GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computing & Mathematics

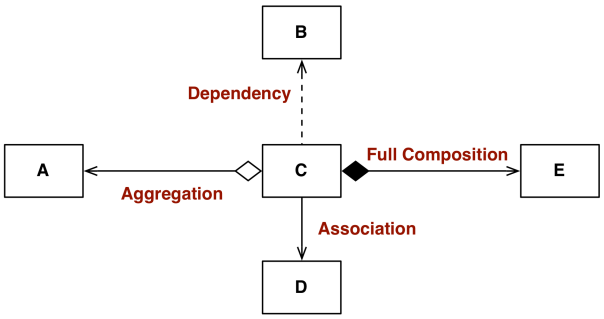
Composition and Object Reuse

One of the principal goals of the object-oriented paradigm is to promote the reuse of software artefacts within an application and between applications. There are two basic mechanisms for achieving reuse in the paradigm – generalisation and composition. The former relates to the reuse of state and behaviour in concrete classes that are derived from a more abstract entity and implicitly supports polymorphism. Composition relates to the construction of new types from a collection of existing classes and represents the primary mechanism for reuse in the object-oriented paradigm.

While it is usually not cited as a "pillar" of the object-oriented paradigm, object reuse through composition is predicated on the proper application of abstraction and encapsulation. Indeed, composition promotes reuse precisely through the encapsulation of reusable behaviour in an abstract entity. Moreover, although inheritance and polymorphism are immediately identifiable as key pillars in the object-oriented paradigm, these concepts are also a consequence of the application of abstraction and encapsulation. The whole point of encapsulation is to separate the stable and volatile parts of an application into abstract and concrete types respectively. The extracted abstract components are highly reusable as their generality confers upon them the flexibility to be used in a variety of different contexts. When not part of a clear hierarchy of responsibility, such abstract entities should be reused through composition. When used correctly, composition enables behaviours and responsibilities to be dynamically assigned to an object at run-time. Implementation inheritance assigns behaviours and responsibilities statically at compile time.

Composition and Scope

There exist a number of different forms of composition that can be used in an application and each form is best categorised on the basis of the scope of the relationship between the container object and the objects that it is composed with. Consider the following UML diagram that illustrates four forms of composition:



The diamonds denote a strong form of composition and require that the containing object maintain an instance variable of that type. For example, the type C should have instance

variables of type A and E, defined at a class level. The full line represents an association between class C and class D, where the classes interact with another class at a method level. The dashed line denotes a dependency, the weakest form of composition. When an object has gone out of scope, the JVM will invoke the inherited *finalize()* method before the garbage collector removes the instance from the heap. The *finalize()* method can thus be used to determine the scope of a composed object and the exact form of composition it manifests.

There are four basic forms of composition:

1. **Dependency (Dashed Line with Arrow):** the scope of composition is restricted to the implementation details of a method. The dependency is not visible or accessible outside the containing class in any way. The composed object is normally fully encapsulated, but may also be shared if singletons are being used.

2. **Association (Fill Line with Arrow):** The scope of a composed object is outside of the class itself, as the composed object is either passed in as a method argument or returned as a type from the method. Either way, the method invocator will have a reference to the composed object at that point in time.

3. **Aggregation (White Diamond with Arrow):** The composed object is declared at a class level, i.e. has the scope of an instance variable. An external call may have direct access to the composed object if it passed the object to the constructor of a container class. Indirect access to the composed object can arise from returning a reference to a mutable instance variable from an accessor method. In an aggregation relationship, the composed object may be referenced by some other object and thus may have a scope greater than its container. Consequently, if the container goes out of scope and is garbage collected, the composed object may outlive its container.

4. **Full Composition (Black Diamond with Arrow):** The composed object is completely encapsulated inside the containing class, with no possibility of direct or indirect access. The scope of the composed object is thus restricted to the class using it. Consequently, when the containing class goes out of scope and is garbage collected, the composed class is guaranteed to be garbage collected with it.

Note that composition and delegation go hand-in-glove. The whole point of composing objects is to reuse their behaviour in another class. That reuse of behaviour relates to the invocation of the methods exposed by the composed class. Consequently, the methods of the containing class should delegate tasks to composed objects where possible. It is important to realise that, using composition, a class can be capable of doing many different things without violating the Single Responsibility Principle.
Finalise

Exercises

In this practical, we will employ composition to reuse the cryptographic capability already provided in the Java SDK and explore how composition can be combined with abstraction and encapsulation to create cohesive and loosely coupled designs.

- Using Eclipse create a new class called ***RSACypher***, composed with the following instance variables imported from the *java.security* an *javax.crypto* APIs:
    - *Cipher* **cypher**
    - *KeyPair* **keyRing**

Override the inherited method ***finalize()*** to output the name of the class and its Object ID.

- In the constructor of ***RSACypher***, implement the functionality to initialise both instance variables as follows:

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(2048);
keyRing = keyGen.generateKeyPair();
cypher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
```

Explain the scope of the three variables defined and the form of composition for each.

- Declare the following two methods for encryption and decryption:

```
public byte[] encrypt(byte[] plainText) throws Throwable{
public byte[] decrypt(byte[] cypherText) throws Throwable{
```

Implement each method by calling the *init()* method of **cypher** with the appropriate parameters. Use the key returned by *keyRing.getPublic()* and *keyRing.getPublic()* for encryption and decryption respectively.

- Create a ***TestRunner*** class with a main method and exercise the functionality of the class to ensure that it works correctly.

- Consider the following code required to encrypt using the DES and AES standards. Note that DES and AES are symmetric cryptographic methods:

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES");
keyGen.init(128);
key key = keyGen.generateKey();
Cipher cypher = Cipher.getInstance("DES/ECB/PKCS5Padding")
cypher.init(Cipher.DECRYPT_MODE, key);
bytep[] result =  cypher.doFinal(cypherText);

KeyGenerator keyGen = KeyGenerator.getInstance("AES");
keyGen.init(128);
key key = keyGen.generateKey();
Cipher cypher  = Cipher.getInstance("AES/ECB/PKCS5Padding");
cypher.init(Cipher.DECRYPT_MODE, key);
bytep[] result =  cypher.doFinal(cypherText);
```

Identify and implement the alterations to the class ***RSACypher*** required to ensure compatibility with symmetric keys.

- Using the ***refactoring*** menu, ***exact a superclass*** called ***AbstractCypher*** from the class ***RSACypher*** that contains a corpus of reusable code. Use abstract methods where appropriate. ***Explain the impact that this action has on the overall design of the application.***

Create two new classes called ***DESCypher*** and ***AESCypher*** that directly inherit from ***AbstractCypher*** and implement the remaining code required to complete the functionality of each class. In each class, override the inherited method ***finalize()*** to output the name of the class and its Object ID.

- Using the ***refactoring*** menu, ***extract an interface*** called ***Cypherable*** from the class ***AbstractCypher*** and alter the type used in ***TestRunner*** to the new interface. ***Explain the impact that this action has on the overall design of the application.***

- Create an *enum* called ***CypherType*** with the options of AES, DES and RSA.

- Create a new class called ***CypherFactory*** defined as follows and alter ***TestRunner*** to return the type returned by the factory method:

```
public class CypherFactory {
  private static CypherFactory f = new Throwable();

  private CypherFactory(){}

  public static CypherFactory getInstance(){
    return f;
  }

  public Cypherable getCypher(CypherType type) throws Throwable {
    if (type == CypherType.DES) {
      return new DESCypher();
    }else if (type == CypherType.RSA) {
      return new RSACypher();
    }else {
      return new AESCypher();
    }
  }
}
```

Explain the impact that this action has on the overall design of the application.