BCS HIGHER EDUCATION QUALIFICATIONS
BCS level 5 Diploma in IT

April 2010

EXAMINERS' REPORT

OBJECT ORIENTED PROGRAMMING

General Comments

Candidates should take care to ensure that they write their answers in the correct answer book. In addition, they should record the questions they have attempted on the front of the answer book.

There were a number of candidates this year who were unable to construct a coherent answer for some of the questions they attempted. Candidates are reminded that this examination is set at a level that is consistent with that expected in the second year of a UK undergraduate degree and it is unlikely that someone who has not undertaken an academic study of Object Oriented Programming will be successful in passing this paper.

Question A1.

Answer Pointers

a) Both mock and fake objects are used in program testing as substitutes for real (fully functional) objects. To some degree, they both simulate the behaviour of real objects. The difference between fake and mock objects derives from the authenticity of that simulation. Fake objects are simpler, typically returning pre-set responses when their methods are invoked. Mock objects, on the other hand, react more specifically to the parameters supplied when their methods are invoked (e.g., a mock object might be used to simulate a hardware device). Since mock objects react differently in response to different invocation scenarios, they mimic the behaviour of the object they are substituting with greater realism.

(5 marks)

b) Ad-hoc polymorphism refers to the scenario in which polymorphic code must be explicitly provided by the programmer to account for each different data type that the code may encounter (e.g., function overloading). Conversely, in parametric polymorphism, code can be written that can operate upon any data type, dynamically adapting its behaviour without the requirement for specific, detailed instructions that describe how to deal with each individual data type that may be encountered (e.g., template classes).

(5 marks)

c) This relates to the fact that seemingly minor changes to a base class can have unforeseen effects on derived classes (i.e., may stop them from behaving correctly, and may even stop them from compiling). Such effects may be difficult to predict, since the source code for a base class does not inform the programmer how many derived classes have been created, or what assumptions those derived classes may make with respect to the base class implementation. One solution is to declare base classes as final or constant (prohibiting them from being modified in the class hierarchy subsequently), another is to favour interfaces over concrete base classes, and/or to use private methods and data member to prohibit access to and modification of potentially sensitive code.

(5 marks)

d) A term typically used in reference to base and derived classes, wherein a code fragment referring to a base class object may seamlessly utilise an object of a derived class without needing to be modified. It is often discussed in relation to LSP (the Liskov Substitution Principle), and, more generally, aims to guarantee the interoperability of types in a class hierarchy. An example might be a book object that can substitute for a publication object, since methods originating from publication will have been inherited into book, and will thus still be available where a book object substitutes for a publication object. However, some specific inheritance relationships violate this principle, particularly where a derived class redefines some base class behaviours.

(5 marks)

e) Virtual inheritance is used to prevent ambiguities in multiple inheritance. For example, in creating a class D we might inherit from two base classes (B and C) that were themselves both inherited from a common base class (A) creating a “diamond”. In class D, we effectively inherit from class A twice, meaning that any reference to a data member or function originating from A is ambiguous, since they are visible through both B and C. Virtual inheritance ensures that D only contains one occurrence of the class A, resolving this ambiguity. The “diamond” inheritance issue is particularly problematic where functions originating from A are overridden both in derived classes B and C. In this case, it is not clear which of the two versions of that function should take precedence in our new class D.

(5 marks)

Examiners Comments

This question examines Syllabus 8D: Practice

Less than 8% of candidates attempted this question, despite that it tests bookwork and a comprehension of fairly common object oriented terminology. Furthermore, those that did attempt this question scored very poorly. It is likely that, since the concepts included in this question did not appear in earlier examination papers, that candidates were ill-prepared to answer them. Candidates should read widely around the subject matter to prepare

themselves for a broad variety of questions, in addition to studying past examination papers, the syllabus and recommended texts.

Question A2.

Answer Pointers

a)

i. association: solid arrow-headed line

Association indicates that two classes are related to one another, but not that one class is formed from or possesses another. It is common to annotate an association with an indication of one classes' role with respect to the other, and the multiplicity of that relationship. For example, 0 or more readers may subscribe to 0 or more magazines in a publishing company.

ii. aggregation: solid arrow-headed line emanating from an empty diamond

Aggregation represents a "has-a" relationship between classes, where, unlike composition, both classes may also have an independent existence. For example, patient has-a doctor, but both patient and doctor may still sensibly exist without each other.

iii. composition: solid arrow-headed line emanating from a filled diamond

Composition represents an "owns-a" relationship between classes. Unlike aggregation, the subservient class does not have an existence independently of the class that it is owned by. For example, a linked list node may be owned by a linked list, but may not exist without the linked list that it is contained within.

(10 marks)

b) Concrete classes are represented in a UML class diagram as boxes with three sections. In the top section, the name of the class is specified. In the middle section, the attributes (data members) of the class are listed. In the bottom section, the behaviours (member functions) of the class are listed.

Abstract classes also have 3 sections, as identified above, but the top section of the box, containing the class name, is shown in italics. It would be normal for an abstract class to contain one or more abstract function, also presented in italics.

(5 marks)

c) The "–" symbol specifies that a data member or method is private, "+" that it is public, and "#" that it is protected. We can distinguish between attributes and operations in a UML class diagram both by their relative positions (attributes appear in the middle section of the box and behaviours in the bottom section) and possibly by the existence of round brackets () and, potentially, arguments after method names that do not exist for data members.

(5 marks)

d) A class diagram provides a model of the system, detailing the types and names of data members, and the signatures of functions for each class, and indicating how classes are inter-related. An object diagram provides a "snapshot" of a particular scenario, in which objects of the classes represented in a class diagram exist, and are populated with relevant data. Object diagrams can be used to generate or model examples of the system's state at particular times, and thus focus on attributes (data members) rather than behaviours (member functions).

(5 marks)

Examiners Comments

This question examines Syllabus 8C: Design

This was the most popular question in this year's exam, having been attempted by 87% of candidates; it was also the question that was answered most successfully. In part (a), many candidates confused the symbols for aggregation and composition, and a significant number lost marks by neglecting to text to address that question, which asked candidates to "distinguish between" these relationships as well as indicating how they are represented in UML. In part (d), a significant number of candidates were unable to adequately differentiate between class and object diagrams.

Question A3.

Answer Pointers

a) The candidate may choose to describe any three practical example of polymorphism, which may include one or more of the following: function overloading, function overriding, dynamic (late) binding, compile time polymorphism (e.g. templates), and operating overloading. Examples of the first 3 are provided below:

Function Overloading:

Function overloading is the technique of having multiple occurrences of a function with a common name, but with different arguments. A specific version of the function is selected such that it matches the arguments used at invocation. For example:

```
class A
{
  void b(int n)    {cout <<n;}  // version 1
  void b(double n) {cout <<n;}  // version 2
}

int main(void)
{
  A mya;
  a.b(10);  // uses version 1
  a.b(6.5); // uses version 2
}
```

Function Overriding:

Function overriding is the technique of creating a new implementation of a base class function in a derived class. The derived class function must share the same name and argument list as the base class function for overriding to occur, otherwise the derived class function will simply be added to the set of functions available. For example:

```
class A
{
  virtual void b()  { cout << “message 1”;}
}
class B: public A
{
```

```
  void b()          { cout << “message 2”;
}


int main(void)
{
  A mya;
  mya.b();
  B myb;
  myb.b();
}
```

Dynamic (Late) Binding:

Dynamic binding enables us to defer the exact behaviour of a code fragment until run-time, and exploits the substitutability principle. It is implemented using a pointer to a base class type that may also be used to point to classes derived from that base class. When invoking a member function originating from the base class that is overridden in the derived class, the behaviour of that function will correspond to the specific derived class type pointed to at the particular time that the function was invoked.

```
class Event
{
  public:
    virtual void handleEvent() {};
};
class specialEvent1: public Event
{
  public:
    void handleEvent() { cout << “type 1\t”; }
};
class specialEvent2: public Event
{
```

```
  public:
    void handleEvent() { cout << “type 2\t”; }
};


int main(void)
{   specialEvent1 a,b;
    specialEvent2 c,d;
    Event* event[4];
    event[0]=&a;
    event[1]=&b;
    event[2]=&c;
    event[3]=&d;
    for(int i=0 ; i<4 ; i++)
      event[i] -> handleEvent();
}
```

(10 marks)

b) i. abstract vs. concrete:

Defining a method as abstract, which is possible only within an abstract class, enables us to defer implementation to a later stage in the inheritance hierarchy. This is used when we know that a particular method must be made provided, but it is too early to prescribe exactly how it will be accomplished. For example, an abstract class animal may contain an abstract method reproduce, since this is a basic requirement of all organisms classed as animal, before we know whether the specific animal sub-class we create will require a sexual or asexual implementation. Concrete classes, on the other hand, are fully implemented.

ii. accessor vs. mutator:

Accessor methods are used to retrieve data member values from within an object, commonly known as getters, and mutator methods are used to change the value of data member, and are commonly known as setters. Mutator methods therefore accept arguments (reflecting the

data member that they are destined to update), whereas accessor methods usually possess a return type reflecting the data member that they are to convey back to the caller. Both are typically designated as public to facilitate their invocation through an object: accessors to report information pertaining to the object's state, and mutators to modify the object's state.

iii. support vs. service:

Support methods, which are typically set with private or protected visibility, are used to assist other methods (functions) in performing internal tasks, but are not available to call through the object. Service methods, which are set with public visibility, provide services to the user of an object, and may be invoked through the public interface of an object.

iv. constructor vs. destructor:

Constructor methods are called when an object is created, and destructor method are called when an object goes out of scope. Destructors may not exist in some languages with garbage collection (e.g. Java). Constructors undertake initialisation duties such as setting data to default values, preparing files, sockets, or GUI components. Destructors are used to end the existence of an object gracefully, closing open files, sockets, or GUI components, and, in some languages, are used to de-allocate dynamically allocated memory to prevent memory leakage.

v. instance vs. class

Instance methods operate upon instances of a class (i.e., objects), and are typically concerned with the manipulation of instance variables – i.e., those data members that may differ from object to object. Class methods are typically concerned with the manipulation of class variables, and do not require the existence of any instance to operate.

(15 marks)

Examiners Comments

This question examines Syllabus 8D: Practice

82% of candidates attempted this question. Overall, the question was answered relatively well. In part (a), marks were lost by many students who were unable to provide 3 practical examples of polymorphism (instead they provided only 2), or were unable to show illustrative code fragments. In part (b), candidate's knowledge of basic theory was tested by contrasting

common terms in object oriented programming; a significant proportion of candidates were unable to correctly distinguish between instance and class methods, but most students managed to score a large proportion of the available marks.

Question B4.

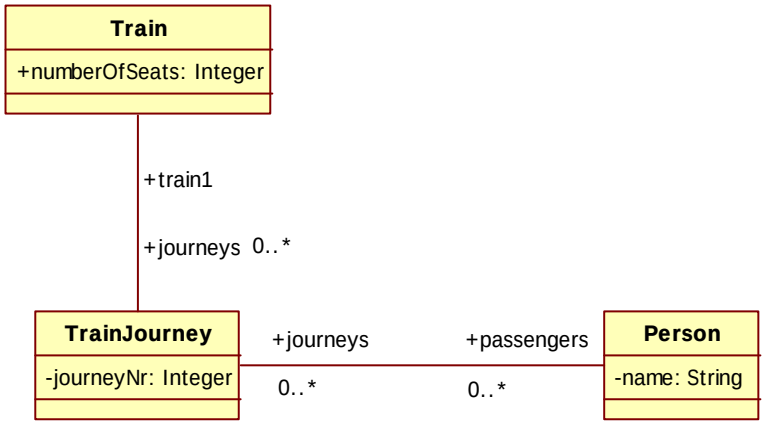
a) Write a report which explains to a software team manager how the UML can contribute to the development of object oriented software.

(10 marks)

Answer Pointers

Specification and Design are important parts of software construction. It is impossible to construct high quality software without first having a specification and developing it into a design. The quality of specifications and designs is greater when they are communicable. The nature of what is to be constructed should be communicated with at least the potential user and also with colleagues in the implementation team. An ability to check the early stages of software development is known to be an important factor in reducing the cost of the software and increasing its reliability. It is important when specifications and designs are shared that the understanding of the stakeholders is unambiguous. For this reason it is important that a design or specification method is at least semi-formal in its approach. The Unified Modelling Language (UML) provides a semi-formal mechanism for specifying and designing object oriented systems. Its set of diagrams which include use case diagrams, class diagrams, activity diagrams and others provide useful techniques whereby aspects of a design or specification can easily communicated to all the stakeholders in a system. Object Constraint Language (OCL) can be used in addition to the diagramming techniques to provide precise specifications of the semantics of the design.

b) Describe the meaning captured by the following class diagram and OCL statement



context Train Journey

inv: passengers->size() <= train.number Of Seats

(15 marks)

Answer Pointers

There are three classes captured in the design Person, Train and TrainJourney.

For each Person we record their name in a private variable of type String.

For each Train we record the number of seats in a public variable of type Integer.

For each TrainJourney we record the journey number in a private variable of type Integer

A Train may be related to none or many TrainJourneys.

A Person may be a passenger on none or many TrainJourneys.

A TrainJourney is associated with one Train.

A TrainJourney is associated with none or many passengers (Persons).

On any TrainJourney the number of passengers will not exceed the number of seats on the Train.

Examiners Comments
This question examines part 8C of the syllabus: Design

This was a popular question and the majority of students who chose to attempt it were able to offer satisfactory answers. In part a), candidates demonstrated that they had a good grasp of the uses of UML and the advantages it can offer to software developers.

Part b) was a reworking of a question which had been set in the previous year. On this previous occasion, candidates had not been familiar with OCL even though it is explicitly referenced in the syllabus. Candidates answering this year showed a greater awareness of the Object Constraint Language which indicates that this topic has now been incorporated into courses that prepare people for the examination.

Question B5.

a) Describe two testing techniques which may be used to ensure that methods produce the expected results.

(10 marks)

Answer pointers

White box testing and black box testing would assist this task. In white box testing the programmer designs test data on the basis of a knowledge of the code forming the method. The aim of the test data is to test every possible route through the method. Consequently, the test data is chosen on the basis of the decision branches in the code. In black box testing the test data is determined by an examination of the specification of the method. No assumption is made about the way in which the code is written. From the specification the programmer derives boundary values and equivalence partitions. The test data consists of values from each of the equivalence partitions and the boundary values. The programmer predicts the expected outcome for each of the values and runs them through the code in order to check that the actual value matches the expected value.

b) Object oriented systems almost invariably involve a number of interacting objects. Describe an approach which can be used to test the interactions

(10 marks)

Answer Pointers

Integration testing is a technique for testing assemblages of classes which will previously been tested separately. In big bang integration testing, the majority of the components of the system are put together. This is a very quick way of building the system but requires careful design of the test data. In bottom up testing the components at the lowest levels of the hierarchy are combined and tested first. The software is then put together by including successively higher level components. In contrast, in top down testing the higher levels are integrated first and tested. Initially lower levels will be stubs but as testing continues the stubs are replaced by actual classes.

c) Discuss the following statement:

“Program testing can be used to show the presence of bugs, but never to show their absence!” (Dijkstra)

(5 marks)

Answer Pointers

The assumption of testing is that the programmer will submit a set of test data which is sufficiently inclusive so as to fully test the program, but this may not be the case. If the test data submitted by the programmer shows a variance between the predicted and the actual output, this is a bug which will be addressed. There are many programs however which are expected to process an infinite number of acceptable inputs. The programmer can only

supply representative samples of these inputs and therefore it is possible that undiscovered problems will remain.

Examiners Comments
This question examines part 8D of the syllabus: Practice.

This question was answered by just over half of the candidates. Despite the fact that it is a very straightforward question which has been asked in a variety of forms in previous papers, on this occasion candidates offered very poor answers. In part a), candidates were often able to name the two techniques cited in the answer pointers but were unable to offer any details about techniques. In order to obtain the full 10 marks, candidates were expected to give an outline description of how one would carry out the two testing methods. Many candidates simply did not offer answers to parts b) and c).

Question B6.

a) Explain the following terms:

i) *Structured programming*;
ii) *Modular programming*;
iii) *Abstract data types*;
iv) *Typed languages*;
v) *Untyped languages.*

(10 marks)

Answer Pointers

Structured programming is a type of programming methodology where a complex problem is decomposed into tasks which can be modelled by simple structures such as for…next, if….then and while…do

Modular programming is a technique for decomposing a programming problem into simpler task which together can be composed to form the solution.

Abstract data types or ADTs are data types described in terms of the operations they support—their interface—rather than how they are implemented

In typed languages, there usually are pre-defined types for individual pieces of data (such as numbers within a certain range, strings of letters, etc.), and programmatically named values (variables) can have only one fixed type, and allow only certain operations: numbers cannot change into names and vice versa..

Untyped languages treat all data locations interchangeably, so inappropriate operations (like adding names, or sorting numbers alphabetically) will not cause errors until run-time.

b) Define the terms *coupling* and *cohesion* and explain how these concepts contribute to the quality of a program. Show how the object oriented concept of *encapsulation* aids a programmer to produce good quality code when measures of coupling and cohesion are used to gauge quality.

(10 marks)

Answer Pointers

Coupling refers to the degree to which each program module relies on each other module. Coupling can be "high" or "low". Low coupling means that one module does not have to be concerned with the internal implementation of another module. Low coupling is a sign of a well structured computer system.

Cohesion refers to the degree to which each part of a module is associated with each other part, in terms of functional relation. Parts of a module are functionally related if each part is essential to the functionality and the interface of a well-defined module (a well-defined module is one that has a single task, or models a single object). Cohesion can be considered "high" or "low". High cohesion means each part of a module is functionally related, and low cohesion means each part of a module is not. High cohesion of a module is considered better design.

Encapsulation supports the notion of gathering together related aspects of a design, grouping them and hiding the implementation. It therefore encourages high cohesion. The combination of information hiding with a well defined set of interface methods also supports low coupling as it prevents a component of the design gaining access to the internals of an implementation

c) Contrast the way in which classes are implemented in typed object oriented programming languages and untyped object oriented programming languages.

(5 marks)

Answer Pointers

In a typed language a class is usually also a type. Variables can be declared to have a type which is identical to a class name. In an untyped language a class is only an object factory which is used to create object which conform to that class. Typed languages perform compile time checks on the messages sent to objects. In untyped languages the object must report at run time if it receives a message it cannot understand.

Examiners Comments
This question examines part 8A of the syllabus 8a: Foundations

Once again this was a standard question which should have been familiar to candidates who have worked through past papers and the associated examiners' reports. It was attempted by almost 80% of the candidates (demonstrating that there was general familiarity with the topic of the question) but less than half of these produced a satisfactory answer.

Part a) sought definitions of terms that appear explicitly in the syllabus. A candidate who had learnt these definitions and was able to reproduce them could have obtained a pass mark for the question. Some candidates had clearly not encountered any of the terms list, others had memorized the definitions but muddled them up. In part b), candidates were once again able

to give definitions for coupling and cohesion but were unable to explain how object oriented constructs encouraged low coupling and high cohesion. Many candidates did not attempt part c).