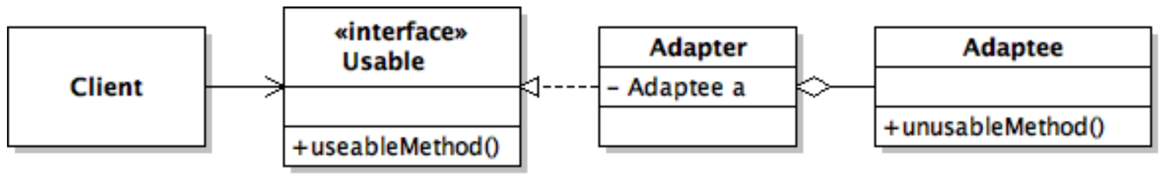
CECS 277 – Lecture 23 – Design Patterns

Adapter

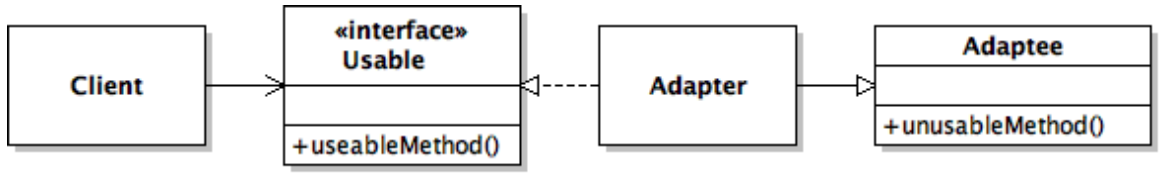
The Adapter pattern is a Structural design pattern that allows objects with incompatible interfaces to work together. The incompatible class may be from an outside source, like a library or another part of the program, but the code works and you don't want to change it, rewrite it, or copy it over. The incompatible class might have different method names, parameter lists, return types, or the value returned may be different than what is expected. In the client, you want to preserve the polymorphic behaviors your interface has, you don't want to have to construct the incompatible object separately and have to remember all the different methods for just that one object. An Adapter can be wrapped around the incompatible object (or class) in order to hide the conversions that are necessary to get it to work the way that it is expected.

Adapter UML – There are two versions of the Adapter pattern: Object Adapter and Class Adapter. An Object Adapter has an instance of the incompatible Adaptee class, and a Class Adapter inherits from the Adaptee class. The Usable interface (or class) provides the methods that are available to the Client. The Client is the code that requires the use of the incompatible Adaptee methods.

Object Adapter:



Class Adapter:



Adapter Template Code:

```
public class Adaptee {
   public void unusableMethod( ) {
      System.out.println("Unusable");
   }
}
public interface Usable {
   public void usableMethod( );
}
public class Client{
   public static void main( String [] args ) {
      Usable u = new Adapter( );
      u.usableMethod( );
   }
}
```

Object Adapter:

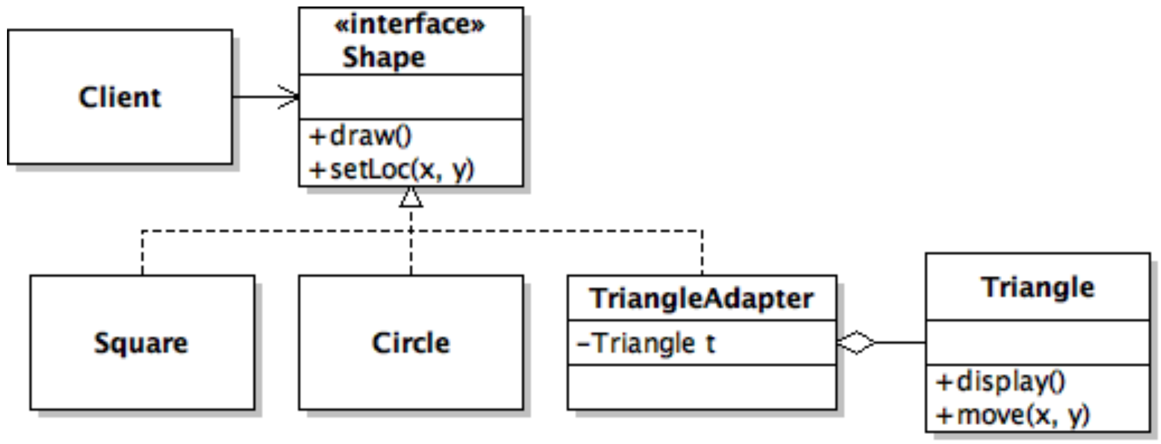
```
public class Adapter implements Usable {
   private Adaptee a = new Adaptee( );
   @Override public void usableMethod( ) {
      a.unusableMethod( );
   }
}
```

Class Adapter:

```
public class Adapter extends Adaptee implements Usable {
   @Override public void usableMethod( ) {
      unusableMethod( );
   }
}
```

Adapter Example:

In this example, the Client is already using the interface Shape. The Shape interface had two implementing classes, Square and Circle, that overrode the functions draw() and setLoc(). Your boss has asked you to add a Triangle to the program. You could create a new Triangle class that extends the Shape class, but then you'd have to write and test a lot of new code. You find out that someone on another team has already written a Triangle class for a similar purpose, unfortunately, you are not allowed to change the code or copy it, so you decide to create an Adapter for it. Now the Client may use triangles the same way it used Squares and Circles before.



```
public class TriangleAdapter implements Shape {
   private Triangle t = new Triangle( );
   @Override
   public void draw( ) {
      t.display( );
   }
   @Override
   public void setLoc( int x, int y ) {
      t.move( x, y );
   }
}
```