GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computing & Mathematics

Applying Abstraction and Encapsulation

The object-oriented paradigm is primarily concerned with the creation of software artefacts based on the principles of cohesion and coupling. Developers should strive to create classes and components that are both highly cohesive and loosely coupled. While the ideas of composition, inheritance and polymorphism are fundamental to object-oriented programming, the concepts of abstraction and encapsulation have primacy and form the taproot of the paradigm. Unfortunately, most texts on object-oriented programming give light treatment to abstraction and encapsulation and focus mainly on composition, inheritance and polymorphism. This is a grave mistake, as these other concepts only exist in the paradigm because of the capability of creating an abstract data type and re-using encapsulated functionality. The objective of this lab is to expand your understanding of abstraction and encapsulation though a set of practical exercises that should provoke thought and the questioning of assumptions.

Abstraction – Less is More

The whole notion of objects revolves around the capability of defining and instantiating abstract data types. Abstraction is first and foremost a thought process – factoring out the commonality of state and behaviour from a group of entities into a reusable abstract representation. Abstraction must be *minimalist* – only behaviours that are *universally common* belong in an abstraction. Including incidental details yields feature-laden classes at the top of an object hierarchy resulting in subtypes inheriting functionality they rarely use. The situation is compounded when using interface types, as incidental details place an absolute requirement on a developer to implement a method they may never wish to use.

The mechanics of abstraction depends on the programming language, but can be categorised into three elementary forms:

1. **Role Abstraction:** object behaviour is completely separated from object state in an absolute abstract type, i.e. an *interface*. In this context, the abstraction really defines a *role* that different objects can play in different contexts.

2. **Hierarchical Abstraction:** the base class of a hierarchy contains some implemented state and behaviour, mixed with some abstract methods, i.e. an *abstract class*. This form of abstraction is governed by the ***Single Responsibility Principle (SRP)*** and describes generic behaviour that can be applied to a whole tree of objects.

3. **Class Abstraction:** a concrete class at the base of a hierarchy represents an abstraction that can be utilised in many different contexts, reused though delegation and composition and extended through inheritance. For example, the *String* class defines methods relating to the manipulation of a sequence of characters. Although these methods can be refined or augmented though inheritance, the *String* class itself is a useful and reusable abstraction of a character sequence.

Regardless of the form of abstraction used, the methods, and possibly state, employed should be determined by the SRP. Violating this principle will result in ***a lower level of cohesion*** in a design and the creation of feature-laden and *God* classes, either of which undermines the design goals of an object-oriented application. As abstraction is minimalist, the resultant abstract data types are unlikely to have to change and are highly stable. Consequently, ***a good design will use abstractions as much as possible*** throughout an application, including as method parameters and return types. Using abstractions promotes a loosely coupled design, enabling different implementations of an abstract type to be swapped at run-time, producing software that is both polymorphic and highly adaptable to change.

It should be understood that the concept of ***abstraction scales in a software system***, from fine-grained class design, to the interplay of whole components in an application, to the interaction of applications at an architectural level. If interactions are restricted to abstract types where possible, dynamic run-time configuration and deployment of architectural artefacts can be achieved in much the same way as at a class level.

Encapsulation – Dealing with Known Unknowns

The word *encapsulate* literally means to "surround in a capsule", a definition apposite to its use in the context of object-oriented programming. The motivation for encapsulation is to promote reuse by surrounding commonly used behaviour (methods) and state (data structures) in a capsule (class). Encapsulation works hand-in-glove with abstraction and underpins the most commonly used approach to object reuse – composition. Encapsulation is often described as ***information hiding***, and works at different levels in an application. The process of encapsulation requires the developer to *separate the parts of a programme that may vary from the parts that stay the same*, i.e. separate the volatile and stable into different classes. If part of a class is stable, it can be abstracted and can therefore be reused. If the SRP is applied during this process, the resultant stable abstraction should only have one reason to change. The reason part of a programme may be volatile is because it has many reasons to change. By separating volatile state and behaviour into a class, each axis of change can be represented by a separate new class and can easily interact will stable components provided a useful abstraction is also created, i.e. the volatile behaviour should be represented by an abstraction that can be dynamically assigned at run-time.

In addition to information hiding, encapsulation also ***protects instance variables*** and data structures from being used incorrectly. If instance variables are private, which they should always be, they can only be accessed indirectly by an external class through non-private methods. These methods provide a mechanism for controlling access to instance variables and even methods and can be used to ensure that they are being used correctly. If an external class has no direct or indirect way of accessing the internal data structures of a class, it cannot have any dependency on that data structure. If there is no external dependency on a data structure, that structure can be changed without adversely impacting other classes. Viewed in this way, encapsulation is used to ***shield*** the users of a class from the effects of change.

Encapsulation works at many different levels in an application:

- **Visibility Modifiers and Data Hiding:** All instance variables should *always be private*. Access to instance variables should be controlled using accessor methods. There is a strong argument to be made for making all methods private unless there is a good reason to do otherwise. Visibility of behaviours should be granted in the order of private, protected and public. Once visibility has been extended, it cannot be revoked without potentially breaking other parts of an application. Once published, the visibility of class members is irrevocable. Protecting instance variables and data structures from direct access is the most basic form of encapsulation.

- **Mutable Objects and Defensive Cloning:** depending on the nature of a programme, it is often desirable to ensure that instance variable are also protected from indirect access (through accessor methods). This only applies to mutable object types and can be accomplished using either cloning or by initialising and returning new objects with similar state to the object being requested.

- **Inner Classes:** if a strong degree of cohesion exists between two separate entities, they can be combined into one class and still uphold the SRP. Classes can be nested, i.e. a class within a class, into inner classes. An inner class can see all of the private class members of its containing class and is within the scope of its container (obviously). Consequently, if declared as private, there is no possibility that an external class can create a dependency on an inner class and the inner class is completely encapsulated. When used incorrectly however, inner classes can create problems. For example, a concrete class can contain an inner interface (this approach is used in the Android API), resulting in a high degree of unnecessary coupling between classes and inducing rigidity into a design.

- **Separate Classes:** a common approach when applying encapsulation is to move the volatile parts of a class into a separate class using the SRP and then extract an interface or similar abstraction that declares the required behaviour. The original class can then be composed with the abstract type, allowing different implementations of the volatile component to be specified at either compile or run time.

- **Deployment Descriptors:** the ultimate form of encapsulation is to remove a volatile part of a class out of an application completely and place it in some type of an initialisation file that is read when the application is started. When used with abstraction, this technique provides the maximum level of separation between stable and volatile parts of a software application.

In addition to these forms of encapsulation, the concept can also be applied to object creation. Using the ***new*** keyword in an application effectively hard-codes a dependency between an object variable and its value (a class constructor). This begs the question of whether or not it is possible to *encapsulate object creation*. A related question is whether or not it is possible to *encapsulate a method invocation*. In both cases, the answer is yes, but will require the application of specific design patterns.

Exercises

In this practical, we will create a simple programme to implement a Caesar cypher that uses inner classes and, through a set of refactorings, encapsulate the programme at different levels to promote reuse and loose-coupling.

- Using Eclipse create a new sub type of ***Throwable*** called *CypherException* that over-rides all the inherited constructors from its super class.

- Create a class called *CaesarCypher* with the following methods:

    - *public String encrypt(String plainText) throws CypherException*
    - *public byte[] encrypt(byte[] plainText) throws CypherException*
    - *public String decrypt(String cypherText) throws CypherException*
    - *public byte[] decrypt(byte[] cypherText) throws CypherException*

    The class should also override the inherited *finalize()* method.

- Add an instance variable called *key* of type ***int*** to the *CaesarCypher* class. Make sure that the visibility modifier is set to ***public***.

- Implement the *encrypt* and *decrypt* methods by using the value of *key* to change the character at each index in the *plainText* to its current value +/- the key value.

- Create a new class called *CypherRunner* containing a *main()* method and use this to create an instance of the class *CaesarCypher*. As the instance variable *key* is public, you should be able to see the variable from the *main()* method.

- Using the ***refactoring*** menu in Eclipse, select the ***Encapsulate Field*** option after highlighting the variable *key* in the class *CaesarCypher*. The refactoring should make the instance variable private and create getter and setter methods.

- Create an inner interface in the class *CaesarCypher* as follows and ***explain any issues that such an approach may give rise to*:**

```
public interface CypherKey{
   public void setKey(String key) throws CypherException;
   public String getKey();
}
```

- Add another inner class to the class *CaesarCypher* with the following class signature:

  public class CypherKeyImpl implements CypherKey

  Implement the required methods (use the ***Override/Implement Methods*** option on the *Source* menu in Eclipse) and add an instance variable called *keyText* of type ***int***. The *setKey()* method should use the static method *Integer.parseInt()* to extract the integer value from the method parameter and throw an exception if the value is not a number. ***Explain the consequences of using an inner class in this way.***

- Using a console, change to the *bin* directory of the Eclipse project and examine the compiled class files. Use the **JD-GUI** tool to reverse-compile the class files for the inner classes in the *bin* directory. ***Explain how the inner class files are linked to their containing classes.***

- Highlight the interface name *CypherKey* and select the ***Move Type to New File*** option from the ***refactoring*** menu. This will move the inner class into a new type declaration of its own. ***Explain how the benefit of this over using the inner class declaration of an interface.***

- In the class signature of *CaesarCypher*, declare that the class implements the interface *CypherKey*. Change the instance variable *key* in the class *CaesarCypher* from an ***int*** to a *CypherKey*. Using the Source menu, select the ***Generate Constructor using Fields*** option to add a constructor parametrised with an instance of *CypherKey*. Add accessor methods for the instance variable *key* and implement the required methods by ***delegating*** calls to the *key* object . ***Explain the benefits that these actions confer on the class.***

- Using the ***Extract Interface*** option in the ***refactoring*** menu, create a new interface called *Cypherable* that declares the four encrypt and decrypt methods. ***Explain the impact that this action has on the overall design of the application.***

- Create a new class called VernamCypher that implements the *Cypherable* interface. The class should also be composed with a *CypherKey* that is passed in as a constructor argument. The encrypt and decrypt methods should use the **XOR** operator (^) operator to mask each character in the plain text and cypher text.