BCS THE CHARTERED INSTITUTE FOR IT
BCS HIGHER EDUCATION QUALIFICATIONS
BCS Level 5 Diploma in IT

September 2013

EXAMINERS' REPORT

Object Oriented Programming

Section A

Question 1
Answer Pointers

a)
A method is commonly identified by its unique **method signature**. This usually includes the method name, the number and type of its parameters, and its return type.

b)
Method overloading is a feature found in various programming languages that allows the creation of several functions with the same name which differ from each other in terms of the type of the input and the type of the output of the function. In overloading the name of the function is the same but some other part of the signature is different.

```
class Myclass{

        void myMethod(int x){
        .
        .
        }

        void myMethod(float x){
        .
        .
        }
}
```

The method myMethod is overloaded

c)
Method overriding allows a subclass to provide its own implementation of a method already provided by one of its superclasses. A subclass can give its own definition of methods which also happen to have the same signature as the method in its superclass.

```
class MySuperclass{

        void myMethod(int x){
        .
        .
        }
}
```

```
class MySubclass extends MySuperclass{

        void myMethod(int x){
        .
        .
        }
}
```

The method myMethod is overridden.

d)
Operator overloading is a specific case of polymorphism in which some or all of operators like +, = or == are treated as polymorphic functions and as such have different behaviours depending on the types of their arguments.

e)
Polymorphism is the idea of allowing the same code to be used with different types, resulting in more general and abstract implementations

Overloading allows multiple functions taking different types to be defined with the same name; the compiler or interpreter automatically calls the right one. This way, functions appending lists of integers, lists of strings, lists of real numbers, and so on could be written, and all be called *append*—and the right *append* function would be called based on the type of lists being appended. This differs from parametric polymorphism, in which the function would need to be written *generically*, to work with any kind of list. Using overloading, it is possible to have a function perform two completely different things based on the type of input passed to it;

Overriding is used to implement parametric polymorphism. Using **parametric polymorphism**, a function or datatype can be written generically so that it can deal equally well with objects of various types. Object oriented languages employ the idea of *subtypes* to restrict the range of types that can be used in a particular case of parametric polymorphism. In these languages, **subtyping polymorphism** allows a function to be written to take an object of a certain type *T*, but also work correctly if passed an object that belongs to a type *S* that is a subtype of *T* (according to the Liskov substitution principle).

Examiner's Comments

This question examines Concepts

This question was answered by nearly 80% of the candidates. The majority of the answers were satisfactory, however, the average score was only just above the pass mark for the question. In general candidates were able to give good answers to parts a, b and c. Operator overloading is not a feature of many object oriented programming languages, however, it is explicitly mentioned in the syllabus. Only a few candidates could explain this concept. In part e, candidates were generally able to define polymorphism but were unable to state how overloading and overriding could be used to implement polymorphic behaviour.

Question 2

Answer Pointers

a)

i) the identification of common features and operations;

ii) the process of combining elements to form an new entity;

iii) a mechanism for hiding the details of the implementation of an object from the code that uses it.

b)
A process of abstraction should indicate that all collection classes do essentially the same task. In particular every collection class will be able to add to the collection and remove items from the collection. Via polymorphism the nature of these operations may however be different so adding to a Set will be implemented in a different way from adding to a List (duplicates are allowed in lists but not in sets). Data hiding will conceal these differences so that an implementer needn't worry about them but just be aware that adding may in some cases fail. Encapsulation permits the packaging of different parts of an entity together as an object. Collection classes which will operate on objects can be used in any context so that it is not necessary to implement a specialist class for each different type of object in the collection.

c)
By identifying the most general cases and hiding the details of specific implementations from the code using it, it is possible to develop set of classes which operate on all objects rather than just objects belonging to specific classes. Such classes are easily reusable.

Examiner's Comments

This question examines Concepts

This question was answered by roughly half the candidates for the paper. Only a few answers were acceptable. In part a, many candidates confused abstraction with abstract classes and were unable to distinguish between encapsulation and data hiding. Only a few candidates could recognize that the data structures in part b were similar enough to have a common implementation. Many candidates did not attempt part c at all.

Question 3

Answer pointers

a)

Answers could have included (but were not limited to) the following features:

Object – the combination of data and the operations that apply to that data.

Class – the pattern for all objects which have identical data and behaviour.

Inheritance – the facility to derive new classes from existing classes.

Methods – the way in which behaviour is defined.

Messages – the mechanism for invoking behaviour.

b)
There are three main ways in which object-oriented programming might improve programmer productivity:

i) By providing an environment which assists programmers to improve the quality of their code;
ii) Making it easier for programmers to reuse their own code;
iii) Providing simple mechanisms to make use of existing code libraries.

Object-oriented programming embodies practices which have been known for some time to lead to well constructed programs. It associates procedures with the data those procedures use. It can be used to form a clear separation between underlying data structures and functionality. The concept of object is a good abstraction mechanism which helps a designer separate out a small part of a problem and concentrate on that simpler part consequently increasing the probability of that part of the design being correct. Object-oriented programming languages encourage and support object thinking.

In general the fewer lines of code a programmer has to write the more productive they will be. This can be facilitated by providing powerful features in a language (such as a matrix multiplication operator). The disadvantage of such features is that almost certainly in a bespoke environment they will not do exactly what a programmer needs. Consequently programmers will write their own code to do this. A classic example is a date checking routine. Often such routines are textually copied from one program to the next. This creates a maintenance nightmare. Object-oriented programming facilitates and encourages the use of re-usable classes which allow programmers to reuse the same code over and over again without physically copying it.

Just as programmers may reuse their own classes, they may also easily reuse classes provided by third parties. This is particularly useful where programmers have to produce complex interfaces to their programs. The inheritance mechanism allows programmers to enhance third party classes to meet their individual requirements without the need to alter the original code and therefore tailored software can be developed. The majority of code in any application will often be incorporated in this way and therefore productivity is greatly enhanced.

The disadvantages of object-oriented programming include the learning curve necessary to become proficient in it and the fact that code produced by an object-oriented language compiler is unlikely to be as efficient as code produced by the best machine code programmers.

Examiner's Comments

This question examines Foundations

This question was answered by just over 70% of the candidates and the majority of answers were satisfactory. Both parts of the question have been posed in various guises in previous papers and therefore candidates who had studied past papers were well prepared to produce good answers. Some candidates lost marks by repeating their answer for part a in part b. Part a was looking for technical details whilst part b, where the report was targeted at a manager was looking for the practical gains from those technical details.

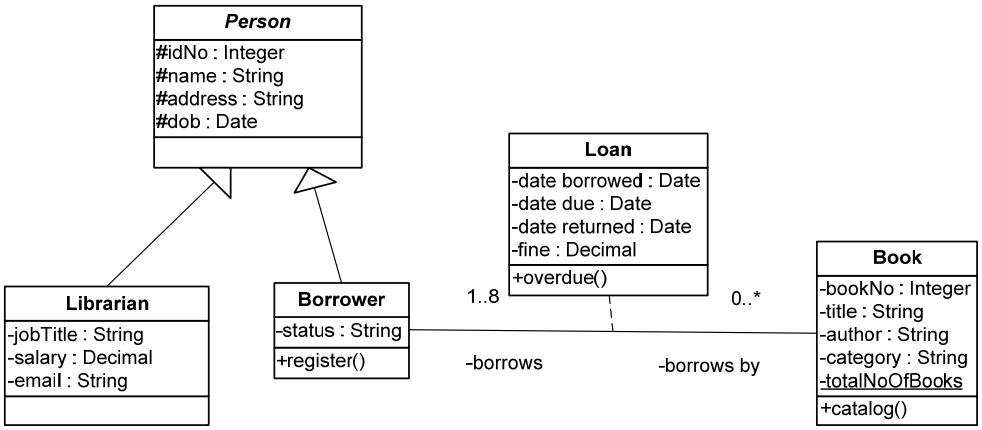
Section B

Question 4

Answer pointers

a)

Sample class diagram:



b)

The candidates can pick any of their classes to show some correct and incorrect instances. A valid example would be an association between a Borrower and Loan, whereas an invalid one would be between Staff and Loan.

c)

In a class diagram:
"–" symbol specifies that a data member or method is private,
"+" that it is public
"#" that it is protected.

A class is made up of compartments:
Top – class name
Middle – attributes
Bottom – methods

Examiner's Comments

This question examines parts 8c of the Syllabus: Design

This question was attempted by 78% of the candidates, with a 53% pass rate. To achieve a good mark, the classes needed to be associated correctly, candidates who

linked each class to every other class and with few, or no constraints shown achieved poor marks.

For full marks, the candidate also needed to show the attributes and methods in the correct class. Some candidates produced an Entity-Relationship (ER) diagram instead of a Class diagram.

A small number of candidates did not fully understand that the Loan class should be linked to both a Borrower and a Book, however, provided that it was linked with the appropriate constraints to at least one of these classes, then credit was awarded.

Most candidates could provide example instances for part b, marks were lost where either only valid or invalid examples were given.

In part c, marks were lost where the candidate mixed up the symbols, or failed to give an appropriate example.

Question B5

Answer pointers

a)
An open-ended question. The Candidate may look at different approaches, for example:

- Fault based testing
- Scenario based testing

The candidate must discuss which UML diagrams can be used for testing, e.g., Use Case scenarios, or State machine diagrams, which show the different states an object can be in and the transitions between those states.

b)

The following five are listed design patterns from the syllabus.

Adapter (Structural)

The Adapter is intended to provide a way for a client to use an object whose interface is different from the one expected by the client, without having to modify either. This pattern is suitable for solving issues such as replacing one class with another when the interfaces do not match and creating a class that can interact with other classes without knowing their interfaces at design time.

Decorator (Structural)

Ordinarily, an object inherits behaviour from its subclasses. The decorator pattern allows the behaviour of an object to be extended and dynamically compose an object's behaviour. Decorator allows this without the need to create new subclasses.

Iterator (Behavioural)

The Iterator pattern provides a way to access the elements of an aggregate object sequentially without having to know the underlying representation. Iterator also provides a way to define special Iterator classes that perform unique processing and return only specific elements of the data collection. The Iterator is useful because it provides a common interface so programmer does not need to know anything about the underlying data structure.

Observer (Behavioural)

The Observer pattern is useful when data is to be presented in several different forms at once. The Observer is intended to provide a means to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The object containing the data is

separated from the objects that display the data and the display objects observe changes in that data. Often used in MVC.

Singleton (Creational)

The singleton pattern applies to the many situations in which there needs to be a *single instance* of a class, a single object. Print spoolers and window managers are examples of Singletons. The Singleton is intended to provide a way to ensure that a class provides one instance of itself, and to provide a global point of access.

For each category, the candidate should give a short description of one of the diagrams, include a simple example and say when they should be used.

Other examples of valid design patterns were also acceptable.

Examiner's Comments

This question examines parts 8c & 8d of the syllabus Design and Practice

This question was attempted by 30% of the candidates, with a 37% pass mark. For part a some candidates just said what the diagrams were, without relating them to testing. In order to gain a high mark, the role of the diagram in the testing process had to be discussed.

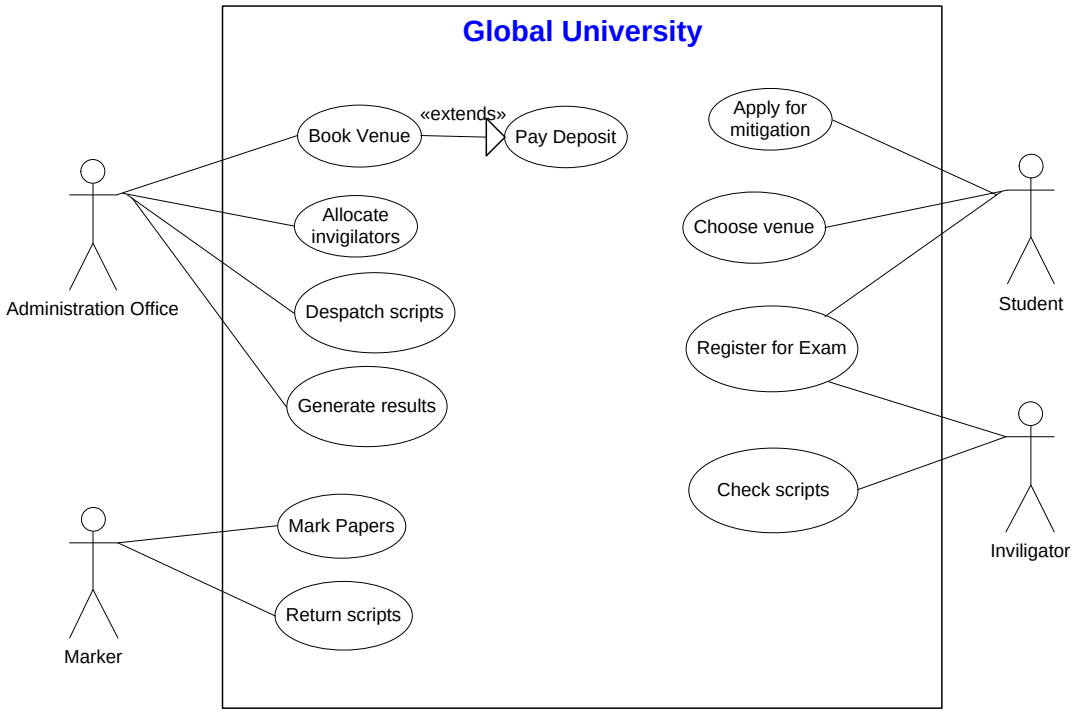
Most candidates could produce a good answer for part b. The candidates that failed were those who attempted to describe all the design patterns, not just one from each group. In such cases the answers were too brief to gain much credit. It must be noted that only one design pattern per group should be described, no additional credit is given for those who discuss several.

Some candidates only discussed two of the groups; in particular the structural design pattern was not attempted.

Question B6

a)

Sample Use Case diagram:



Answer pointers

b)

The candidate should discuss where Use Cases should be used in developing a system.

For each Use Case on the Use Case Diagram the user should develop a set of scenarios. Scenarios are natural language descriptions of an instantiation of the Use Case.

These can be used in these areas:

- Initial investigation
- For identifying what functionality is required from the system
- The descriptions give the fuller detail
- Testing purposes

The Use Case descriptions can be used by the testers to develop test plans, checking that the system meets the requirements of the system

The candidate should include a brief example of a scenario to illustrate their points, e.g., book venue.

Examiner's Comments

This question examines parts 8c of the syllabus Design

This was a popular question with 78% of the candidates attempting it and a similar number passed it. Most candidates made a good attempt at Part a, identifying the main use cases. Marks were lost where the use cases were either too brief to identify what they represented adequately, or conversely were too descriptive, specifying the use case in great detail. To gain full marks, the use cases has to be linked to the correct actor, with any use of the <<extends>>, or <<includes>> dependencies used correctly. Some answers linked every use case to every other use case and gained little, or no credit. Weak answers also mixed up the actors with the use cases.

For part b, some candidates just listed a use case description, with no attempt to explain why they were useful in the development of an object-oriented system. Examples were welcome, however, as part of a discussion on their role in the design process. Some answers were too brief to gain full credit, for example, it is not sufficient to just say use cases can be used for testing purposes, a fuller explanation of how this can be achieved is needed for a higher mark.