Abstract

The union design pattern is a structural pattern that depicts the inheritance relationship between a superclass and its subclasses. The superclass is an abstract representation of the union of all the subclasses. Due to this polymorphism, the subclasses can thus be used wherever the superclass is required.

Inheritance and polymorphism (discussed below) are two sides of the same coin and represent very foundational concepts in object-oriented programming. The union design pattern is an expression of these relationships and enables us to talk about them in a more tangible manner.

1 Union Design Pattern

Consider the following "is-a" or inheritance relationships between "concrete" entities Coca Cola, sulfuric acid, milk and the "abstract" liquid (named "ALiquid" by convention since it is an abstract entity):

![Diagram of Union Design Pattern]

**Figure 1:** ALiquid is the union of CocaCola, SulfuricAcid and Milk
The UML diagram shows us that Coca Cola, sulfuric acid and milk are all liquids. Conversely, the diagram tells us that a liquid could be either Coca Cola, sulfuric acid or milk. Note of course, that liquids are not constrained to these 3 entities but that doesn’t affect the discussion here—in fact, this will be an important feature later on.

Another way that we can express the notions depicted by the diagram is to say that the abstract \textit{ALiquid} superclass represents the union of Coca Cola, sulfuric acid and milk. That is,

\textit{a superclass represents the union of all of its subclasses.}

or in other words

\textit{a superclass represents all that is abstractly equivalent about its subclasses.}

For instance, the notion of an abstract liquid embodies all the behaviors and attributes such as having no definite shape, randomized atomic positions, freezing and boiling points that are common to Coca Cola, sulphuric acid and milk. Note the fine distinction between having a value and having the same value.

\textit{NOTE:} In general, an interface can be substituted for the abstract superclass discussed here with no loss of generality.

The above diagram illustrating the relationship between a superclass and its subclasses is called the \textit{Union Design Pattern}. The union pattern shows us \textit{inheritance} in how the Coca Cola, sulfuric acid and milk will all inherit the abstract behaviors of liquids, such as the lack of a definite shape and freezing/boiling points. Conversely, it also shows that if a situation utilizes a liquid, either Coca Cola, milk or sulphuric acid can be used as they are all abstractly equivalent as liquids. Note that this does not imply that all three will act identically! For instance, the human throat can swallow any liquid because it is made to work with fluids that can flow. However, the reaction of the throat to sulphuric acid is markedly different than it reaction to milk! This ability to substitute any subclass for its superclass and get different behavior is called \textit{polymorphism}.

\section*{1.1 Abstraction vs. Commonality}

A subtle but extremely important point is that

\textit{Commonality does not imply abstract equivalence.}

Just because a feature is common to every item in a set, does not necessarily mean that it represents some sort of abstract feature of those elements. For instance, cats, dogs, humans, and rats are all mammals where a mammal is defined as an animal that produces milk to feed its young. One could thus make a class model where a superclass \textit{Mammal} has subclasses \textit{Cat}, \textit{Dog}, \textit{Human} and \textit{Rat}. One common feature is behavior is that cats, dogs, humans and rats all give live birth of their young. So it is tempting to say that the \textit{Mammal} class should also embody the "live birth" behavior. However, as one wanders the world discovering new mammals, in the backwaters of Australia one finds the duck-billed platypus\footnote{http://www.creationscience.com/onlinebook/LifeSciences13.html} which produces milk and is therefore clearly a mammal. However, the duck-billed platypus also lays eggs. Thus the "live birth" behavior does not belong in the \textit{Mammal} superclass as it was only a coincidence that it was common to our initial set of subclasses. More importantly, being able to give live birth was never part of the abstract definition of a mammal and thus should never have been included in the \textit{Mammal} superclass in the first place.

Cats, monkeys and whales, while diverse creatures, are all mammals. Hence to model such a system in the computer, it makes sense to make \textit{Cat}, \textit{Monkey} and \textit{Whale} all subclasses of an abstract \textit{Mammal} superclass. Each species has many behaviors (methods) but I will only concentrate on 3 in particular:

\begin{itemize}
\item \texttt{1. boolean givesMilk()} : returns true if the animal can give milk to feed its young, false otherwise
\item \texttt{2. String makeSound()} : returns a String representation of a common sound the animal makes.
\end{itemize}
3. boolean givesLiveBirth(): returns true if the animal bears live young.

In the table below are the methods and what happens when each species executes that method:
<table>
<thead>
<tr>
<th>Mammal</th>
<th>Method</th>
<th>boolean givesMilk()</th>
<th>String makeSound()</th>
<th>boolean givesLiveBirth()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat</td>
<td>true</td>
<td>&quot;Meow&quot;</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Monkey</td>
<td>true</td>
<td>&quot;Screech&quot;</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Whale</td>
<td>true</td>
<td>&quot;[whale song]&quot;</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

Table 1

We could start out with the following class implementation (Mammal0.java2):

Let’s start our analysis:

- A mammal is defined by the fact that it gives milk to feed its young. It is thus not surprising that all the givesMilk() methods in the subclasses return true. The givesMilk() method is a prime candidate for "hoisting" up into the Mammal superclass ("hoisting" = moving the method upwards from the subclass to the superclass).
- makeSound() returns a different value for each species, but intrinsically, we expect any animal, which includes mammals, to be able to make some sort of sound. Thus Mammals should have a makeSound() method, but since, at the Mammals level, we don’t know exactly how that sound will be made, the method at that level must be abstract. The makeSound() method at the concrete Cat, Monkey and Whale level however, would be concrete because each animal makes its own unique sound.

Figure 2: No common methods defined in the superclass.

NOTE: Italics signify abstract methods or classes

NOTE: return_value : method_name(parameter_type_#1 parameter_name_#1, parameter_type_#2 parameter_name_#2, etc)

2http://cnx.org/content/m11796/latest/Mammal0.java
• `givesLiveBirth()` returns exactly the same value for all of our rather diverse selection of animals here. It seems like a fine candidate for hoisting. Or is it...? Let’s go ahead and hoist it anyway.

This is what we have so far (`Mammal1.java³`):

![Model of Mammals](http://cnx.org/content/m11796/latest/Mammal0.java)

**Figure 3:** Abstract and common methods hoisted to the superclass.

Before we go charging ahead, let’s stop for a moment and review what we’ve done: Cats, monkeys, and whales do represent a wide spectrum of mammals, but remember, the abstract `Mammal` class is a representation of **ALL** mammals, not just the ones we have so far. The correlation of like behavior with all our represented animals does not imply its inclusion in their abstract representation!

For instance, one day, in our wanderings through Australia, we encounter a Duckbilled Platypus⁴. Let’s see how it behaves with respect to our 3 methods:

<table>
<thead>
<tr>
<th>Mammal</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>boolean givesMilk()</code></td>
</tr>
<tr>
<td>Duckbilled Platypus</td>
<td>true</td>
</tr>
</tbody>
</table>

**Table 2**

**Duckbilled platypus lay eggs!!**

Giving live birth is not part of the definition of a mammal. On the other hand, the question of whether or not the animal gives live birth can always be asked of any animal, including all mammals. The result may be true or false however, so the method must be abstract at the `Mammal` level.

Our class structure should look like this (`Mammal2.java⁵`):

³http://cnx.org/content/m11796/latest/Mammal0.java
⁵http://cnx.org/content/m11796/latest/Mammal0.java

http://cnx.org/content/m11796/1.11/
Model of Mammals

![Model of Mammals Diagram]

Figure 4: Properly abstracted model.

Hoisting does not guarantee proper abstraction. Hoisting should be driven by a need for abstraction, not by coincidence.

Another key notion that the union pattern emphasizes is levels of abstraction. What we see is that the concept of a liquid is more abstract than milk. In turn, the general concept of milk is more abstract than "2% milk" or "skim milk" which would be subclasses of milk. In general we can see that a superclass is a distinctly higher level of abstraction than any of its subclasses. One of the key tools we use to help us design and build high quality object-oriented systems is careful attention to the abstraction level at any given moment.

Good OOP code always maintains a consistent level of abstraction.

Abstraction levels are links in a chain. A chain is only as strong as its weakest link. A program is only as abstract as its lowest abstraction level.

Levels of abstraction illustrate another important aspect of an OO program. Since a superclass represents the union of the subclasses or conversely, that the superclass can be represented by any of its subclasses, we see that the superclass is an embodiment of all the invariant aspects of the subclasses. That is, the superclass's definition is all that is abstractly equivalent about the subclasses—all that does not change from subclass to subclass. Note that this does not imply that the values of common fields are necessarily the same, just that, perhaps, that the field exists. Not does it imply that what is common to all the subclasses is necessarily what is abstractly equivalent about them (see the note above). The differences between the subclasses is what creates the variations in how the program behaves when any given subclass is used in place of the superclass. We call this the variant aspects of the system.

The total behavior of a program is the combination of its variant and invariant behaviors.

2 Inheritance and Polymorphism

Inheritance and polymorphism are really just two ways of looking at the same class relationship.

Inheritance is looking at the class hierarchy from the bottom up. A subclass inherits behaviors and attributes from its superclass. A subclass automatically possesses certain behaviors and/or attributes simply
because it is classified as being a subclass of an entity that possesses those behaviors and/or attributes. That is, a cherry can be said to automatically contain a seed because it is a subclass of Fruit and all fruit contain seeds.

Inheritance is useful from a code reuse perspective. Any (non-private) code in the superclass does not have to be replicated in any of the subclasses because they will automatically inherit those behaviors and attributes. However, one must be very careful when transferring common code from the subclasses to the superclass (a process called "hoisting"), as the proper abstraction represented by the superclass may be broken (see note above).

Polymorphism, on the other hand, is looking at the class hierarchy from the top down. Any subclass can be used anywhere the superclass is needed because the subclasses are all abstractly equivalent to the superclass. Different behaviors may arise because the subclasses may all have different implementations of the abstract behaviors defined in the superclass. For instance, all liquids have a boiling temperature. They may have different values for that boiling temperature which leads to different behaviors at any given temperature.

Polymorphism is arguably the more useful perspective in an object-oriented programming paradigm. Polymorphism describes how an entity of a lower abstraction level can be substituted for an entity of a higher abstraction level and in the process, change the overall behavior of the original system. This will be the cornerstone that enables us to build OO systems that are flexible, extensible, robust and correct.

### 3 Exploring Polymorphism

Let's explore some different ways in which polymorphism presents itself. Consider the following example of the union design pattern:

```java
/*
 * An interface that represents an operation on two doubles
 */
public interface IBinaryOp {
    double apply( double x, double y); // all interface methods are public and abstract by default
}

/*
 * An IBinaryOp that adds two numbers
 */
public class AddOp implements IBinaryOp {
    public double apply( double x, double y) {
        return x+y;
    }
}

/*
 * An IBinaryOp that multiplies two numbers
 */
public class MultOp implements IBinaryOp {
    public double apply( double x, double y) {
        return x*y;
    }
    public String getDescription() {
        return "MultOp is a multiplying function.";
    }
}
http://cnx.org/content/m11796/1.11/
```
Exercise 1
Is the following legal code? `IBinaryOp bop = new IBinaryOp();`

(Solution on p. 9.)

Exercise 2
Is the following legal code? `IBinaryOp bop = new AddOp();`

(Solution on p. 9.)

Exercise 3
Given the above declaration and assignment of `bop`, is the following assignment then possible? `bop = new MultOp();`

(Solution on p. 9.)

Exercise 4
Suppose we have `bop = new AddOp();`, what is the result of `bop.apply(5,3)`?

(Solution on p. 9.)

Exercise 5
Suppose we now say `bop = new MultOp();`, what is the result of `bop.apply(5,3)` now?

(Solution on p. 9.)

Exercise 6
Suppose we have some variable, called `myOp` of type `IBinaryOp` what is the result of `myOp.apply(5,3)`?

(Solution on p. 9.)

Exercise 7
Suppose we have `bop = new MultOp();`, is it legal to call `bop.getDescription()`?

(Solution on p. 9.)

Exercise 8
Is the following legal code? `AddOp aop = new AddOp();`

(Solution on p. 9.)

Exercise 9
Given the declaration in the previous exercise, is the following legal? `aop = new MultOp()`

(Solution on p. 9.)

Exercise 10
Suppose we have definitions of `aop` and `bop` from above. Is the following legal? That is, can we compile and run the following statement without error? `bop = aop;`

(Solution on p. 9.)

Exercise 11
Is the converse legal as well? That is, using the above definitions, can we compile and run the following statement? `aop = bop;`
Solutions to Exercises in this Module

Solution to Exercise (p. 8)
No, it won’t compile. IBinaryOp is an interface and does not specify any actual executable code, so it cannot be instantiated.

Solution to Exercise (p. 8)
Yes! AddOp is an concrete class and can be instantiated. AddOp is an IBinaryOp (technically, AddOp implements the IBinaryOp interface), so bop can reference it.

Solution to Exercise (p. 8)
Yes, for the same reasons as the previous exercise! MultOp is an concrete class and can be instantiated. MultOp is an IBinaryOp, so bop can reference it.

Solution to Exercise (p. 8)
The result is 8 because bop refers to an AddOp instance, whose apply method adds its two input values.

Solution to Exercise (p. 8)
The result is 15 because bop now refers to a MultOp instance, whose apply method multiplies its two input values.

Solution to Exercise (p. 8)
It is impossible to tell because it depends on the exact type of the object instance to which myOp refers.

Solution to Exercise (p. 8)
No, because bop is a variable of type IBinaryOp, which is not defined as having a getDescription method. This is true even if bop references an object of type MultOp. This is because the static typing of bop tells the compiler that it references an IBinaryOp, not the particular concrete type of the object it currently references. If we had MultOp mop = new MultOp(), then mop.getDescription() is perfectly legal.

Solution to Exercise (p. 8)
Yes, because aop is a variable of type AddOp, and thus can reference an instance of the same type.

Solution to Exercise (p. 8)
No, because aop is a variable of type AddOp, and MultOp is not an AddOp, so aop cannot reference an instance of MultOp.

Solution to Exercise (p. 8)
Yes! bop is a variable of type IBinaryOp, and aop is defined as referencing an AddOp object, which is an IBinaryOp.

Solution to Exercise (p. 8)
Not as written, because bop is a variable of type IBinaryOp (i.e. statically typed as such), which and does not necessarily reference an object of type AddOp, to which aop must reference. That is, a variable of the type of the superclasses can always reference an object of the type of any subclass, but a variable of the type of a particular subclass cannot necessarily reference something of the type of its superclass. Another way of saying this is that a superset contains its subsets but not the other way around. The above assignment will cause a compile time error because the compiler cannot know if the assignment is possible. An explicit cast is required to suppress the compile time error (aop = (AddOp) bop), but may trigger a run-time error if indeed the instance referred to by bop is not of type AddOp.