BCS THE CHARTERED INSTITUTE FOR IT

BCS HIGHER EDUCATION QUALIFICATIONS
BCS Level 5 Diploma in IT

OBJECT ORIENTED PROGRAMMING

Wednesady 23rd March 2016 – Afternoon
Answer **any** FOUR questions out of SIX. All questions carry equal marks
Time: TWO hours

Answer any Section A questions you attempt in Answer Book A
Answer any Section B questions you attempt in Answer Book B

The marks given in brackets are **indicative** of the weight given to each part of the question.

Calculators are **NOT** allowed in this examination.

Section A

Answer Section A questions in Answer Book A

General comments on candidates' performance

Future candidates are advised to note the following observations:

- ***Answer all parts of the questions attempted.*** *Some candidates omitted parts of the question. Even if unable to provide a full answer, a partial answer might result in the extra marks that could make a difference between a failure and pass.*
- ***Avoid repetition.*** *Some candidates attempted to gain extra marks by repeating points already made. Future candidates are advised that no marks will be gained for such efforts.*
- ***Answer the questions set.*** *Some candidates attempted to gain marks by providing answers that were related only vaguely, if at all, to the questions set. Future candidates are advised that no marks are given for such answers.*
- ***Use care when writing code.*** *Care and precision is necessary when answering questions using code. Often the examiners had difficulty identifying which variables or commands were being used; inevitably, the candidates lost marks as a result.*

Note that the answer pointers contained in this report are examples only. Full marks were given for valid alternative answers.

Section A

A1.

Describe the following design patterns:

i) Adapter;
ii) Decorator;
iii) Singleton.

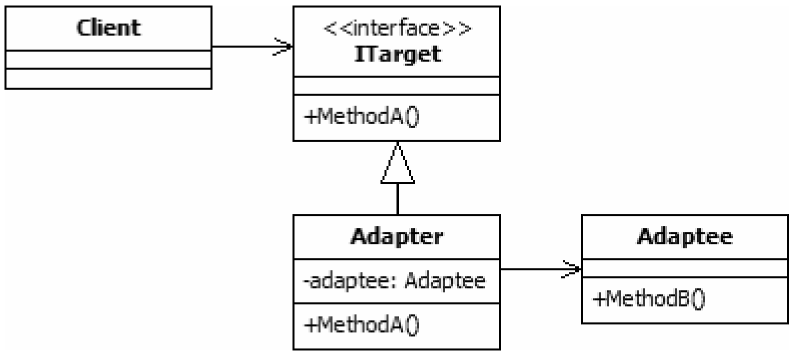
For each pattern, state the motivation for the pattern, give a UML class diagram for the pattern and an explanation of the classes which participate in the pattern.

(25 marks)

Answer Pointers

From: http://www.blackwasp.co.uk/gofpatterns.aspx

The adapter pattern is a design pattern that is used to allow two incompatible types to communicate. Where one class relies upon a specific interface that is not implemented by another class, the adapter acts as a translator between the two types.



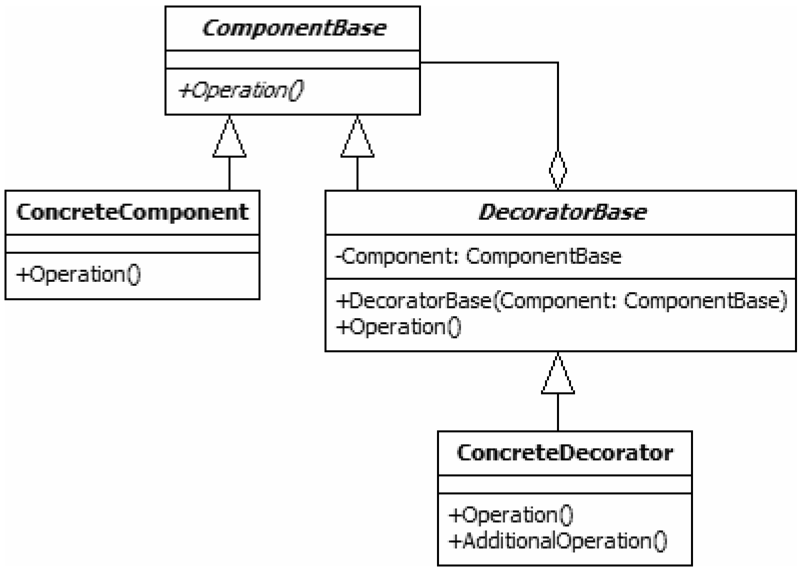
- **Client**. The client class is that which requires the use of an incompatible type. It expects to interact with a type that implements the ITarget interface. However, the class that we wish it to use is the incompatible *Adaptee*.

- **ITarget**. This is the expected interface for the client class. Although shown in the diagram as an interface, it may be a class that the adapter inherits. If a class is used, the adapter must override its members.

- **Adaptee**. This class contains the functionality that is required by the client. However, its interface is not compatible with that which is expected.

- **Adapter**. This class provides the link between the incompatible Client and Adaptee classes. The adapter implements the ITarget interface and contains a private instance of the Adaptee class. When the client executes MethodA on the ITarget interface, MethodA in the adapter translates this request to a call to MethodB on the internal Adaptee instance.

The decorator pattern is a design pattern that extends the functionality of individual objects by wrapping them with one or more decorator classes. These decorators can modify existing members and add new methods and properties at run-time.



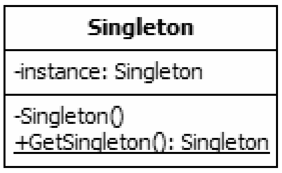
- **ComponentBase**. This abstract class is the base class for both the concrete components and all decorator classes. The base class defines any standard members that will be implemented by these classes. If you do not wish to create any actual functionality in this class you may decide to create it as an interface instead.

- **ConcreteComponent**. The ConcreteComponent class inherits form the ComponentBase class. There may be multiple concrete component classes, each defining a type of object that may be wrapped by the decorators.

- **DecoratorBase**. This abstract class is the base class for all decorators for components. The class inherits its public interface from ComponentBase so that decorators can be used in place of concrete objects. It adds a constructor that accepts a ComponentBase object as its parameter. The passed object is the component that will be wrapped. As the wrapped object must inherit from ComponentBase, it may be a concrete component or another decorator. This allows for multiple decorators to be applied to a single object. When calls to the DecoratorBase's members are made, these are passed unmodified to the matching member of the wrapped object.

- **ConcreteDecorator**. The ConcreteDecorator class provides a decorator for components. In the diagram, an additional method has been included in the decorator. The Operation member can be implemented in two manners. Firstly, the operation may be unchanged. In this case, the implementation simply calls the base method. Secondly, the underlying operation may be modified or replaced completely. In this case, new functionality will be added, which may or may not call the base method.

The singleton pattern is a design pattern that is used to ensure that a class can only have one concurrent instance. Whenever additional objects of a singleton class are required, the previously created, single instance is provided.



The UML class diagram describes an implementation of the singleton pattern. In this diagram the only public interface element is the static "GetSingleton" method. This method returns the single instance held in the private "instance" variable. Usually an instance of the class is created only when first requested.

The constructor for the class is marked as private. This prevents any external classes from creating new instances. The class is also sealed to prevent inheritance, which could lead to subclassing that breaks the singleton rules.

Examiner's Comments

This question examines the Design aspect of the syllabus

The answer pointer sets out a very complete answer to the question whereas candidates received full credit for answers where the UML was correct but the explanation was less complete. The syllabus explicitly mentions design patterns and further lists five specific design patterns which candidates should study. The list includes Adapter, Decorator and Singleton. The question could therefore be answered by recalling book work. Despite this, the question was poorly answered which may indicate that this area of the syllabus has not been studied in sufficient depth. A large number of candidates who answered this question only attempted to describe the Singleton pattern and therefore were only eligible for partial credit.

A2.

a) Give the object oriented terminology for each of the following object oriented features and supply an example of code that illustrates the feature:

   i) A blueprint for an object which defines all the data items contained in the object and the operations that are permitted for the data;
   ii) A representation of something within the domain that the program models which contains values of data and which implements operations on that data;
   iii) An operation which will manipulate the data contained in an object;
   iv) A variable which holds data that describes an individual object;
   v) A variable which holds data that is relevant to all the objects created from the same template.

(5 x 3 marks)

Answer Pointers

i) This is known as a class

```
class MyClass {
      private int a;
      private static int b;
      public void increment() {
            a = a + 1;
      }
}
```

ii) This is an object

```
MyClass anObject = new MyClass();
```

iii) This is known as a method

```
public void increment() {
      a = a + 1;
}
```

iv) This is known as an instance variable. It can only be referenced within the context of an object

```
class MyClass {
      private int a;
      private static int b;
      public void increment() {
            a = a + 1;
      }
}
```

a is an instance variable

v) This is known as a class variable

```
class MyClass {
      private int a;
      private static int b;
      public void increment() {
            a = a + 1;
      }
}
```

b is a class variable

b) Using an object oriented language with which you are familiar to give an example of delegation.

(10 marks)

Answer Pointers

```
public interface ISoundBehaviour {

    public void makeSound();
}

public class MeowSound implements ISoundBehaviour {

    public void makeSound() {
        System.out.println("Meow");
    }
}

public class RoarSound implements ISoundBehaviour {

    public void makeSound() {
        System.out.println("Roar!");
    }
}

public class Cat {

    private ISoundBehaviour sound = new MeowSound();

    public void makeSound() {
        sound.makeSound();
    }

    public void setSoundBehaviour(ISoundBehaviour newsound) {
        sound = newsound;
    }
}

public class Main {

    public static void main(String[] args) {
        Cat c = new Cat();
        // Delegation
        c.makeSound();          // Output: Meow
        // change the sound it makes
        ISoundBehaviour newsound = new RoarSound();
        c.setSoundBehaviour(newsound);
        // Delegation
        c.makeSound();          // Output: Roar!
    }
}
```

Examiner's Comments

This question examines the Concepts aspect of the syllabus

In answers to part a) most candidates recognised a class whilst a smaller number recognised the description of an object. The majority of candidates were able to provide acceptable code demonstrating a method, an instance variable and a class variable.

Many candidates attempting this question chose not to answer part b). The answer pointer indicates a very complete answer and full credit was available for simpler solutions. Quite a few candidates were aware that this question had appeared in previous papers and were able to reproduce the answer pointer presented in the corresponding examiners' report.

A3

a) Explain what is meant by the term abstract data type and how abstract data types are implemented using an object oriented programming language.

(5 marks)

Answer Pointers

An abstract data type (ADT) is a mathematical model for data types where a data type is defined by its behaviour (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behaviour of these operations. An ADT is typically implemented as a class, and each instance of the ADT is usually an object of that class.

b) Explain why it is possible that the same abstract data type can have a variety of implementations.

(5 marks)

Answer Pointers

Data structures can be implemented in a variety of ways, some are simple to implement, some are more efficient. For example a queue can easily be implemented using an array structure. If this is done then removal from the queue involves a shuffling along of array members so that the element previously second in the queue becomes the first. If instead of using an array a linked list implementation is used then element removal can be more efficient. Whatever the underlying implementation, the interface remains the same.

c) What use might you make of an abstract class in implementing an abstract data type?

(5 marks)

Answer Pointers

Given the answer to part b) we see that whilst different implementations of abstract data types may exist each implementation must implement exactly the same methods. An abstract class can be used to define these methods and then specific implementations can inherit from this class. The compiler can then ensure that every implementation supports the essential methods.

d) Explain the way in which the concept of type can be used in a programming language.

(5 marks)

Answer Pointers

Programming languages which support the concept of type enable the programmer to define the type of every variable. When this has been done the compiler is able to warn the programmer about operations which may yield incorrect results. For example if x is a floating point variable and y is an integer then the programmer can be warned that y=x may result in a truncated value.

e) What is the relationship between the concept of type in a typed programming language and the concept of class in an object oriented programming language?

(5 marks)

Answer Pointers

Declaring that an object belongs to a class is analogous to typing a variable. Typing effectively says what operations are valid for a variable. If an object is identified with a class then the class defines the operations which are valid for it. However, in OOP because of inheritance an object may also inherit valid operations from a superclass. Therefore an object may conform with more than one class but have only a single type.

Examiner's Comments

This question examines the Foundations aspect of the syllabus

This material assessed in the question has appeared in previous examination papers. Despite this, and the fact that much of the material is book work, many of the answers given were very poor. Candidates did not understand what is meant by the term Abstract Data Type (ADT) even though it appears explicitly in the syllabus. Some confused it with abstract class (event though part c) hints that these two concepts are distinct). Others confused it with abstraction. Very few answers indicated that a given ADT may have a number of differing concrete implementations. Parts d) and e) of the question have appeared in numerous past paper, however, there were few good answers offered.

Section B

B4. Consider the following class definition:

```
public class date
{
  private int day;          // from 1 to 31
  private int month;        // from 1 to 12
  private int year;         // from 2000 upwards
  public void advance;      // move to next day
};
```

a) Implement a constructor that initialises new objects of `date` class to be set to the 1st of January 2000.

(5 marks)

Answer Pointers

```
public date()
{
  day = 1;
  month = 1;
  year = 2000;
}
```

or a variant thereof (e.g., if implemented outside the class with a scope resolution operator).

b) Implement setters for `day`, `month` and `year`.

(5 marks)

Answer Pointers

```
public bool setDay(int newDay)
{
  if(newDay>0 && newDay <32)
  {
    day = newDay;
    return true;
  }
  return false;
}

public bool setMonth(int newMonth)
{
  if(newMonth>0 && newMonth <13)
  {
    month = newMonth;
    return true;
  }
  return false;
}

public bool setYear(int newYear)
{
  if(newYear>1999)
  {
    year = newYear;
    return true;
  }
  return false;
}
```

or a variant thereof (e.g., if implemented outside the class with a scope resolution operator). Partial marks were given for implementations that did not ensure that the object retained a valid state (i.e., did not bounds-check the argument before storing it in the data members).

c) Implement the `advance` method, which moves to the next day, ensuring that all data members are updated appropriately.

(15 marks)

Answer Pointers

```
public void advance()
{
  const int daysInMonth[12]={31,28,31,30,31,30,31,31,30, 31,30,31};

  day++;
  if(day>daysInMonth(month))
  {
    day = 1;
    month++;
    if(month>12)
    {
      month = 1;
      year++;
    }
  }
}
```

variants that did not discriminate between month lengths were awarded partial marks.

Examiner's Comments

This question examines Syllabus 8D: Practice

The was the least well answered question in Section B of the exam. For a significant proportion of candidates, it appeared that they had very little practical programming experience with an object oriented language, and were unable to provide a satisfactory answer to the easier questions (a) and (b). Part (c) was poorly answered by nearly all candidates, with most unable to devise an algorithm that could take into consideration the different lengths of months. Several candidates that did propose a working solution presented code that was far longer and more complex than necessary.

B5.

a) Describe the meaning of <<extend>> and <<include>> in a UML use-case diagrams

(5 marks)

Answer Pointers

<<extend>> implies a generalization relationship in which the extending use-case continues (i.e. adds more functionality to) the behaviour of the base use case. The extending use case is optional, and may be invoked depending upon a condition.

<<include>> implies a generalization relationship that entails the inclusion of a behaviour defined in a another use-case. The included use case is mandatorily invoked, and the base use case is incomplete without it.

b) How do we represent the visibility of class members as private, public and protected in a UML class diagram?

(5 marks)

Answer Pointers

private:	-
public:	+
protected:	#

in appropriate sections of a class, represented as a rectangle of a diagram, i.e., preceding the names of data and methods.

c) How are aggregation and composition relationships represented in a UML class diagram?

(5 marks)

Answer Pointers

aggregation: empty diamond

composition: filled diamond

d) How are specialisation and generalisation relationships defined, and how they are represented in a UML class diagram?

(10 marks)

Answer Pointers

specialisation:

The relationship stresses that a more elaborate form of a base class (also known as a super class) has been produced, that includes the heritable components of the base class, and perhaps includes additional data members and methods. For instance, a textbook may be a specialization of a book.

generalisation:

The specialization relationship can be viewed in reverse. A base class is a more generalized version of a derived class, which contains fewer data members or methods, and is less specific in its domain of representation. For instance, a book may be a generalization of a textbook. A group of classes may have been produced in a design that share common properties. A super-class could be used to retain these common properties, which can then be inherited to prevent duplication.

representation in UML:

Both are shown on a UML class diagram using an empty triangle. It is a matter of perspective which of the two relationships is at work: from the perspective of sub-classes, the super-class is a generalization. From the perspective of the super-class, the sub-classes are specializations.

Examiner’s Comments

This question examines the Syllabus 8C: Design

Generally, this question was reasonably well answered, although candidates were frequently unclear about the distinction between generalization and specialization, and between <<include>> and <<extend>>, suggesting that further study and practice is required in UML.

B6.

a) Four basic features of object-oriented programming languages are said to be abstraction, polymorphism, encapsulation and inheritance. Define each of these terms.

(20 marks)

Answer Pointers

Abstraction:

The concept of exposing only essential data and other details to the users of a class, whilst hiding unnecessary information that could lead to either confusion or increase the probability of making mistakes.

Polymorphism:

Allows entities to be used in multiple forms whilst retaining similar syntax (for example, method overloading and overriding, template classes, operator overloading, default arguments, and so on)

Encapsulation:

Prevents our data from unwanted accesses by binding code (methods) and data (attributes) together into a single unit called an object.

Inheritance:

Promotes the reusability of code; a sub-class acquires all the heritable features of its superclass parents, and may use them and extend upon them.

b) How does method overloading differ from method overriding?

(5 marks)

Answer Pointers

Method Overloading:
Having several versions of the same method name, but with different signatures. The appropriate version is executed, depending upon the arguments passed when the method is invoked.

Method Overriding:

Creating of two or more methods with the same name and signature in different classes in the class hierarchy (i.e., super-class and sub-class), such that (for example) a sub-class version can overshadow an inherited (super-class) version.

Examiner's Comments

This question examines the Syllabus 8D: Practice

Many candidates conflated the concepts of abstraction and encapsulation in part (a), and frequently confused overriding and overloading in part (b). However, most of those attempting this question were able to provide a reasonable description of inheritance and polymorphism. Since this question requires an understanding of basic terminology, and appears in the syllabus, it is surprising that fuller answers were not provided.