Chapter 7
Inheritance: Modeling subtype relationships

Pedagogic Goals

- Show how a system can be made flexible and resilient through the use of abstract specifications over concrete implementations.
- Demonstrate how different subtyping relationships identified in a model can be implemented using different forms of inheritance.

Introduction

A key part of building a model of a system is to identify the types from the system's domain which are relevant to the operation of the system, as well as the ways in which parts of the system communicate with each other. We now know the two ways to introduce new types into Java's type system: defining an interface and defining a class. We have also learned that implementation of an interface by a class is called the realization relationship. When a class realizes an interface the class' type becomes a subtype of the interface's type. The importance of this is that we can use an instance of a subtype whenever the supertype is called for.

The ability to use a subtype object to satisfy a supertype constraint allows us to write *generic* code – code that will work with many different (sub)types of objects. It all comes down to decoupling of components. If we couple components *closely* by making a client component dependent on a specific class we end up with a very *rigid* and *brittle* system. On the other hand, if we decouple components by making a client dependent only on an interface, then that client can work with any implementor of the interface. We end up with a *flexible* and *resilient* system.

In the last chapter we explored how the multiple roles that an object may play in a system can be modeled by having the roles cast as interfaces and by having the object's class defined to implement each of those interfaces. We saw how polymorphism came about by having a capability specified in a supertype and having different subtypes implement that capability in different ways.

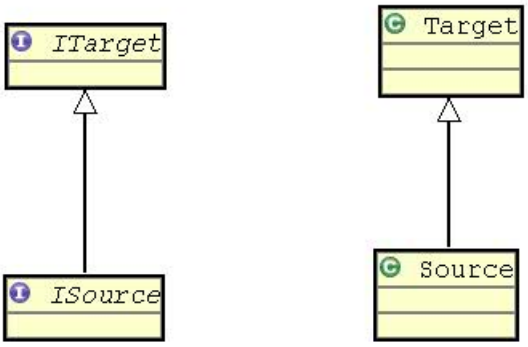
In this chapter we will learn that not only interfaces can play the role of supertype, but also classes. Defining a class as a subclass of another class involves yet another relationship. This relationship goes by many different names, including inheritance, generalization, and extension.[1] Like realization, generalization is a form of subtyping relationship. Unlike

[1] It is not uncommon to see the relationship referred to as an "is a" relationship, because a subtype object is also of the supertype. The Liskov substitution principle requires that subtype objects must be substitutible for supertype objects; some cases of "is a type of" do not fit this mold because the "subtype" is a restriction of the supertype, not an extension of it. "Uncle Bob" Martin discussed a marvelous example of this at the 2006 OOPSLA Educators' Symposium, showing that while we think of squares as a special type of rectangle, it is entirely inappropriate to model the relationship between these types using inheritance because while a square is a

realization, which is a relationship holding only between a class (as the subtype) and an interface (as the supertype), generalization can occur between two classes or between two interfaces. This chapter explores what this relationship, in its many forms, models.

The generalization relationship in UML

The generalization relationship, because it can occur between two interfaces or between two classes, will show up in UML class diagrams in one of two ways, as demonstrated in the diagrams below. What is common to both diagrams is that the relationship is represented by a solid line with a triangle at the target end of the relationship. Generalization between interfaces is shown in the left diagram, and generalization between classes in the right diagram.



The generalization relationship in code

The generalization relationship is expressed using the `extends` keyword in the interface or class header, as shown in the code snippets below, corresponding to the diagram above. First we see the code for interface extension:

```
package chapter6.generalizationInUML;

/**
 * @author <a href="mailto:alphonce@cse.buffalo.edu">Carl G. Alphonce</a>
 *
 * Created on: Oct 24, 2006
 *
 */
public interface ISource extends ITarget {

}
```

This expresses that the type introduced by the interface `ISource` is a subtype of the type introduced by the interface `ITarget`. The details of what this means is discussed in the next section.

Next, we see the code for class extension. The class header uses the `extends` keyword to identify the target class of the extension:

---

rectangle, it is a restriction, not an extension, of a rectangle. Because of this, we avoid using the name "is a" for this relationship.

```
package chapter6.generalizationInUML;

/**
 * @author <a href="mailto:alphonce@cse.buffalo.edu">Carl G. Alphonce</a>
 *
 * Created on: Oct 24, 2006
 *
 */
public class Source extends Target {

      /**
       * Creates a new instance of Source
       */
      public Source() {
      }

}
```

This expresses that the type introduced by the class `Source` is a subtype of the type introduced by the class `Target`. The details of what class extension implies is discussed later in the chapter.

Interface extension

To understand why subtyping is important, let us explore some examples. First, suppose we have a system in which multiple roles occur, and we wish to model that some roles sometimes must co-occur (occur simultaneously).

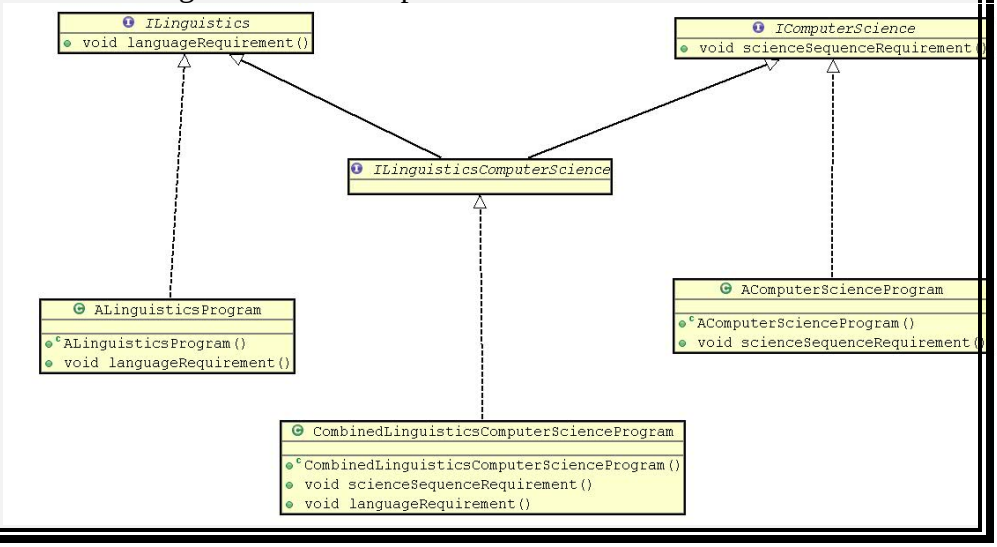
Example 1

Students at university follow a particular course of study, called their major. Different programs of study are often grouped together according to the "general education" requirements they impose. Arts, Science and Engineering programs often have different general education requirements. For example, a student might major in Linguistics (an Arts program), Computer Science (a Science program), or Biomedical Engineering (an Engineering program).

Most students have just one major. Some students decide to pursue two majors at the same time. This requires that students satisfy the general education requirements of both majors.

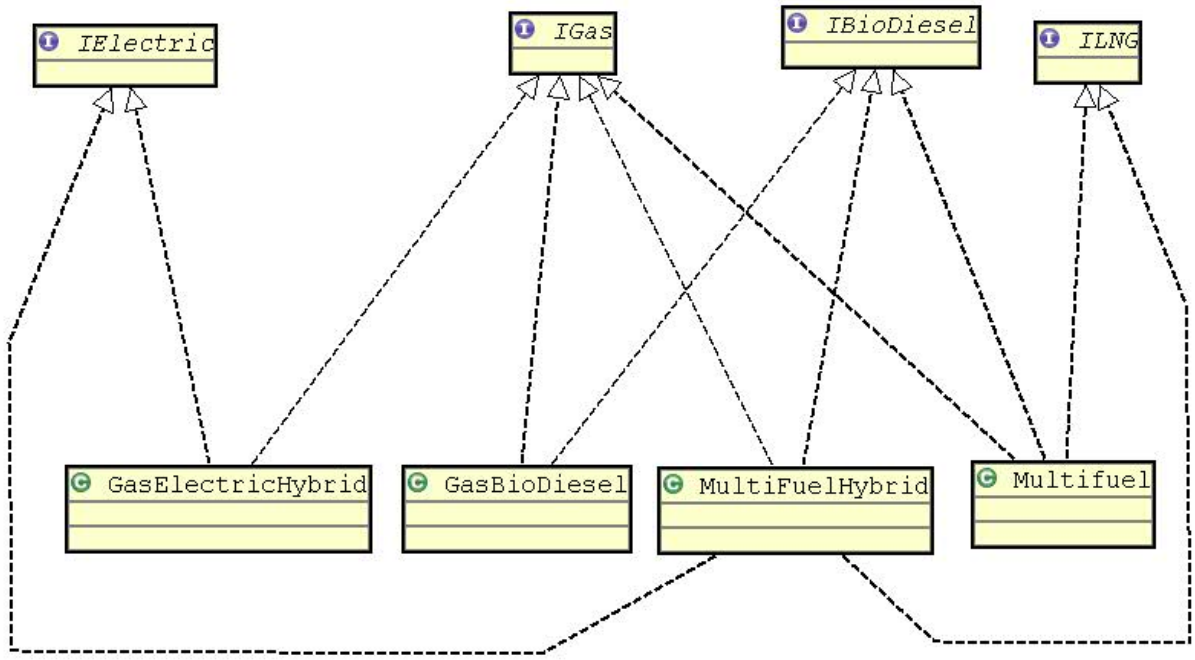
The general education requirements of different type of majors can be specified using interfaces, with specific requirements encoded in implementing classes. For example, a science degree might require a "full year science course sequence", but a specific major may require something more specific, such as a chemistry or physics sequence.

A student doing a double major, such as Linguistics and Computer Science, must satisfy the requirements of both majors. In order to differentiate students who are doing one or the other major from those doing both, it may be helpful to define a Linguistics-Computer Science interface, which extends the two interfaces Linguistics and Computer Science:

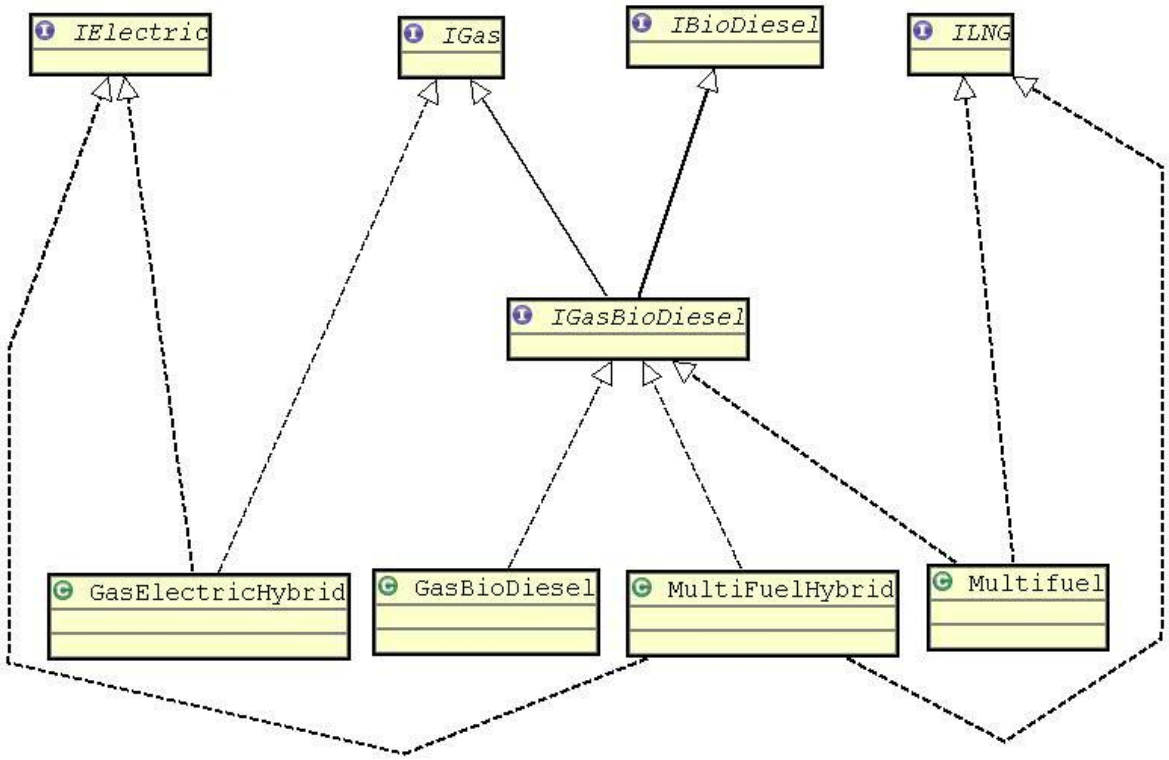


Let us push this a little bit further. There are many different ways to power passenger cars. Most cars use gasoline as a fuel, but some use alternatives. One alternative is to use diesel fuel. Another is to use liquefied natural gas (LNG). You may not have heard of or seen cars powered by LNG, but they are quite popular in some cities, especially amongst taxis because of the cost savings over gasoline or diesel. These cities have invested in building up a LNG infrastructure, so that LNG vehicles can refuel conveniently. Yet another alternative is to use a hybrid gas-electric system.

Suppose now that, in order to encourage more drivers to adopt multi-fuel vehicles, the highway department wants to charge differential tolls for multi-fuel vehicle on toll highways. All vehicles using toll roads must be equipped with a transponder (e.g. many states use the E-ZPass system for electronic moving toll collection). This transponder will transmit a signal indicating the type of fuel combination the vehicle's engine can use. The appropriate toll is charged based on the fuel combination the vehicle supports. In the UML class diagram below we show four different types of fuel: gasoline, liquefied natural gas, biodiesel, and electricity. Each of these types is modeled using an interface (`IGas`, `ILNG`, `IBioDiesel`, and `IElectric`). Classes are used to model vehicles which can use the different types of fuels. If a vehicle type can use more than one different sort of fuel it implements all of the relevant interfaces. For example, the `GasElectricHybrid` class implements both the `IGas` and `IElectric` interfaces.



The problem here is that we do not have a type for a vehicle that runs on gas and biodiesel, and possibly other fuels as well. There are several classes which implement both the `IGas` and `IBioDiesel` interfaces, but there is no type which names this combination. But suppose that the thruway authority wants to charge vehicles with this particular combination of fuel capabilities lower rates than everybody else. In this case there needs to be a single type which can represent those vehicles in our system. What we would like is a diagram which looks more like the following:



The `IGasBioDiesel` interface type can now exists as a real type in our system![2] The toll collection system can now make use of this type to charge the correct (lower) toll to all vehicles which are of this type. It is worth pointing out here the syntax to have an interface extend more than one interface. If an interface is to extend multiple interfaces, the interfaces are listed in the source interface's header, after the `extends` keyword, separated by commas:

```
public interface ISource extends ITarget1, ITarget2, ITarget3
```

What we have done is introduce new interfaces which are *subinterfaces* of existing interfaces. In other words, they *extend* existing interfaces.

The inheritance relationship between interfaces has the same effect on typing relationships as did the realization relationship: the extending interface becomes a subtype of the extended interface. Thus, the `IGasBioDiesel` type is a subtype of both the `IGas` type and of the `IBioDiesel` type.

It is worth noting at this juncture that subtyping is a *transitive* relationship. This means that if X is a subtype of Y and Y is a subtype of Z, then X is a subtype of Z too. More concretely, since `GasBioDiesel` is a subtype of `IGasBioDiesel`, and `IGasBioDiesel` is a subtype of both `IGas` and `IBioDiesel`, it follows that `GasBioDiesel` is a subtype of both `IGas` and `IBioDiesel`. Why is this interesting? It tells us that not only can we use a `GasBioDiesel` object whenever we specifically need an

[2] Notice the arc and arrow used to represent this relationship: it employs same arrowhead as does the realization relationship, but its line is solid rather than dashed.

object of type `GasBioDiesel`, we can use a `GasBioDiesel` object whenever we need an object of any of its supertypes: `IGasBioDiesel`, `IGas`, and `IBioDiesel`.

But more importantly, when we need something of type `IGasBioDiesel`, we can use *any* of its implementations: `GasBioDiesel`, `MultiFuelHybrid`, or `MultiFuel`. In fact, we could even add a new class which implements the `IGasBioDiesel` interface and have it work with out system.

This means that all of Java's types are arranged into a type hierarchy, and whenever a class or an interface is defined Java's type system is extended to include this new type. Part of the task of a software developer is to decide, for a particular problem, what the appropriate types are, and to define and situate them appropriately in Java's type hierarchy.

Declared and Actual Type, revisited

Earlier on in the book we introduced the notions of declared type (of a variable) and actual type (of an object). It is now time to revisit these notions and examine in more detail what the significance of the declared type of a variable versus the actual type of the object it refers to really is.

Why do we declare a variable to be of a particular type? In part we do this to restrict the types of objects that the variable can refer to. A better answer is that the type of the variable determines what methods that can be called on the object which the variable refers to. It will be important for us to remember this as we progress through the chapter.

Inheritance between Interfaces

When an interface inherits from another interface, there are implications beyond subtyping. When one interface extends another interface, it *inherits* all the method specifications of the interface it extends. For example, consider the two interfaces shown below, `IAudioDevice` and `ITuner`. The fact that `ITuner` extends `IAudioDevice` is indicated by the extends clause which is part of the interface header.


```
package chapter6;

/**
 * @author alphonce
 */
public interface IAudioDevice {
	/**
	 * When invoked, this method causes the device
	 * to power down to standby mode.
	 */
	public void turnOff();
	/**
	 * When invoked, this method causes the device
	 * to power up from standby mode.
	 */
	public void turnOn();
	/**
	 * When invoked, this method causes the device's volume to increase
	 * by five dB, up to the maximum volume level.
	 */
	public void volumeUp();
	/**
	 * When invoked, this method causes the device's volume to decrease
	 * by five dB, down to a minimum level of 0 dB.
	 */
	public void volumeDown();
}
```

```
package chapter6;

/**
 * @author alphonce
 */
public interface ITuner extends IAudioDevice {
	/**
	 * When invoked, causes the tuner to scan at higher frequencies for
	 * a strong signal, stopping if it finds one.  If none is found,
	 * stops scanning at the highest frequency of the tuner.
	 */
	public void scanUp();
	/**
	 * When invoked, causes the tuner to scan at lower frequencies for
	 * a strong signal, stopping if it finds one.  If none is found,
	 * stops scanning at the lowest frequency of the tuner.
	 */
	public void scanDown();
}
```

The `ITuner` interface extends the `IAudioDevice` interface: this means that not only does the `ITuner` interface require that the two methods `scanUp` and `scanDown` be defined in an implementing class, but also all of the methods specified in the parent interface, `IAudioDevice`. Thus, a class which implements the `ITuner` interface must define not only the two methods `scanUp` and `scanDown`, but also `turnOff`, `turnOn`, `volumeDown`, and `volumeUp`.

When an interface inherits from more than one interface several questions come to mind:

Q: What happens if two methods with different names are inherited from different parent interfaces?

```
public interface ParentOne {
    public void methodA();
}

public interface ParentTwo {
    public void methodB();
}

public interface Child extends ParentOne, ParentTwo {
    public void methodC();
}
```

A: The `Child` interface inherits both `methodA` and `methodB`.

Q: What happens if two methods with the same name and with the same parameter lists are inherited?

```
public interface ParentOne {
    public void methodA();
}

public interface ParentTwo {
    public void methodA();
}

public interface Child extends ParentOne, ParentTwo {
    public void methodC();
}
```

A: The `Child` interface inherits the same specification from both of its parents. Since the method specification is the same from both parent interfaces, the `Child` interface ends with just two method specifications in total: one for `methodA` and one for `methodC`.[3]

Q: What happens if two methods with the same name but with different parameter lists are inherited?

```
public interface ParentOne {
    public void methodA();
}
```

[3] If this seems odd to you, perhaps the following analogy might help. The methods specified in interfaces are requirements imposed on classes which implement those interfaces. They are similar to expections that parents put on their children. Generally children must satisfy the expectations of both parents. If mom says you must clean your room and dad says you have to cut the grass, then you must do both things. However, if mom says you must clean your room and dad also says you must clean your room, then you only have to clean your room once (because once you've satisfied the requirement for one parent you've automatically satisfied it for the other parent).


```
public interface ParentTwo {
    public void methodA(IHostess h);
}

public interface Child extends ParentOne, ParentTwo {
    public void methodC();
}
```

A: In this case the two methods, even though they share the same name, are treated by Java as different methods. In this case the `Child` interface is left with three method specifications, one for `methodA` taking no arguments, one for `methodA` taking one argument of type `IHostess`, and one for `methodC` taking no arguments.

To summarize what we have learned about inheritance between interfaces: an interface can extend zero or more interfaces, an interface inherits all the method specifications from all of its parents, with identical method specifications inherited from multiple parents treated as the same method specification.

Abstract classes and partial realization

Up to this point we have assumed that when a class implements an interface it must define all the methods specified in the interface. It turns that that this is not quite true. Classes can leave some methods undefined (i.e. they remain as method specifications); a class which does not define all of its methods is known as an *abstract class*. In contrast, a class which is not abstract (and which therefore defines all of its methods) is called a *concrete class*.

An abstract class, because it does not define all of its methods (and is therefore not providing a way to respond to method calls) cannot be instantiated.[4] In other words, there can be no object whose *actual* type is that of an abstract class. There is no problem in having a variable whose declared type is that of an abstract class, just as there is no problem in having a variable whose declared type is that of an interface: the declared type simply specifies that only the methods specified by the type can be invoked on the object referred to by the variable.

Inside the code for an abstract class, we would see the keyword `abstract` in the class header as well as in the method headers for any methods that we will declare abstract (i.e. leave without implementation). For example, the header for an abstract class named `Mammal` would look like this:

```
public abstract class Mammal
```

or it could look like this (Java will accept either):

```
abstract public class Mammal
```

[4] This is not quite accurate. A class can be labelled "abstract" even if it defines all of its methods, simply to prevent instantiation of the class. We will not discuss this sort of abstract class further.

A method declaration for an abstract method named `doSomething` could look like this:

```
public abstract void doSomething();
```

or this:

```
abstract public void doSomething();
```

What might be the motivation for defining an abstract class? An abstract class allows us to model that two or more subclasses should share a method definition, without committing to the idea that there should be objects of this actual type in the model.

Example 2

Consider modeling aircraft for use in an air traffic control application. An air traffic control system manages aircraft of different types. Examples of different type of aircraft include Boeing 747, Airbus A320, and the Saab-Fairchild SF340. Each of these aircraft has very different properties and requirements: the Boeing 747 is a very large jumbo jet which requires quite a long runway to take off and land. The SF340 is a small turboprop commuter aircraft. In this application it is important that the category "aircraft" have some reality (aircraft are shown on the controller's radar screen), yet the actual types of aircraft objects in the system should be specific types of aircraft (like 747, A320 or SF340). There is no real-world counterpart to a generic aircraft object, so no such object should exist in our model either.

Should the type aircraft be modeled using an interface or an abstract class? If there is no shared behavior then an interface is likely most appropriate. If there is some capability for which all these aircraft types share a common definition, then an abstract class is the correct way to model this.

An example of a common, or shared, capability is to respond to an Air Traffic Control interrogation signal. If it is important that this be faithfully represented in the computational model then this should be modeled as a method in the abstract aircraft class. Note, however, that there is no single correct answer: how to model a domain depends on many factors, and the decision may well be left to the judgement of the software architects.

Class extension

Now we study the extension relationship as it holds between two classes. In this case the extended class is often called the superclass, the base class, or the parent class. The extending class is often called the subclass, the derived class, or the child class.

The Java class hierarchy

When dealing with class extension in Java it is important to note that Java supports only *single inheritance* for classes. This means that a class in Java can extend at most one class.

As we saw earlier in the chapter, to indicate that one class extends another an (optional) extends clause is added to the class header, analogously to how the (optional) extends clause is added to an interface header. Note that the extends clause comes before the inherits clause in those cases where a class both extends another class and implements one or more interfaces.

Although the extends clause is optional in the Java class header, in fact every *user-defined* class in Java extends *exactly* one class. There is only one class which is not user-defined: the class whose name is `Object`. The class `Object` is the default parent class when a parent class is not explicitly given.

Java's classes are organized into a hierarchy. Unlike Java's type hierarchy, Java's class hierarchy forms a special structure called a *tree*. This is due to Java being a single-inheritance language (as far as its classes is concerned) and that there is a common superclass for all Java classes: `Object`.

Consequences of Class to Class Generalization

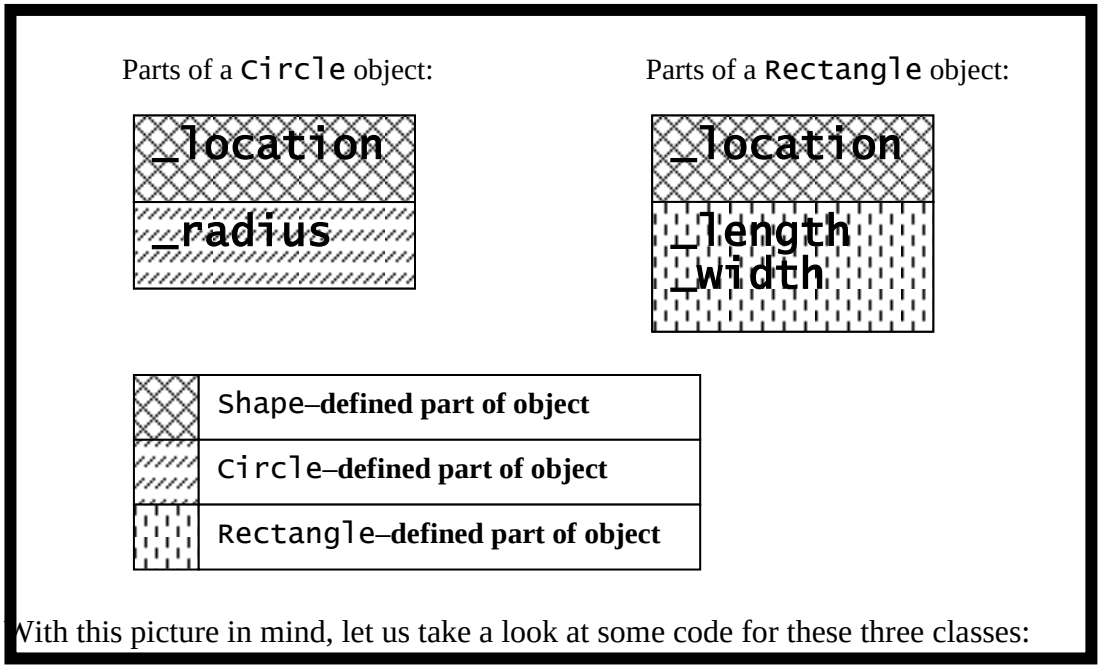
What are the consequences of class to class generalization? It turns out that the same consequences of *interface to interface* generalization apply to *class to class* generalization, but there are some added consequences. Recall that the consequences of generalization for interfaces have to do with subtyping relationships and inheritance of method specifications. Because classes contain more than just method specifications, we have to address what inheritance means for the things that classes have that interfaces do not: method definitions and instance variables.

Object representation and construction

In order to understand some of these new aspects of inheritance, we must first refine our understanding of the representation of an object in the memory of a machine. One new aspect of object structure which is a consequence of inheritance is that an object instantiated from a subclass as a two-part structure: the object has both a superclass part and a subclass part. Recall that the state of an object consists of its instance variables and their values, at a given point in time. With our revised notion of the structure of an object, we must recognize that the variables which define the state of an object are not only of the instance variables declared in the class from which the object instantiated, but also the instance variables declared in its superclasses.

When the constructor of a subclass is run, its job is to initialize the state of the newly created object. How can it do this, when part of the state is contained in superclasses? The constructor of the subclass does not have access to any private instance variables declared in the superclass. Yet we cannot simply ignore the superclass-defined instance variables, because the superclass-defined methods of the object *can* access those variables. To overcome this dilemma let us remember that it is the job of the constructor of a class to initialize the state of an instance of the class. This observation leads us to wonder whether we cannot simply let the superclass' constructor handle the initialization of the superclass part of the object. It turns out that this is exactly what we want to do. In the constructor of the subclass we want to invoke the superclass' constructor. Java provides a special keyword to let us do this, `super`. This keyword can be used to invoke the superclass' constructor, by placing an actual parameter list after `super`.

The following code example demonstrates the basic idea. This example is purposely engineered to demonstrate the use of superclass constructor invocation without unnecessary distractions, so you should not read too much into the code itself. The example consists of three classes, `Shape`, `Circle`, and `Rectangle`. The idea is that these classes represent shapes which have both a location and a size. While each shape has a location, represented using a `java.awt.Point` object, the sizes of `Circle` and `Rectangle` objects are represented quite differently. The size of a `Circle` is determined by its radius, which is a `Scalar` quantity. The size of a `Rectangle` is determined by its length and width, each of which is a `Scalar` quantity. Because the location property is shared between both `Circle` and `Rectangle` objects but their size representation is not shared, `Shape` is an abstract class which both `Circle` and `Rectangle` extend. The location property is part of the `Shape` class, but each subclass has its own property or properties for representing its size. Here is a drawing which shows the different parts of the representations of `Circle` and `Rectangle` objects:



With this picture in mind, let us take a look at some code for these three classes:

```
package chapter6.constructors;

/**
 * @author <a href="mailto:alphonce@cse.buffalo.edu">Carl G. Alphonce</a>
 *
 * Created on: Oct 26, 2006
 *
 */
public abstract class Shape {
	/**
	 * The location of this shape on the screen.
	 */
	private java.awt.Point _location;

	/**
	 * Creates a new instance of Shape
	 * @param initialLocation is the initial _location.
	 */
	public Shape(java.awt.Point initialLocation) {
		_location = initialLocation;
	}
}
```

```
package chapter6.constructors;

/**
 * @author <a href="mailto:alphonce@cse.buffalo.edu">Carl G. Alphonce</a>
 *
 * Created on: Oct 26, 2006
 *
 */
public class Circle extends Shape {
	/**
	 * The radius of this Circle.
	 */
	private Scalar _radius;

	/**
	 * Creates a new instance of Circle.
	 * @param radius
	 * @param location
	 */
	public Circle(Scalar radius, java.awt.Point location) {
		super(location);
		_radius = radius;
	}
}
```

```
package chapter6.constructors;

/**
 * @author <a href="mailto:alphonce@cse.buffalo.edu">Carl G. Alphonce</a>
 *
 * Created on: Oct 26, 2006
 *
 */
public class Rectangle extends Shape {
	/**
	 * The length of this Rectangle.
	 */
	private Scalar _length;

	/**
	 * The width of this Rectangle.
	 */
	private Scalar _width;

	/**
	 * Creates a new instance of Rectangle.
	 * @param length
	 * @param width
	 * @param location
	 */
	public Rectangle(Scalar length, Scalar width, java.awt.Point
location) {
		super(location);
		_length = length;
		_width = width;
	}
}
```

Notice that the `Shape` class, even though it is abstract, has a constructor. This is because each class must take responsibility for setting up the initial state of its instances. Although the `Shape` class is abstract, and therefore cannot be directly instantiated, the `Shape` class is indirectly instantiated whenever an instance of one of its subclasses in created. Notice now that in each of the two subclasses the constructor relies on the superclass constructor to initialize the instance variable declared in the `Shape` superclass.

You might notice that an explicit invocation of the superclass constructor is not *required*: we said above that every user-defined class extends exactly one class, and although we have seen many class definitions up to this point, we have only just learned about `super`. How is it possible that the examples we have seen to this point compiled without errors? It turns out that the Java compiler inserts an invocation of the superclass constructor in any constructor where it is not explicit. The default invocation is of the superclass' no-argument constructor.[5] The invocation of a superclass' constructor must be the first thing done in a constructor.

[5] The Java compiler allows constructors to be omitted from class definitions as well. If a class is defined with no explicit contstuctor, then the Java compiler inserts a no-argument constructor whose sole line of code is a call to `super()`. So, for example, in the definition of a class named Foo, if no explicit constructor is given the compiler inserts the following constructor:

```
public Foo() {
```

Overriding of method definitions

Recall that when an interface extends another interface, it inherits its superinterface's method *specifications*. When a class extends another class, it inherits its superclass' method *definitions*.[6] This means that when a class inherits from another class, all non-private methods of the superclass are available in the subclass as well.

It is not uncommon for the correct implementation of a capability in a subclass object to be different than the implementation of that capability in the superclass. In this case the subclass implementor has two choices: *total* or *partial overriding*. When a subclass totally overrides a method definition inherited from its superclass, it simply provides a completely new definition for the method in question. More interesting is the case where a subclass method should augment the superclass capability in some way. In order to access the superclass definition of an overridden method, we must reference the superclass definition using `super`, which is a reference to the superclass object.

Example 3

Consider a simulation environment in which we can place models of animals. It is common in such environments that the simulation proceeds in timesteps, driven by a timer of some sort. When the timer determines that a pre-set amount of time has passed, it notifies all the objects in the simulation environment to update.

```
}   super();
```

It is important to note that if any constructor is explicitly defined in a class, the no-argument constructor (sometimes called the "default" constructor) is not provided automagically.

A common beginner's error is to define an explicit constructor with an argument in a superclass, but to not define any constructor in a subclass, as in the following code sketch:

```
public class Bar {
       // Explicit constructor: no no-argument constructor.
       public Bar(String s) {
              ...
       }
       ...
 }

public class Foo extends Bar {
       // No explicit constructor: compiler generates default constructor.
       ...
}
```

This results in the following, potenially mysterious, error:

```
Implicit super constructor Bar() is undefined for default constructor.  Must define
an explicit constructor.
```

[6] In this chapter we assume that the two levels of access control are public and private. The complete story of access control in Java is a bit more involved, however. Java has four levels of access control; from least restrictive to most restrictive they are public, protected, package-private and private. Members which are marked with private access are visible only in that class. Members which are marked package-private are accessible to all classes in the same package, but nowhere else. Members which are marked protected, they are accessible within the package and also within subclasses of the class, regardless of what package the subclasses are in. Members which are marked public are accessible everywhere.

One common property of animals is that they move. The default moving behavior might be to move in a straight line at a fixed velocity. Different subtypes of animals might augment this basic moving behavior in different circumstances. For example, a predatory animal may speed up and track a prey animal, assuming it is hungry and close enough to the prey. A prey animal may speed up and take some evasive action, changing direction suddenly, to avoid becoming a nice, light snack.

If a class Animal defines an update method that causes an animal to move in a straight line (whichever way it is currectly facing), then the two subclasses Predator and Prey could be defined to (partially) override the update method in different ways. Assuming that the method header of update in Animal is

```
public void update()
```

then an overriding definition would look something like this:

```
public void update() {
   ...
   super.update();
   ...
}
```

Special references

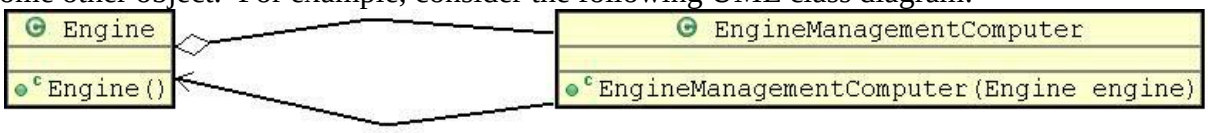
Java provides two special object references, `super` and `this`. The `super` reference we encountered above, when discussing class to class extension. The `super` reference gives us access to members defined in a class' superclass.

The `this` reference is a self-reference. It refers to the instance of a class on which a method has been invoked. It can be used to explicitly call an instance method defined for an object from another method within that same object. For example:

```
this.someMethod()
```

invokes the method `someMethod()` on the current object.

More commonly, this is used to pass a reference to the current object along to a method on some other object. For example, consider the following UML class diagram:



How can this be implemented? Consider, in particular, the constructor for the Engine class. This constructor creates an instance of EngineManagementComputer, whose constructor needs a reference to an Engine object. We don't want to write,

```
new EngineManagementComputer(new Engine())
```

for a variety of reasons. From a modeling perspecitve, this would mean that when we create a new Engine, it has a new EngineManagementComputer, which controls *some other* Engine!

This is clearly a pretty bad idea. What we want is for the EngineManagementComputer to control the Engine in which it is installed! Using `this`, we write the code as follows:

```
new EngineManagementComputer(this)
```

Putting everything together, we have:

```
package chapter6.selfReference;

/**
 * @author <a href="mailto:alphonce@cse.buffalo.edu">Carl G. Alphonce</a>
 *
 * Created on: Oct 29, 2006
 *
 */
public class Engine {

	/**
	 * _emc is this engine's Engine Management Computer.
	 */
	private EngineManagementComputer _emc;

	/**
	 * Creates a new instance of Engine.  Takes responsibility for
	 * creating a new EngineManagementComputer for this engine.  Passes
	 * 'this' along to the EMC.
	 */
	public Engine() {
		_emc = new EngineManagementComputer(this);
	}

}
```

```
package chapter6.selfReference;

/**
 * @author <a href="mailto:alphonce@cse.buffalo.edu">Carl G. Alphonce</a>
 *
 * Created on: Oct 29, 2006
 *
 */
public class EngineManagementComputer {

	/**
	 * The engine that this EMC controls.
	 */
	private Engine _engine;

	/**
	 * Creates a new instance of EngineManagementComputer,
	 * hooked up to the engine it controls.
	 * @param engine
	 */
	public EngineManagementComputer(Engine engine) {
		super();
		_engine = engine;
	}

}
```

Summary

In this chapter we have learned about the generalization relationship. In particular we have learned that an interface can extend multiple other interfaces, thereby inheriting the method specifications of its parent interfaces. We have also learned that every user-defined class extends exactly one class; if no class is explicitly mentioned as a parent class, the parent class defaults to the class `Object`. When a class extends another class it inherits its public methods. A subclass can choose to inherit methods unchanged, to alter them completely (total overriding), or alter them somewhat (partial overriding).

Case Study

Stay tuned…

Chapter Wrap-Up

Learning Objectives

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

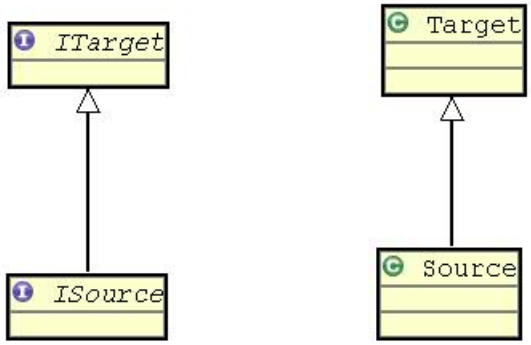
- Describe what an abstract class is and what differentiates it from a concrete class.
- Describe the effects of inheritance between two interfaces, two concrete classes, and an abstract class and a concrete class.

- Identify when it would appropriate to use inheritance while building a piece of software.
- Recognize when a class is calling methods or constructors from its superclass.

Relationships Covered

Generalization

Informal Name: *extends*
First introduced in: Chapter 6 – page 2
UML representation:



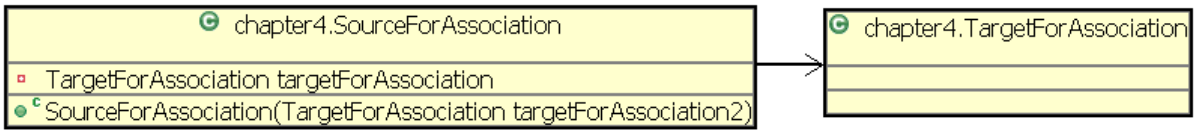
In code: (from above diagram)

```
public interface ISource extends ITarget {
}

public class Source extends Target {
}
```

Note: There is no code inside either of the classes `Target` or `ITarget` 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)

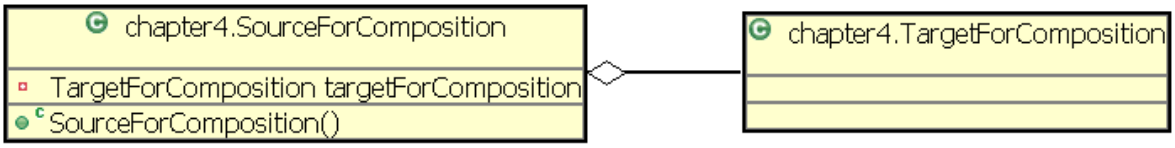
```
package chapter4;

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

Composition

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



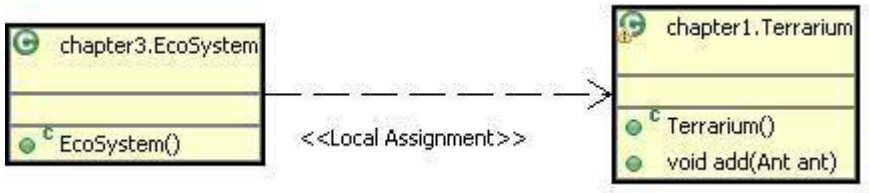
In code: (from above diagram)

```
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

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() {
```

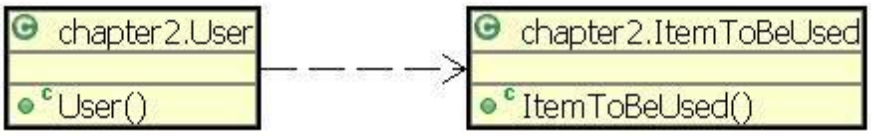
```
            chapter1.Terrarium _terrarium = new chapter1.Terrarium();
      }
}
```

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:



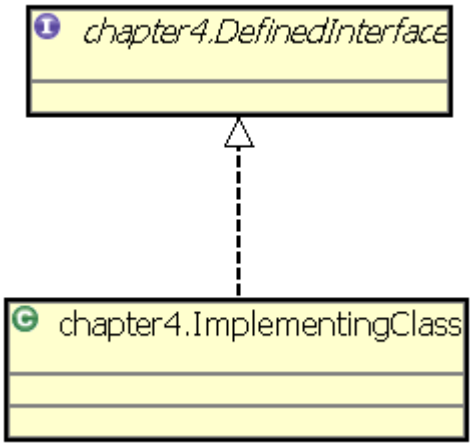
In code: (from above diagram)

```
package chapter2;

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

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

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

Keywords Covered

`abstract` (page 10)
`extends` (page 2)
`super` (page 12)
`this` (page 17)

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

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:

(in next draft)

Previously Covered:

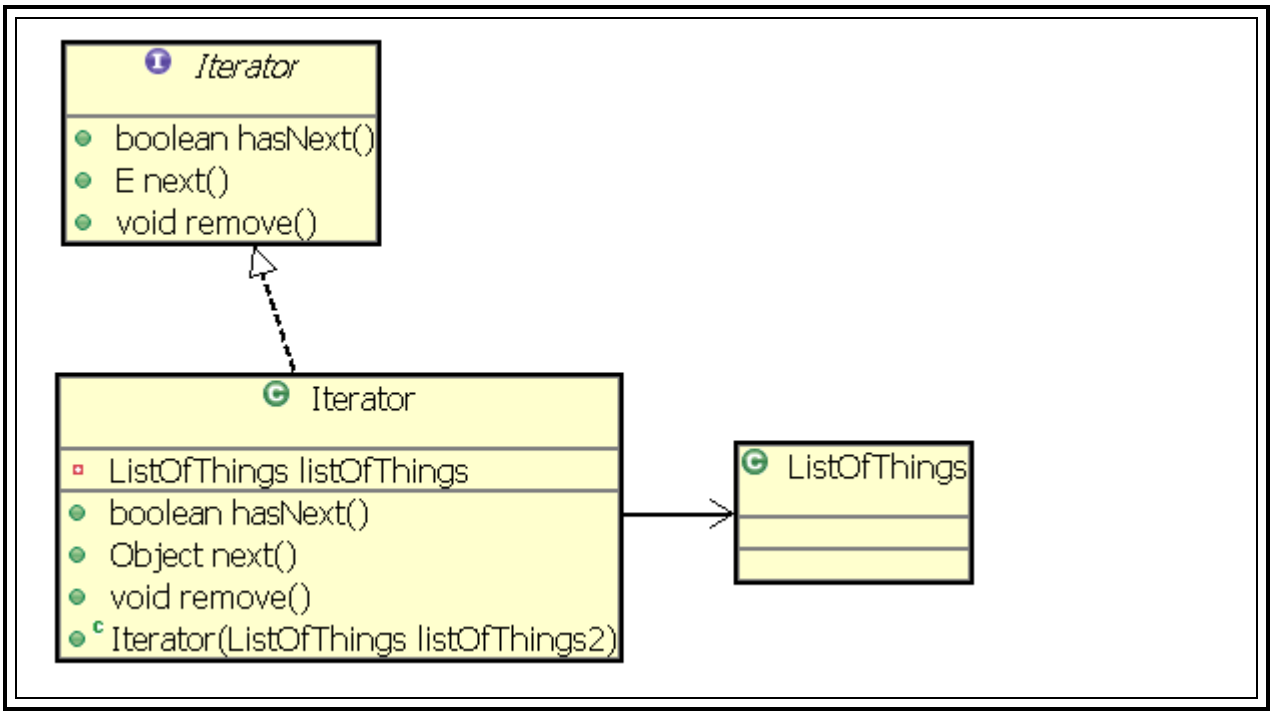
Design Patterns Covered

Iterator

First introduced in: Chapter 5 page **Error! Bookmark not defined.**

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:

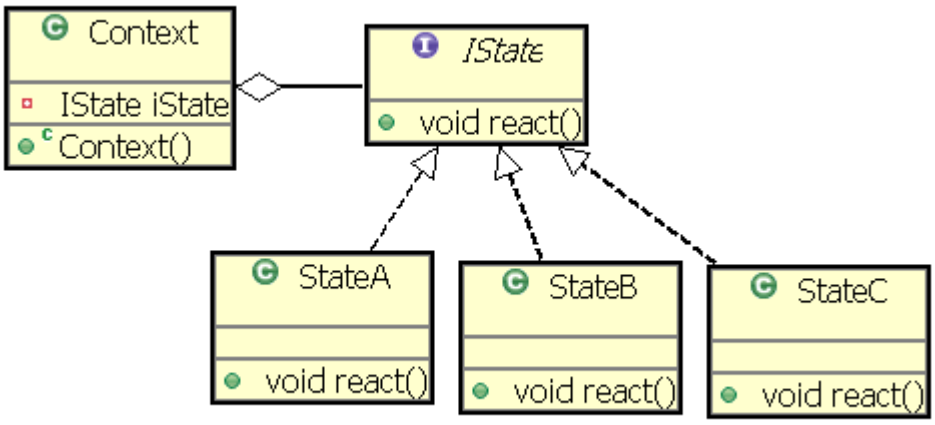


State

First introduced in: Chapter 5 page **Error! Bookmark not defined.**

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:

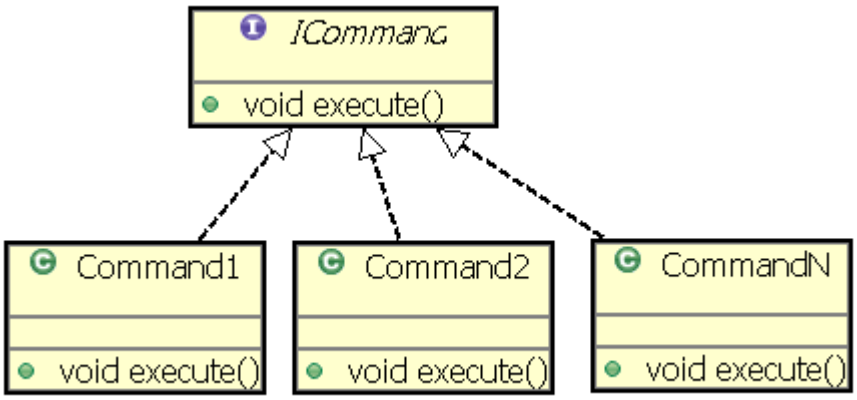


Command

First introduced in: Chapter 5 page **Error! Bookmark not defined.**

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

Simplified UML:

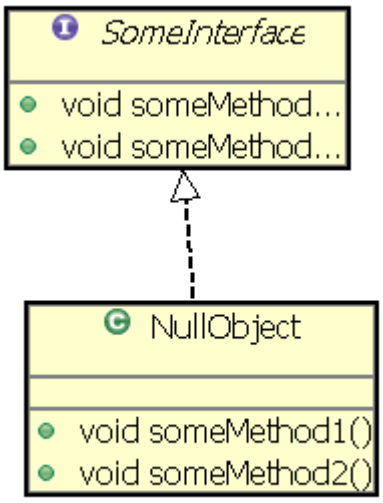


Null Object

First introduced in: Chapter 5 page **Error! Bookmark not defined.**

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 **Error! Bookmark not defined.**

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

Simplified UML: