public UserInterface() {
        _controller = new ClothFrictionController();
    }
}
```

In this code the instance variable `_controller` is declared to be of type `CarWashController`, yet an object of type `ClothFrictionController` is assigned to it. This is perfectly acceptable, but only because the type `ClothFrictionController` is a subtype of the type `CarWashController`.

A note about terminology. We make a distinction between the type of a variable (which is the type specified in the variable's declaration) and the type of the object that the variable refers to (the class from which the object is instantiated). The former we call the *declared type* of the variable, while the latter is referred to as the *actual type* of the object. In the code snippet above we would say that the declared type of `_controller` is `CarWashController`, and that the actual type of the object assigned to `_controller` is `ClothFrictionController`. We will, by slight abuse of terminology, sometimes refer to the actual type of a variable – what we mean in this case is the actual type of the object that the variable currently refers to.

Let us now look at what the code for the `UserInterface` class might look like in a bit more detail.

```
package chapter5.carwash;

/**
 * @author alphonce
 */
public class UserInterface {

      private ICarWashController _carWashController;

      /**
       * @param carWashController
       */
      public UserInterface(ICarWashController carWashController) {
            _carWashController = carWashController;
      }

      /**
       * Instructs a CarWashController to
```

[3] This idea that subtyping should imply substitutability is due to Barbara Liskov and Jeanette Wing, "A Behavioral Notion of Subtyping", *ACM Transactions on Progamming Languages and Systems*, volume 16, number 6. It is called the *Liskov Substitution Principle*.

```
         *     1) rinse
         *     2) wash
         *     3) dry
         */
        public void economyWash() {
              _carWashController.rinse();
              _carWashController.wash();
              _carWashController.dry();
        }

        /**
         * Instructs a CarWashController to
         *     1) rinse
         *     2) wash
         *     3) wax
         *     4) dry
         */
        public void regularWash() {
              _carWashController.rinse();
              _carWashController.wash();
              _carWashController.wax();
              _carWashController.dry();
        }

        /**
         * Instructs a CarWashController to
         *     1) rinse
         *     2) wash
         *     3) undercarriage spray
         *     4) wax
         *     5) dry
         */
        public void deluxeWash() {
              _carWashController.rinse();
              _carWashController.wash();
              _carWashController.undercarriageSpray();
              _carWashController.wax();
              _carWashController.dry();
        }
}
```

Consider the method `economyWash`. This method's body consists of a sequence of method calls. The method calls are made to the object referred to by the variable `_carWashController`. Do we know, by looking at this code, whether this variable refers to a `ClothFrictionController` object or a `TouchlessController` object? In fact, we do not. Does this matter? No! Regardless of what type of object `_carWashController` refers to, the method calls will work. The actual implementation of those methods may well be different in the two classes `ClothFrictionController` and `TouchlessController`. That is, in fact, the point of polymorphism. The interface tells us what methods are callable, whereas it is up to the implementing classes to define how to respond to those method calls.

Design Patterns: A Software Design Interlude

Let us now step back a bit and consider the software construction foundation which we have laid down. A trait of a good software developer (or more generally of a good problem solver) is being able to use their tools in effective ways. In this section we introduce some good ways to solve problems; they are called *design patterns*. Design patterns are what are known as *best-practices solutions to recurring problems*. Design patterns are mined from existing software, and are recognized as being the *best* solutions to common problems.

How can we measure the goodness of a software solution? Goodness is not measured in absolute terms, but is measured relative to one's needs in a given situation. For instance, a "quick-and-dirty" solution may be just what is called for in a proof-of-concept situation, where we want to show that a particular solution is feasible. Such a solution may be completely unacceptable in a production environment where things such as correctness, robustness and security are paramount.

We will measure goodness according to the following criteria, since these are generally regarded as desirable characteristics of software:[4]

- Correctness – The software we build must perform according to the requirements of the customer, as expressed in the software specifications.
- Robustness – The software must be able to gracefully handle invalid input, or when expected resources (such as memory) are unavailable.
- Scalability – The software must be able to handle large tasks as well as small tasks.
- Flexibility – The software must be able to adapt easily to new operating environments.
- Extensibility – The software must be easily extended to cover new, hitherto unforeseen situations.
- Maintainability – The software must be well-documented and constructed in a manner conducive to making adjustments to fix behavior which is not quite right.[5]
- Dynamicity – The software must be dynamic: it must be designed and structured in such a way that choices about program behavior are not arbitrarily fixed prior to runtime. There are cases in which program behaviors should be fixed prior to runtime, and not be permitted to vary, but in many cases the software is more desirable if it permits runtime variation.

In the following subsections we will explore an example system and see how design patterns help us to achieve some of these goals.

---

[4] We do not mean to imply that these are the only criteria according to which software quality should be judged. Clearly missing from this list is security. Security of software and the data it processes is very important, yet a proper discussion of security issues is beyond the scope of this text.

[5] The cause of the problem is immaterial: whether the specifications were incorrect to start with, or the specifications were correct but were incorrectly implemented, or the specifications were correct and were correctly implemented, but the customer's requirements changed, the functionality of the software must be adjusted for it to continue working acceptably in its current operating conditions.

A real-world application[6]

At some point in time you have probably come across an interactive program guide (IPG) on a TV. An interactive program guide (IPG) allows a user to browse television (cable/satellite) content in various ways, such as by channel, title, timeslot, and genre. Some systems provide access to weather forecasts. It is also possible to use the IPG to set subtitle or closed captioning options. To control the IPG a user presses keys on a remote control. The remote control typically has a small number of buttons used for navigation and selection. Depending on the current state of the IPG system, different things might happen when a given button is pressed.

For example, selecting a program to watch in the normal TV mode will switch to the indicated channel. However, in pay-per-view (PPV) mode some additional level of confirmation is required, so that a user does not accidentally incur a charge for a program they do not wish to pay for.

Similar systems are used in hotels to present guests with various kinds of information. For example, hotel systems allow guests to order things as diverse as movies and room service. They typically also allow guests to view their hotel bill on-screen and also to check out.

A real-world software design

Let us now explore, from a fairly high-level perspective, some of the design issues which crop up in this application.[7] Among the many patterns incorporated in this example are Iterator, State, Command, Null Object, and Singleton. The roles these patterns play and the issues they are meant to address, are discussed below. Please note that you are not expected to completely understand the role of these patterns in the software just yet. Rather, this section is intended to give you a sense of the breadth of patterns being used in software, why the patterns are used, and how use of patterns supports the construction of good software (good according to the metrics given previously).

Iterator Pattern

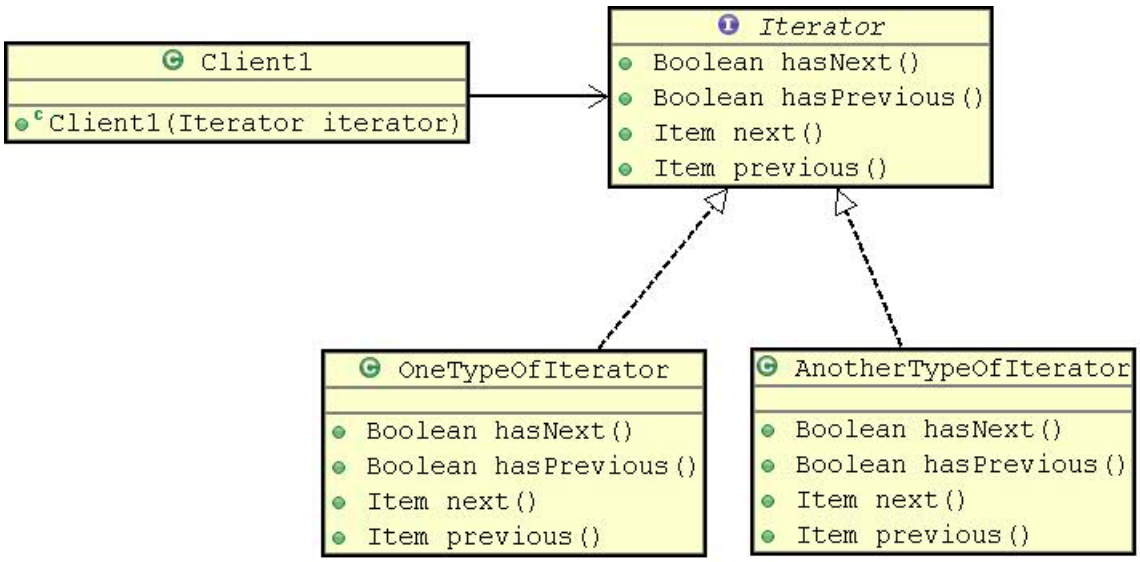
The IPG system keeps track of many different lists, such as lists of channels, lists of programs, and lists of genres. One of the things that a user of the system can do is browse from one item in a list to another, either forwards or backwards. An iterator is an object which keeps track of a position in a list; in the case of the IPG system it keeps track of the current list element that the user has chosen. The notion of a traversal across a list is captured in a pattern called the *iterator pattern*. One of the nifty thing about the iterator pattern is that it abstracts the notion of an iteration and separates it from the structure of the list it is traversing. This means that the IPG system is *decoupled* from any specific list that is being used to store channel, program, genre or other similar information. A significant benefit of

---

[6] Much of the background for the discussion in this section is drawn from a presentation at the *Third "Killer Examples" for Design Patterns and Objects First* workshop, by A. Sterkin, of NDS Technologies, a manufacturer of IPGs for many content providers worldwide. The discussion is not entirely faithful to the workshop presentation, however – it has been tailored to meet the needs of this text.

[7] This example is especially interesting because it is a real-world example combining a large number of *design patterns*, which nonetheless is understandable because the patterns address issues which are in your everyday experience. Oftentimes patterns in real-world software systems are less concrete.

this is that the IPG system will work with any collection of data, as long as it is *iterable*.



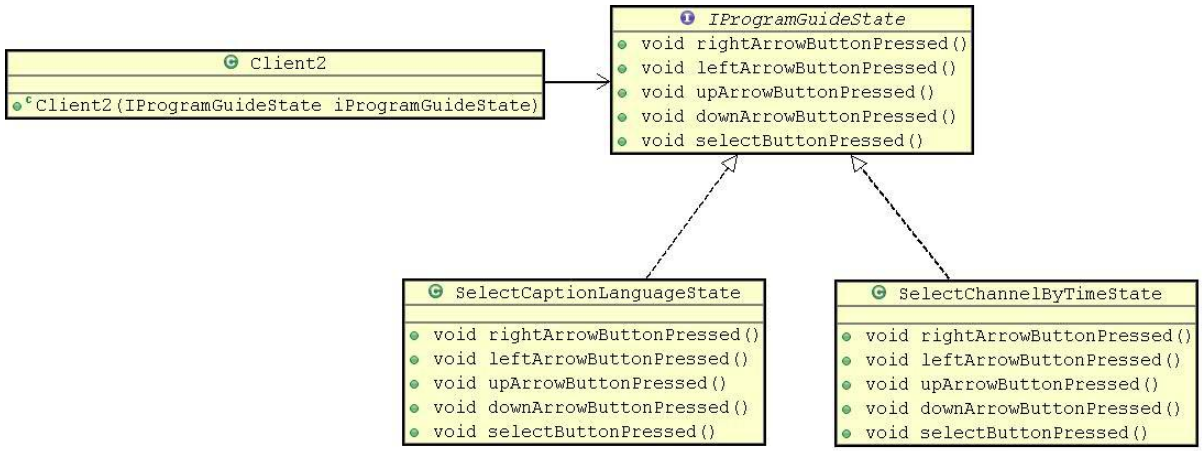
Polymorphsim is at work in the iterator pattern because the code using an iterator, the `Client1` in the diagram above, is decoupled from any specific iterator. In this way the actual iterator object that the `Client1` is using at any given point in time may well be different from the actual iterator object being used at a different point in time. The iterators, while they implement the same interface, define the typical iterator capabilities in different ways.

State Pattern

If you think about an IPG system, you will likely notice that the behavior of the system depends on its current state. When the IPG is in normal channel-browsing mode, the up and down buttons have one meaning (they affect which channel is selected) whereas when the IPG is in subtitle mode these buttons control whether subtitles are on or off. In other words, the behavior of an IPG system is governed by the particular state that it is in. We say that the system is *state-based*.

It is not simply that the behavior of just one or two buttons changes as the state of the system changes. Indeed, the behavior associated with most of the buttons on the controller change together as the state of the IPG changes. This is wholesale change in behavior is easily modeled using a *state pattern*. The pattern ensures that state-based behavior of the IPG system is coherent and consistent, by guaranteeing that the behavior off all the relevant buttons is changed all at once.. A class diagram of a typical state pattern implementation might look as

follows:

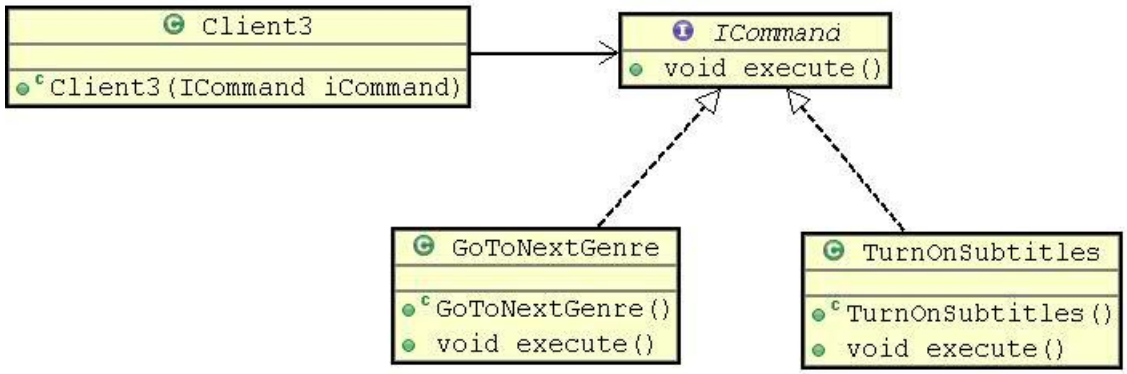


Using the state pattern in this example helps to ensure robustness: the program guide system itself has an `IProgramGuideState`, which encapsulates all the necessary behavior for the appropriate mode of operation.

Polymorphsim is at work in the state pattern because the state-based client, `Client2` in the diagram above, is decoupled from its specific state. The client knows a state, but it does not know which specific state it is in. As in the case of the iterator, the actual state object that `Client2` is using at any given point in time may well be different from the actual state object it is using at a different point in time. The states, while they implement the same interface, define the typical state capabilities in different ways.

Command Pattern

The behaviors that are triggered when various buttons on the remote control are pressed can be "objectified", or encapsulated as objects. The advantage of doing this is that it becomes easy to do things like keep track of what "commands" have been carried out, undo commands. It is also easy to attach the same behavior to buttons in different states without having to write the behavior more than once: the command object representing the behavior can be shared. A typical *command pattern* is shown in the diagram below.



In the IPG system an added benefit of using this pattern is that because the behaviors are objectified as command objects, the system retains the flexibility to easily accomodate new menus with new features. The keyboard layout can also be changed quite easily, without having to rewrite large amounts of code – command objects can be dynamically rebound to different keys.

Polymorphsim is at work again in the command pattern. The client code using a command, `Client3` in the diagram above, is decoupled from any specific command. The client knows a command, but it does not know which specific command it has access to. The actual command object that `Client3` is using at any given point can vary. The commands, while they implement the same interface, define the typical command capabilities in potentially different ways.

In each of these three patterns polymorphism allows the client to be decoupled from the concrete classes which realize a relevant interface. This is a recurring theme in many design patterns.

Null Object Pattern

In some states pressing certain buttons should not trigger any particular effect. In these cases there is an appropriate command that we can use, the so-called *null command*. A null command satisfies the interface for the command, but provides a minimal implementation of the methods required by the interface. In our example, the implementation of a null command for the `ICommand` is:

```
package chapter5.ipg;

/**
 * @author alphonce
 *
 */
public class NullCommand implements ICommand {

	/**
	 * Constructor for this class.
	 */
	private NullCommand() {}

	/**
	 * @see chapter5.ipg.IRightArrowButtonStrategy#buttonPressed()
	 */
	public void execute() {
		// DO NOTHING - THIS IS THE NULL COMMAND.
	}

}
```

The null command pattern is a specific example of a more general pattern, called the *null object pattern*. The null object pattern is useful whenever there is a need to model the absence of specific behavior. Null objects show up in many different situations, and simplify the design of systems because the null object case is handled just like any other case: with an object of the appropriate type injected into the system. This helps maintain the robustness of

the system: the same mechanisms are used to handle the typical and the exceptional cases, rather than special-purpose code.

This pattern can be used in conjunction with many other patterns, to indicate a do-nothing case. For example, in a state pattern a null state may well be defined as the initial state of a system. A null command is useful in the context of the IPG system in case there is no particular behavior associated with a given key on the remote control in some IPG mode: the correct behavior if the user presses this key is to do nothing!

Singleton Pattern

Since a null object (typically) has no internal state, there is no point in having more than one instance of a null object class existing at runtime. A null object class is therefore a perfect candidate for the application of another useful pattern, the *singleton pattern*. In the singleton pattern, the class to which it is applied takes responsibility for instantiating itself, making available a reference to the single instance this exists. This pattern can be implemented in a number of different ways; we have enough machinery in Java built up to show one implementation, using a *static* and *final* variable. Unlike an instance variable, a *static* variable is associated with the class, not any particular instance of the class. A static variable is accessed via the name of the class, as is shown in the example below. A *final* variable can be initialized, but cannot be assigned to once initialized. Lastly, in this implementation of the singleton pattern the constructor for the class is declared private, so that it may be invoked only within the body of the class definition. This allows the class to guarantee that exactly one instance of the class is created.

```
package chapter5.ipg;

/**
 * @author alphonce
 *
 */
public class SingletonNullCommand implements ICommand {

	/**
	 * This is a public class variable which holds a reference to the
	 * sole instance of this class (its singleton instance).
	 */
	public static final SingletonNullCommand SINGLETON = new
SingletonNullCommand();

	/**
	 * Private constructor for this class, in support of the SINGLETON
	 * pattern.
	 */
	private SingletonNullCommand() {}

	/**
	 * @see chapter5.ipg.ICommand#execute()
	 */
	public void execute() {
		// DO NOTHING - THIS IS THE NULL COMMAND
	}
}
```

Using the singleton pattern helps to support robustness because unnecessary memory resources are not expended on storing multiple instances of a class when only one is needed.

This example has demonstrated the potential application of a handful of design patterns in a real-world software system, and we briefly discussed some of the benefits that derive from their use.

Summary

This chapter introduced the notion of polymorphism. Polymorphism is a mechanism for dealing with alterative choices in a software system. As we have seen, polymorphism to this point it has involved the use of an interface as a mechanism which decouples an invariant component from a set of variant components. We also learned that design patterns can be useful tools in building software systems which exhibit desirable characterstics, such as robustness, flexibility and extensibility. Design patterns help because they allow us to decouple the invariant parts of our systems from their variant parts. The separation of concerns makes it easy to extend the code in controllable ways (by adding new variants to a pattern, for example). Interfaces and polymorphism play an important role in realizing this decoupling.

Case Study

Stay tuned…

Chapter Wrap-Up

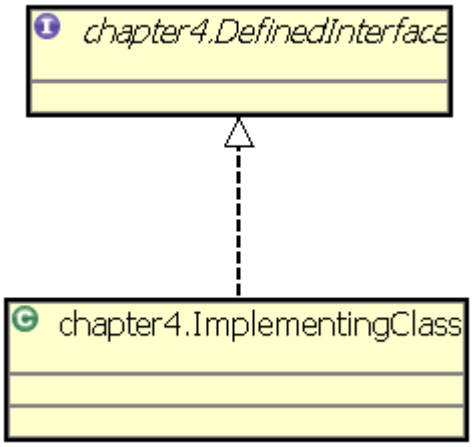
Learning Objectives

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

- Describe what polymorphism is.
- Identify polymorphism when used in code.
- List desirable properties of high quality software.
- Explain how judicious use of polymorphism can lead to high quality software.

Relationships Covered

Realization Informal Name: *realizes, implements* **First introduced in:** Chapter 4 – page **Error! Bookmark not defined.**



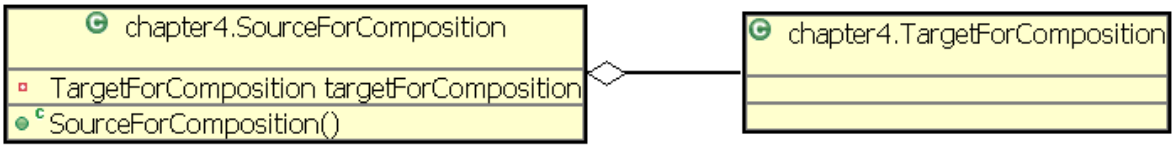
UML representation:
In code: (from above diagram)

```
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:



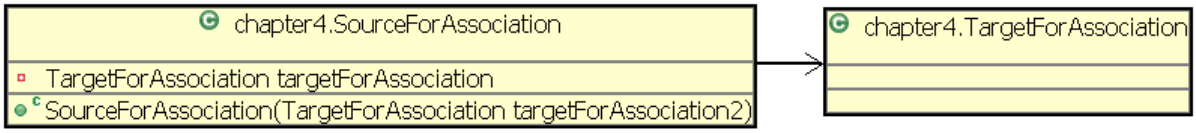
In code: (from above diagram)

```
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:



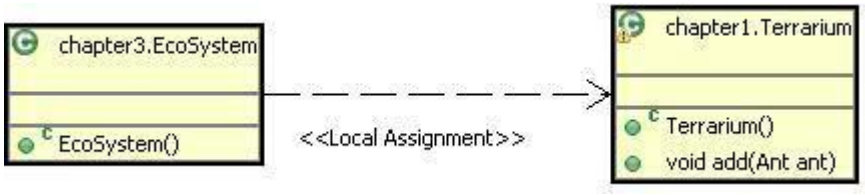
In code: (from above diagram)

```
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)

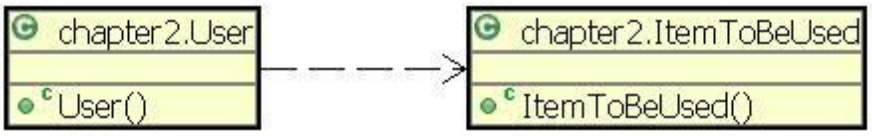
```
package chapter3;

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

`final` (page 13)

`static` (page 13)

`Previously:`

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

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

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

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

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

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

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

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

`return` (page **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:

correctness (8)

design patterns (8)

dynamicity (8)

extensibility (8)

flexibility (8)

maintainability (8)

polymorphic (1)

polymorphism (1)

robustness (8)

scalability (8)

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.**)

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

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

Church-Turing thesis (**Error! Bookmark not defined.**)

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

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

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

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

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

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

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

conceptual model (**Error! Bookmark not defined.**)

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

constructor body (**Error! Bookmark not defined.**)

constructor definition (**Error! Bookmark not defined.**)

constructor header (**Error! Bookmark not defined.**)

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

edit-compile-run cycle (**Error! Bookmark not defined.**)

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

empty method (**Error! Bookmark not defined.**)

executable model (**Error! Bookmark not defined.**)

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

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

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

implement an interface (**Error! Bookmark not defined.**)

instantiate [a class] (**Error! Bookmark not defined.**)

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

interactions pane (**Error! Bookmark not defined.**)

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

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

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

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

invoking [a method] (**Error! Bookmark not defined.**)

Java Virtual Machine (JVM) (**Error! Bookmark not defined.**)

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

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

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

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

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

method definition (**Error! Bookmark not defined.**)

method stub (**Error! Bookmark not defined.**)

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

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

naming conventions (**Error! Bookmark not defined.**)

naming rules (**Error! Bookmark not defined.**)

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

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

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

parameter list (**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.**)

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

supertype (**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.**)

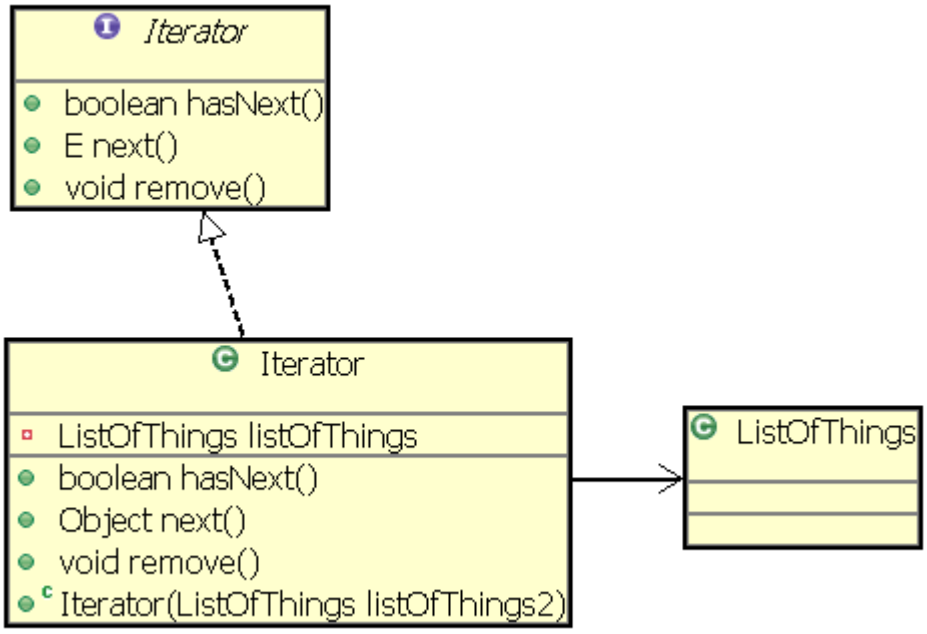
Design Patterns Covered

Iterator

First introduced in: Chapter 5 page 9

Usage: Used to traverse some list of objects. This pattern is natively implemented in Java with the use of the `java.util.Iterator` interface.

Simplified UML:

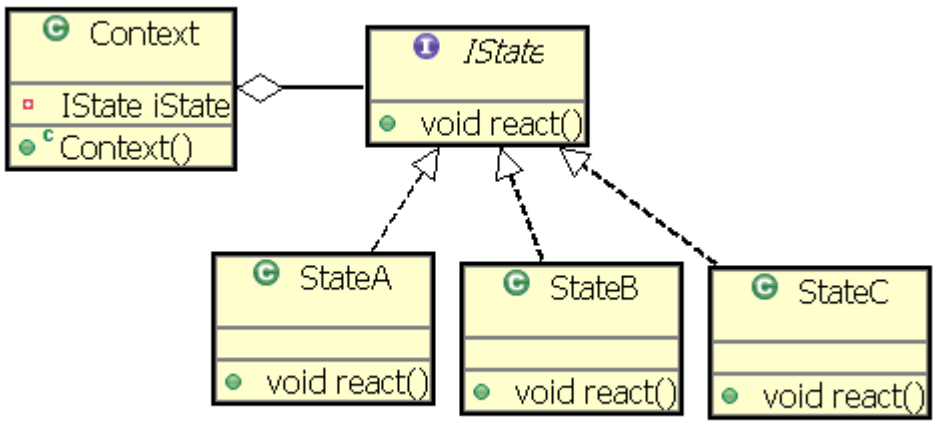


State

First introduced in: Chapter 5 page 10

Usage: Used to objectify the internal state of an object (the context) so the behavior of that object can be specific to the particular state it is in.

Simplified UML:

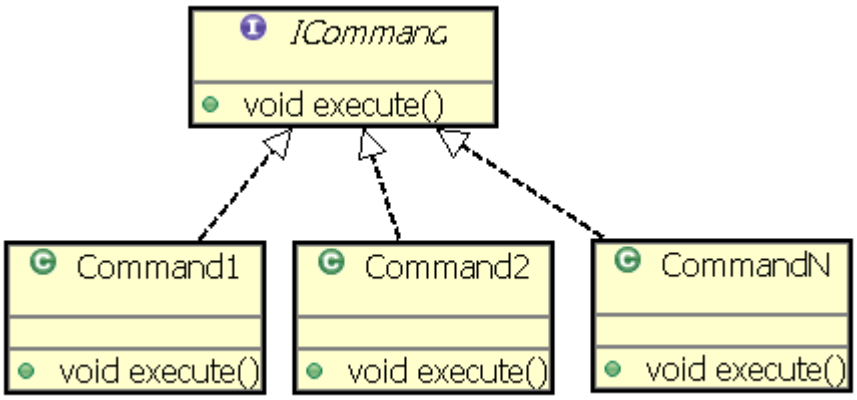


Command

First introduced in: Chapter 5 page 11

Usage: Used to objectify behaviors to keep track of what has happened in a system or what should happen in a system.

Simplified UML:

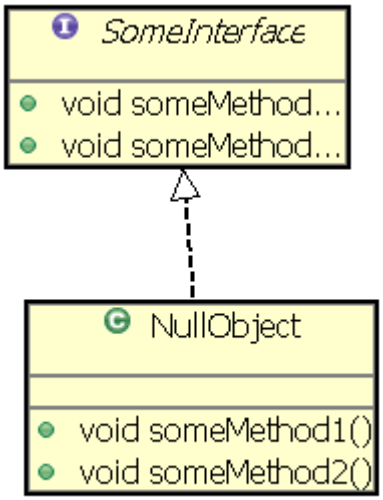


Null Object

First introduced in: Chapter 5 page 12

Usage: Used to represent a null value or the absence of behavior. This is an object that does nothing – a placeholder in the system.

Simplified UML:



Singleton

First introduced in: Chapter 5 page 13

Usage: Used when we will only ever need one instance of a particular object at runtime.

Simplified UML: