Department of Computer Science & Applied Physics

Multiple Inheritance and Polymorphic Service Providers

Support for the declaration of an abstract data type in the object-oriented paradigm implies that some mechanism must be available to create a concrete realisation of abstract behaviour. Inheritance allows the reuse of existing state and behaviour by coupling a class to other types as part of the class definition. This is a ***compile-time activity*** and has major consequences for the design of loosely coupled object-oriented applications. Inheritance creates transitive relationships and dependencies between classes in an inheritance hierarchy or graph. The relationship between abstraction, inheritance and polymorphism is also transitive, as the latter two concepts are directly derived from the first. By definition, abstractions are highly reusable in a variety of different contexts. As classes become more specialised, their degree of reuse becomes more restricted. Inheritance can therefore be viewed as both a process of specialising abstract entities as concrete realisations and as a process of moving from the abstract to the specific. Inheritance underpins the concept of ***polymorphism*** or ***dynamic binding***, where a method implementation is determined at run-time by the specific instance of a type being used.

The Principle of Substitutability (PoS)

The fundamental mechanism of inheritance in object-oriented languages is governed by the Principle of Substitutability (**PoS**), also called the Liskov Substitution Principle (LSP) after Prof. Barbara Liskov, who published a number of seminal papers on type theory. Substitutability refers to the ability to dynamically assign to a variable of a given type an instance of another type, with ***no perceived difference in behaviour***. In Java and C#, the terms subclass and subtype are synonyms, as are the terms superclass and supertype. But what exactly is a subtype? Liskov answers this question with the following definition[1]:

If, for each object O_1 of type S there is an object O_2 of type T such that, for all programs P defined in terms of T, the behaviour of P is unchanged when O_1 is substituted for O_2, then S is a subtype of T.

This definition implies that methods which use references to base types must also be able to use references to derived types ***without knowing the difference***. Consequently, unless a class or method is marked as *final* or a class has been *sealed*, all method invocations are *potentially polymorphic*, i.e. the behaviour of a method depends upon the object instance that the method belongs to. Note that the phrase *"without knowing the difference"* does not imply that all methods declared in a type must be implemented the same way. Rather, this means that a method signature, i.e. the visibility modifier, return type, name and parameters, must conform to its original declaration. Liskov goes further to state that the public features of a class must be viewed in terms of the reasonable assumptions that will be made by users of that design, i.e. a method should do what a reasonably-minded programmer thinks it should do from its signature.

Violating the PoS will require a class referencing a non-PoS conforming class to know about every derivative of that base class. Consider the example of a circle and an ellipse. A circle

[1] Liskov, B., *Data Abstraction and Hierarchy,* Journal of the ACM, Vol. 23, Issue 5, pp. 17-34, 1988.

contains a single focus (x/y coordinate) and a radius. An ellipse is a set of two foci (x_1/y_1 and x_2/y_2 coordinates) surrounded by a curve, where the sum of the distances between the foci and the curve is constant for every point on the curve. Mathematically, a circle is just a specialised ellipse (a subclass of an ellipse) where the foci have the same x and y values. From a mathematician's perspective, a circle ***is-an*** ellipse. If this formulation is implemented as a class *Ellipse*, the expected behaviour is to have $setF_1(x_1, y_1)$ *and* $setF_2(x_2, y_2)$ methods. However, a subtype of *Ellipse* called *Circle* will inherit the two methods, even though a circle has only a single focus. This clearly violates the PoS as there is no way to specify which method should be called to specify the focus and what the class *Circle* does if the two foci have different x and y coordinates. A class conforming to the PoS will specify the *Circle* as the base type and the *Ellipse* as the derived type, thus eliminating the ambiguity and ensuring that each class is used correctly. The PoS therefore should force us to consider carefully how we abstract behaviour to promote re-usability through inheritance. Both specification and implementation inheritance allow for the creation of subtypes and each subtype should uphold the PoS.

Specification Inheritance

A class definition can be abstracted to the point where it contains no implementation of state and behaviour, but only the declaration of method signatures. In C++, this form of declarative programming is possible through the use of *pure virtual functions* in an abstract class. More modern object-oriented languages have a more explicit mechanism for declaring a purely abstract entity through the built-in support for interfaces. Languages, such as Java and C#, make a clear distinction between interfaces and abstract classes, as the semantics of each are very different. An interface definition creates an abstract data type that can be used in class, method and variable declarations. As interfaces are purely abstract entities, their use in these contexts induces a high degree of loose coupling in a design and promotes polymorphic behaviour.

An interface definition creates an abstract type with a declared set of abstract, default, private or static methods, all of which must be present in any implementing class. Interfaces, as declarative entities, convey ***a role that a class can play*** and leave the implementation of that role to the class. The interface name and its methods (abstract, default and static) form a ***contract***, with a set of preconditions, postconditions and invariants, between the provider of the interface and a user. As interfaces define behaviour that ***must be fully implemented*** by a derived type, changing an interface without retro-fitting with default methods is guaranteed to break the code of every derived type that ever used it. Consequently, interfaces should be considered ***immutable*** and should not be changed after they have been published. It is therefore of paramount importance that interfaces are abstracted correctly with due regard to minimalism. In particular, an interface definition should uphold the ***Interface Segregation Principle (ISP):*** *Users of an interface should not be forced to depend on methods that they do not use*. A violation of this principle only arises because the *Single Responsibility Principle* has already been breached. An interface should declare a cohesive and minimalist set of behaviours, all of which are required by every subtype. The *"if in doubt, leave it out"* mantra works well in this context. It is far better to omit required behaviour that can be added later (through inheritance or default methods) than forcing unwanted responsibilities on every concrete realisation of an interface. Note that, at a more general level, any non-private method of any class can be argued to constitute an interface, although the use of the term interface in this wider context has different semantics to that of a purely abstract type.

A class can implement more than one interface, with each interface denoting a different role that the class can play. Specification inheritance is therefore a form of ***multiple inheritance and creates a dependency graph*** of both classes and objects. By over-riding inherited default methods in derived types, multiple interface inheritance can lead to ambiguities and complexity, including the ***Deadly Diamond of Death*** (DDoD). Multiple interface inheritance also gives rise to the question of how a class can play multiple roles, i.e. have multiple responsibilities, without violating the *Single Responsibility Principle*. This can be achieved through the use of composition and delegation and, where appropriate, a ***dependency injection***. For each declared role, a class should compose itself with another class that already encapsulates that role, ideally implementing the same interface, and then delegate calls from an interface-defined method to the composed class. The container class should therefore use the interface type in the variable declaration. In this manner a class can simultaneously support ***has-a*** and ***is-a*** relationships with the same type.

Interfaces themselves can inherit responsibilities from other interfaces through subclassing and, unlike implementation inheritance in most programming languages, ***can inherit from more than one supertype***. For example, an interface *A* can extend interfaces *B*, *C* and *D*. Any concrete realisation of interface *A* will be forced to implement all the declared abstract methods in *A*, *B*, *C* and *D*. Defining interfaces in this way is uncommon as the semantics of such use implies that the derived interface is combining multiple roles into a single set of aggregated roles. Such an approach is difficult to justify as not being in breach of the Interface Segregation Principle.

Specialisation Inheritance

In contrast with specification inheritance, which requires a class to implement a set of completely abstract or default methods, specialisation inheritance enables the ***reuse of existing state and behaviour*** derived from a supertype. By subclassing an existing class, the derived type represents a specialisation or extension of the supertype or base type. A derived type refines or extends the responsibility of a supertype in some way and is therefore ***permanently bound at compile time to a specific position in a hierarchy of cohesive responsibilities***. The exact position of a class in a type hierarchy is determined by the Single Responsibility Principle. In languages that support single specialisation inheritance, the hierarchy is always a tree structure that should be balanced and have a low tree depth. An imbalance implies that one or more derived types have been over-abstracted, requiring subtypes to implement behaviour that should already have been present in the hierarchy. In languages that support multiple specialisation inheritance, the hierarchy is more complex and forms a graph of responsibilities.

If the intent of a derived type is extension, then the derived type should declare and implement a set of ***additional methods*** that match the extended responsibility. If the intent is refinement, the derived type should ***over-ride***, either partially or completely, one or more methods of the superclass. The intent can also be mixed, with a derived type both adding additional behaviours and refining existing behaviours through over-riding. As only concrete classes can directly be instantiated as objects, the invocation of a method that has been over-ridden more than once in a hierarchy ***always results in the most specific method being called***.

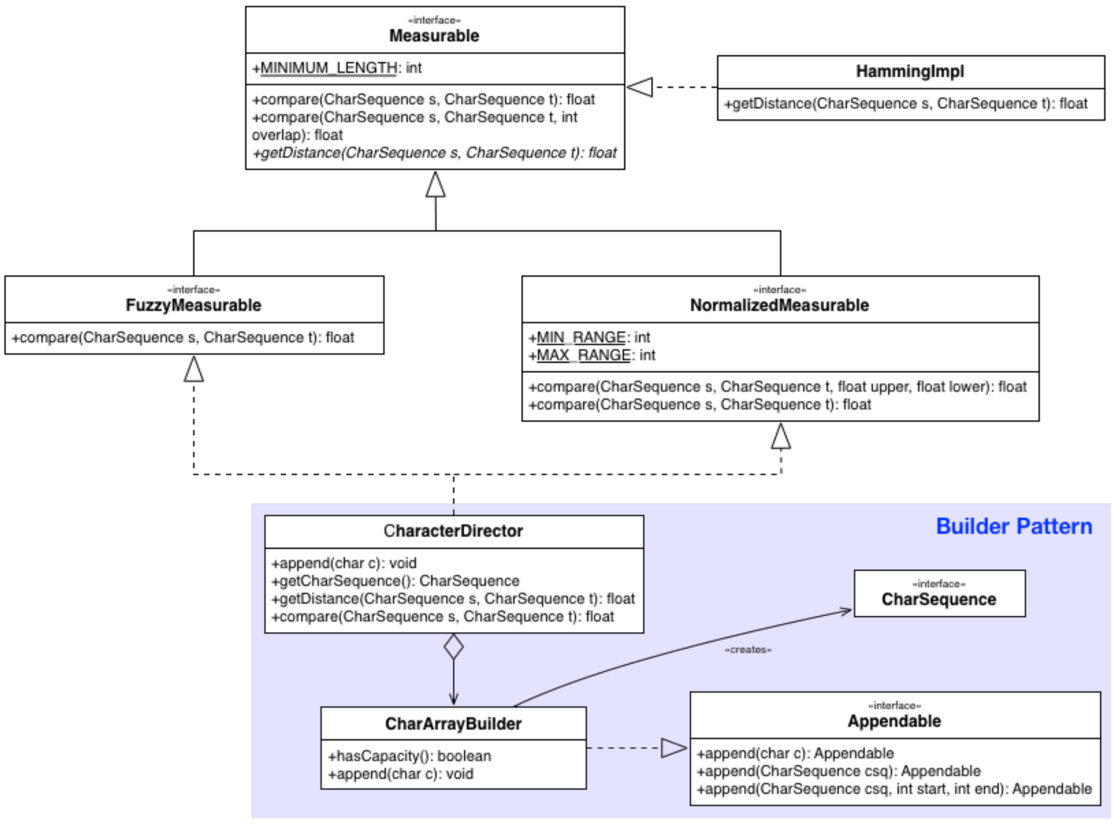
At the root of a tree of responsibilities, a base class should implement the minimal state and behaviour universal to all possible derived types. When designing a base class, there will be some responsibilities that can be implemented fully and others that can be declared, but whose implementation must be deferred to specific usages of derived types. The latter may be implemented as ***abstract methods*** and are declared in a similar way to interface methods. Abstract methods form a template that all derived concrete types must implement fully. Therefore, an abstract method can be safely invoked from an implemented method in an abstract class, as the abstract method is guaranteed to be implemented in any concrete realisation of the abstract type. If a class declares an abstract method, the class itself must be marked as abstract. It is good design approach to ensure that an abstract class forms the base of a hierarchy of responsibilities and that those responsibilities are defined in an interface that the abstract class itself implements. Note that it is possible to create an abstract class *A* with a concrete subtype *B* that is extended in turn by an abstract class *C*. Such a hierarchy is semantically nonsensical as it is inherently inane to move from the abstract to the specific to the abstract.

Sometimes the motivation for using specialisation inheritance is not to create an ***unbounded*** hierarchy of reusable abstractions, but to **model the full set of types** that exist in a domain. In such situations, ***sealed types*** allow the restriction of an inheritance hierarchy into a ***bounded*** suite of enumerated types.

While specification and specialisation inheritance are clearly distinct, they are not mutually exclusive. For example, interfaces can provide some implementation of universal behaviour in the form of default or static methods. Indeed, a good design will enable an encapsulated set of roles, defined by a hierarchy that ranges from the general to the specific, to be reused through a combination of specification and implementation inheritance. In this context, interfaces can be used to act as a bridge between different inheritance hierarchies, enabling functionality to be extended and reused in a loosely coupled manner throughout an application.

Exercises

In this practical we will use specification inheritance to create the ***Deadly Diamond of Death (DDoD)*** and see how its ambiguities can be resolved. In addition, we will create and declare a service interface and a ***polymorphic service provider*** using the Java Platform Module System (JPMS). Finally, we will implement the ***Builder Pattern*** by consuming the classes in the DDoD. An overview of the system components is shown below:


«interface»
Measurable
+MINIMUM_LENGTH: int
+compare(CharSequence s, CharSequence t): float
+compare(CharSequence s, CharSequence t, int overlap): float
+getDistance(CharSequence s, CharSequence t): float
HammingImpl
+getDistance(CharSequence s, CharSequence t): float
«interface»
FuzzyMeasurable
+compare(CharSequence s, CharSequence t): float
«interface»
NormalizedMeasurable
+MIN_RANGE: int
+MAX_RANGE: int
+compare(CharSequence s, CharSequence t, float upper, float lower): float
+compare(CharSequence s, CharSequence t): float
Builder Pattern
CharacterDirector
+append(char c): void
+getCharSequence(): CharSequence
+getDistance(CharSequence s, CharSequence t): float
+compare(CharSequence s, CharSequence t): float
«interface»
CharSequence
«creates»
CharArrayBuilder
+hasCapacity(): boolean
+append(char c): void
«interface»
Appendable
+append(char c): Appendable
+append(CharSequence csq): Appendable
+append(CharSequence csq, int start, int end): Appendable

- ***Download*** the resources for the lab from Moodle and unpack the content of the Zip archive into the following two projects as shown below. You will need to ensure that any required modules are on the ***Module Path*** of the project that depends on them.

Project Name	Module Name	Package Name
CharSequenceMetrics	atu.character.service	ie.atu.character
Measurable Levenshtein • The module descriptor should export the package ***ie.atu.character***. • Export the class files into the project directory as **measurable.jar**.		

Project Name	**Module Name**	**Package Name**
HammingMetrics	atu.character.metrics	ie.atu.sw.hamming
FuzzyMeasurable NormalizedMeasurable HammingImpl • The module descriptor should require the module **atu.character.service** and export the package **ie.atu.sw.hamming**. • Export the class files into the project directory as **hamming.jar.**		

- The interface ***Measurable*** is a service interface and is implemented in the *atu.character.service* module by the class ***Levenshtein***. Declare the service provider as follows in *module-info.java* and then rebuild the JAR archive ***measurable.jar***.

```
module atu.character.service {
  exports ie.atu.character;
  provides ie.atu.character.Measurable with ie.atu.character.Levenshtein;
}
```

  The declaration *provides..with* creates a service provider interface (SPI) for ***Measurable***. *There can be only one SPI per module (Highlander Principle)*.

- The class ***HammingImpl*** is also a service provider of ***Measurable*** and should be declared in the module descriptor of the project ***HammingMetrics*** as follows:

```
module atu.character.metrics {
  requires  atu.character.service;
  exports ie.atu.sw.hamming;
  provides ie.atu.character.Measurable with ie.atu.sw.hamming.HammingImpl
}
```

  Rebuild the JAR archive ***hamming.jar***.

In the next part of the practical, we will create the ***DDoD*** by implementing both ***FuzzyMeasurable*** and ***NormalizedMeasurable*** in a single class.

- Create a new project called ***CharacterBuilder*** with a module descriptor as follows. You will need to add both ***measurable.jar*** and ***hamming.jar*** to the *Module Path*:

```
module atu.character.builder {
  requires atu.character.service;
  requires atu.character.metrics;
  uses ie.atu.character.Measurable;
}
```

  The restricted keyword *uses* declares that this module is a consumer of the service interface ***Measurable***. Note how ***the implementation is missing*** from the declaration. We will use dynamic class loading to bind to any available service providers at runtime, i.e. polymorphism.

- Add a package called ***ie.atu.sw.builder*** to the project and then create a class called ***CharacterDirector*** as follows:

```
import java.util.ServiceLoader;
import ie.atu.character.*;
import ie.atu.sw.hamming.*;

public class CharacterDirector implements FuzzyMeasurable, NormalizedMeasurable{
  private Measurable measurable; //Compose
```

```
    public CharacterDirector() {
      loadSPI();
    }

    public CharacterDirector(Measurable  m) { //Dependency injection
      this.measurable = m;
    }

    private void loadSPI() { //Dependency injection
      ServiceLoader<Measurable> services = ServiceLoader.load(Measurable.class);
      Measurable m = services.iterator().next();
      if (m != null) this.measurable = m;
    }

    @Override
    public float getDistance(CharSequence s, CharSequence t) throws Exception {
      return measurable.getDistance(s, t); //Delegate
    }
  }
```

You should be presented with the error *"Duplicate default methods named compare with the parameters (CharSequence, CharSequence) and (CharSequence, CharSequence) are inherited from the types NormalizedMeasurable and FuzzyMeasurable"*, as the compiler cannot infer which of the inherited methods in the DDoD to inherit. To resolve the problem, we must explicitly declare the behaviour by over-riding the method again as follows:

```
    @Override
    public float compare(CharSequence s, CharSequence t) throws Exception {
      return FuzzyMeasurable.super.compare(s, t);
      //return NormalizedMeasurable.super.compare(s, t);
    }
```

We can replace, delegate and even combine supertype methods to get the behaviour we want, but we must define the behaviour explicitly to resolve the ambiguity induced by multiple inheritance.

- The class *java.util.ServiceLoader* dynamically loads all implementations of Measurable that are on the ***Module Path***. The service consumer is loosely coupled to the service provider and does not even know the name of concrete instance it will be using. *This is how a service consumer and a service provider are linked in the Java Platform Module System (JPMS)*. You can verify that all the implementations are available using a *for...in* loop as follows:

```
  for (Measurable me : services) {
    System.out.println(me.getClass().getName());
  }
```

In the final part of the practical, we will implement the Builder Pattern to create instances of the interface java.lang.CharSequence, a "complex" and multi-step operation. This interface is implemented by the classes String, StringBuilder and StringBuffer.

- Create a class called ***CharArrayBuilder*** as shown below.

```
  import java.io.IOException;
  public class CharArrayBuilder implements Appendable{
    private char[] sequence;
    private int index;

    public CharArrayBuilder() {
      this(8);
    }

    public CharArrayBuilder(int length) {
      this(new char[length]);
    }

    public CharArrayBuilder(char[] s) {
      this.sequence = s;
    }

    public char[] getSequence() {
      return sequence;
    }

    private void expand(){
      char[] t = new char[(sequence.length * 3) / 2]; //Amortized expansion
      for (int i = 0; i < sequence.length; i++){
        t[i] = sequence[i];
      }
      sequence = t;
    }

    @Override
    public Appendable append(char c) throws IOException {
      if (index >= sequence.length - 1) expand();

      sequence[index] = c;
      index++;
      return this;
    }

    @Override
    public Appendable append(CharSequence csq) throws IOException {
      return append(csq, 0, csq.length());
    }

    @Override
    public Appendable append(CharSequence csq, int start, int end) throws IOException {
      for (int i = start; i < end; i++) {
        append(csq.charAt(i));
      }
      return this;
    }
  }
```

The interface ***Appendable*** represents the abstract "Builder" interface and ***CharArrayBuilder*** the "ConcreteBuilder". The "Product" that the builder creates is a ***CharSequence***. Note that each concrete builder knows how to assemble different kinds of parts for different kinds of products, in this case characters. The class ***CharArrayBuilder*** encapsulates the way in which a concrete object, an instance of ***CharSequence***, is constructed and hides its internal representation, a char array in this case, from clients.

- Add an instance variable called ***builder*** of type ***CharArrayBuilder*** to the class ***CharacterDirector*** and then add the following two methods.

```
public void append(char c) throws Exception {
  builder.append(c);
}

public CharSequence getCharSequence() { //The product is a CharSequence
  return new String(builder.getSequence()).trim().intern();
}
```

- The director knows what needs to be built and the coarse steps for how to build it, but it does not know how to put individual parts together, i.e. how to assemble the set of characters into a ***CharSequence***. The director uses the ***Appendable*** interface to direct the ***CharArrayBuilder*** to put the various parts (characters) together.

- Create a class called ***Client*** with a *main()* method and test the application.