GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computing & Mathematics

Composition v/s Inheritance

One of the elementary requirements of an object-oriented language is the ability to create objects that are capable of interacting with one another. The whole point of object interaction is to *reuse existing state and behaviour* that has already been well abstracted and encapsulated in a different class. The form of reuse is determined by the relationship between two objects and is typically categorised on the basis of the ***has-a*** or ***is-a*** rules, implying either one of the four forms of composition, or one of the two forms of inheritance respectively. It is important to understand that composition and inheritance are not mutually exclusive and that an application may indeed maintain both *has-a* and *is-a* relationships simultaneously. However, the established best practice is to follow the principle of ***favouring composition over inheritance where possible***. This does not imply that inheritance should be completely discarded, as inherited abstractions provide the basis of loose coupling in an application. Rather, it means that discernment should be applied in choosing how and where inheritance is applied in an application.

The Limitations of Inheritance

Both implementation and specification inheritance confer the ability to exploit dynamic binding and polymorphism in an application. As inheritance is predicated on the pre-existence of abstractions, such as abstract classes or interfaces, it forms the basis of loose coupling and makes an application less resistant to change. Because an inheritance relationship is declared at a class level, it *statically binds a class to a hierarchy of other classes at compile time*. Furthermore, as an inheritance relationship forms part of the signature of a class, ***the is-a relationship is both public and structural***.

However, if an inheritance relationship is not thought through correctly, i.e. if abstractions in an inheritance hierarchy are not cohesive with respect to one another or are in violation of the Single Responsibility Principle (SRP), then inheritance ***can actually weaken encapsulation*** and give rise to numerous problems, including fragile super-classes. A fragile super-class causes cascade effects when changed and thus induces into a design a resistance to any change. In addition to impacting subclasses, cascades may adversely affect any class that utilises the public interface, i.e. methods, defined by the superclass. In the context of specification inheritance, a fragile super-type will guarantee that multiple parts of an application will regularly break simultaneously. The root cause of such fragility is the violation of the SRP, usually by compromising minimalism during design time.

Benefits and Limitations of Composition

Existing state and behaviour can also be re-used through the various forms of composition – dependencies, associations, aggregations and full composition. Composition promotes reuse by enabling ***responsibilities to be dynamically added*** to an object at run-time. Using composition as the basis of reuse promotes classes that are ***less resistant to change by providing a greater level of encapsulation*** than inheritance. External dependencies will not

necessarily be affected if a composite object is changed, provided that a class continues to expose the same interface, i.e. the same methods with the same method signatures.

It is also ***easier to change a class that contains a composite object*** than it is to change a subclass. The corollary of a fragile superclass is a rigid subclass. Any change to the interface of a subclass must be consistent with the interface exposed by a superclass. For example, it is not possible to add a new method to a subtype that contains the same method signature of a super-type but with a different return type. Composition also ***allows object creation to be delayed*** until it is actually needed and provides a capability to create composite objects dynamically during the lifetime of the containing object. Using inheritance, when an object is instantiated, constructor chaining will result in all super-classes in the object hierarchy being created before the subclass in placed in memory.

Composition however, does not provide the ability to directly harness the power of ***polymorphism***. Furthermore, there is an ***overhead associated with the forwarding of method invocations*** in a composite relationship, whether this is done using delegation, aggregation or full composition.

Combining Inheritance and Composition

Where a case for reuse mandates a form of composition, specification inheritance can also be applied to promote loose coupling and to facilitate polymorphic behaviour. If a composed object has been abstracted and encapsulated correctly, it should have a single responsibility and form part of a cohesive inheritance hierarchy of related responsibilities. A container class can combine inheritance and polymorphism by declaring itself to implement an interface, composing itself with a class that already implement that interface and then delegating the implementation of the interface methods to the composed class. In such a case, the container class maintains both a *has-a* and *is-a* relationship with the interface supertype and can be treated uniformly with any other instance of the interface, including the composed type. Indeed, by mixing composition and inheritance in this manner, a class in a single inheritance language can achieve the same degree of reuse provided by languages that support multiple implementation inheritance, without any of the ambiguity induced by the latter.

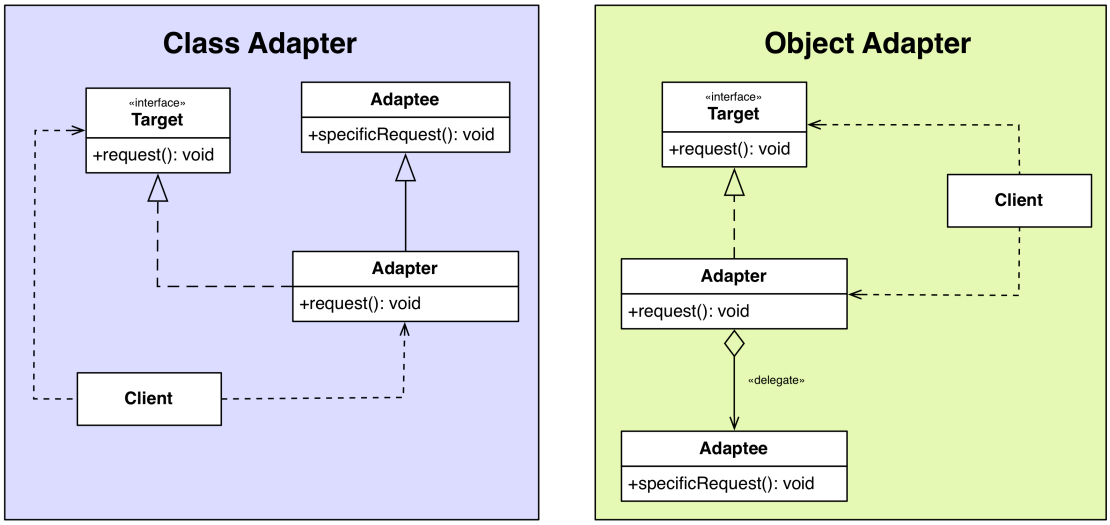
Adapters – How to Put a Square Peg in a Round Hole

If an inheritance hierarchy is designed correctly, each derived class should have a single responsibility that is more refined and specific than its super-type. This does raise the issue of how responsibilities implemented in different hierarchies can be shared and loosely coupled. Consider an inheritance hierarchy, *S*, with 10 different classes and a second hierarchy, *T*, with 50 different classes. The total number of paired responsibilities is the ***Cartesian product*** of the two hierarchies, i.e. 10 x 50 = 500. Using direct inheritance to reuse such functionality will result in a Cartesian product of classes, i.e. a class explosion, leading a highly complex and unstable design. There are three design patterns that can be applied to such a context, all of which combine composition with inheritance to provide ***maximum re-usability and stability*** to a design. They are ***adapters***, ***decorators*** and ***bridges***. Each of these has different intents and solutions, but all are based on the principle of applying composition to reuse the functionality of one or more hierarchies.

Sometimes it is necessary to have unrelated classes, i.e. classes from different inheritance hierarchies, coupled together in a single class. Such an approach is fine, unless the consuming client class has a dependency that may be volatile, i.e. the coupled class is required to be changed from time to time. Each potential axis of change represented by the dependency then becomes a reason to change the client class, as the public methods exposed by the different dependencies may be different. Changing a client class in such as context is a breach of the Open-Closed Principle (OCP). Given such a context, one solution is to apply the ***adapter***

pattern to the problem. *An adapter converts the interface of a class into the interface that a client class expects and enables classes with incompatible interfaces to work together.* An adapter decouples a client class from a volatile dependency by exposing an interface and then encapsulating any change internally. This implies that a client class does not have to be modified if a dependency changes over time, as the dependency is encapsulated inside an adapter. The relationship between the client class and the original dependency becomes loosely coupled, provides greater flexibility and stability and also upholds the OCP.

An adapter can be implemented as either a ***class or object adapter***, with the former relying on inheritance for reuse and the latter on composition. Understanding the benefits and limitations of both approaches illustrate why the design principle of ***favouring composition over inheritance*** is good programming practice. Consider the UML diagram below, where the *Client* class represents a class with a dependency that changes (and *Adaptee*). The *Adapter* class plays the same role as an electrical adapter and ***places a layer of abstraction*** between two class hierarchies.



Using over-riding, a class adapter can modify some of the adapted class methods and also enable other inherited methods to be used. It does however suffer from the drawback that, unless a base class is specified, it cannot work for a class and all of its subclasses. Furthermore, a new *Adapter* class will be necessary if the *Client* class requires behaviour from a different hierarchy of *Adaptees*. The object adapter does not suffer from the same limitations because subclasses from one or more hierarchies can be adapted by passing an instance to the constructor of the *Adapter*. Note that the instance of *Adaptee* can be deferred until runtime, in contrast with the behaviour of the class adaptor, which is fixed at compile-time.

Exercises

In this lab, we will illustrate the advantages of composition over inheritance approach using a *java.util.ArrayList* class as the basis for creating a stack. We will then create class and object adapter to see how the principle of ***favouring composition over inheritance*** really is good programming practice.

Stack Example

A stack is an ordered list in which all insertions and deletions are made at one end, called the top (FIFO). We will define our stack with the following interface:

Method Signature	Description
void push(E o)	Insert object O of type E to the top of the stack
E pop()	Return and remove most recently inserted element
E top()	Return most recently inserted element
int size()	Return the number of elements in the stack
boolean isEmpty()	Return true if the stack is empty
void clear()	Remove all elements

1. Create a project in Eclipse called ***Stack*** and a package called ***ie.gmit.sw***.
2. Declare the above behaviour in an interface called *Stackable*.
3. Implement the *Stackable* interface by sub-classing ***java.util.ArrayList***. The class signature should be as follows:
   ```
   public class InheritanceStack<E> extends ArrayList<E> implements Stackable<E>
   ```
4. Implement the Stack interface by creating a class that composes ***java.util.ArrayList***. The class signature should be as follows:
   ```
   public class ComposedStack<E> implements Stackable <E>{
   private List<E> list = new ArrayList<E>();
   …
   }
   ```
5. Create a class called ***Stacker*** that instantiates both implementations of *Stackable*.
6. Use the "intellisense" feature of Eclipse to check the behaviour of both classes. *Explain how is the LSP and SRP upheld in both implementations?*

Adapter Example

- Create a project in Eclipse called ***Adapter*** and a package called ***ie.gmit.sw***.
- Create an interface called ***ListWriterable*** that declares the following methods:
  ```
  public void open(String filaName);
  public void write(String[] list);
  public void close();
  ```
- Create a class called ***SimpleStringWriter*** defined as follows:
  ```
  package ie.gmit.sw;

  import java.io.*;
  public class SimpleStringWriter {
    private FileWriter fw;

    public void open(String fileName){
       try{
          fw = new FileWriter(fileName);
       }catch(IOException ioe){
          ioe.printStackTrace();
       }
    }

    public void write(String s){
       try{
          fw.write(s + "\n");
       }catch(IOException ioe){
          ioe.printStackTrace();
       }
    }

    public void close(){
  ```

```
            try{
              fw.close();
            }catch(IOException ioe){
              ioe.printStackTrace();
            }
          }
        }
```

- Create a class called ***ListClient*** with a main method defined as follows:

```
public static void main(String[] args){
    String[] names = {"John", "Paul", "Mary", "Anne", "Alan", "Barbara"};
    ListWriterable writer = null; //We do not have a conforming class yet
}
```

- Create a class adapter called ***ArrayStringWriter*** that subclasses *SimpleStringWriter* and implements the *ListWriterable* interface. Implement the abstract *write(String[] list)* method by iterating over the *String* array passed as a parameter and passing each String to the inherited *write(String s)* method.

- Change the main method of ***ListClient*** to use an instance of ***ArrayStringWriter*** and examine the output. *How is the client coupled to ArrayStringWriter? What classes does the client need to compile and what would be the effect of changing the type of ListWriterable being used?*

- Create an object adapter called ***CompArrayStringWriter*** that implements the *ListWriterable* interface and is composed with the class *SimpleStringWriter*. Implement the abstract *write(String[] list)* method by iterating over the *String* array passed as a parameter and delegating each String to the *write(String s)* method of *SimpleStringWriter*.

- Change the main method of ***ListClient*** to use an instance of ***CompArrayStringWriter*** and examine the output. *How is the client coupled to CompArrayStringWriter? What classes does the client need to compile and what would be the effect of changing the type of ListWriterable being used?*