

**THE BCS PROFESSIONAL EXAMINATION
Diploma**

April 2005

EXAMINERS' REPORT

Object Oriented Programming (Version 2)

General

A truly disappointing outcome to this assessment. Disappointment for the examiners but, more importantly, even greater disappointment for the candidates.

The pass rate (60%) and the overall average are some 5 points down from the last sitting (October 2004). The only consolation for the examiners is that 12 candidates have scores in excess of 70%, suggesting that if they are properly prepared then quality outcomes are possible.

The response of the candidates suggested they were ill-prepared for the examination or did not have sufficient experience of the subject matter. In Question 1, for example, a candidate needs to be comfortable in using the various UML diagrams. They also need to be skilled at reading the knowledge contained in a UML diagram. This ability to interpret these diagrams is only achieved through regular usage.

This paper gives a fair coverage of the syllabus. Therefore it is our intention that the next paper we author will have the same architecture and the same style of questions. Centres that prepare students should be advised of this and be aware of our observations here.

Question 1

1. The Unified Modelling Language (UML) class diagram in **Figure 1** below models a software house that employs programmers and project leaders.

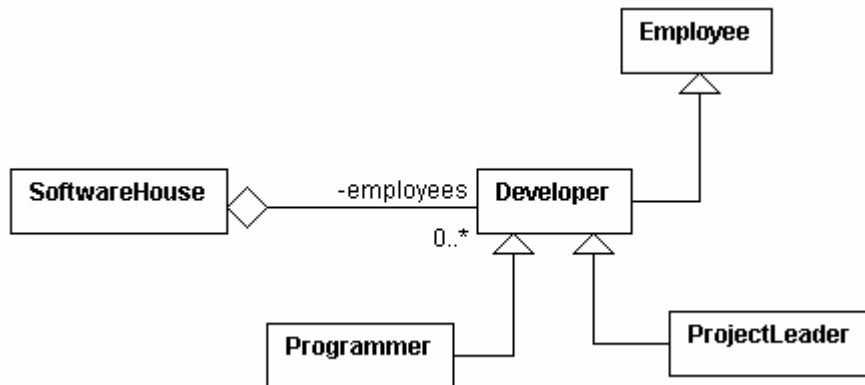
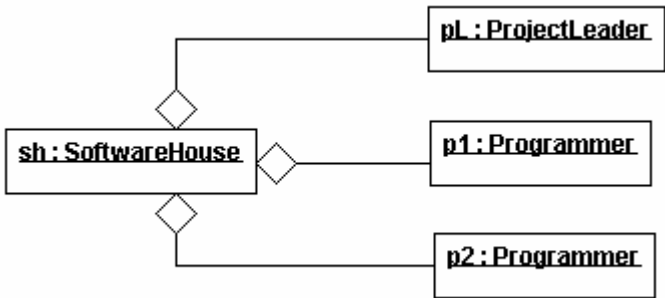
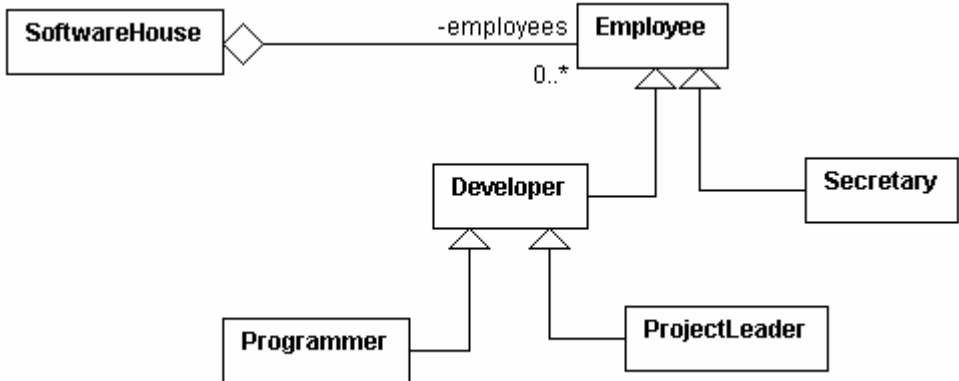


Figure 1

- a) Construct a UML object diagram showing one **ProjectLeader** and two **Programmers** working for the same **SoftwareHouse**. Give a short explanation of the diagram. **(5 marks)**
- b) If the two **Programmers** and the one **ProjectLeader** are present in the **employees** collection, then explain if the **SoftwareHouse** needs to distinguish between these differing types of developer. **(5 marks)**

- c) Revise the above class diagram to introduce a **Secretary** class, representing an employee not involved in any software development activities. Explain the principal revisions that have been made to the diagram. (5 marks)
- d) Revise the above class diagram so that a **ProjectLeader** is given managerial responsibilities for a team of developers. In your scheme explain how a management hierarchy of **ProjectLeaders** would be possible. (5 marks)
- e) Revise the object diagram from part 1a) showing both **Programmers** being managed by the **ProjectLeader**. (5 marks)

Answer Pointers

Question		Mark
1	This question examines Part 3 (Design) of the syllabus	
(a)	As a consequence of the specialisation hierarchy, both Programmers and ProjectLeaders are managed by the employees (role) set. 	5
(b)	A distinguishing feature of specialisation used in OO systems is that the SoftwareHouse would not have to distinguish between either type. The methods describe by the class Developer can be sent to each without concern for what actual type they represent. Method overriding in the class hierarchy might result in differing behaviours, but the SoftwareHouse is not concerned with this.	5
(c)	The first revision introduces the class Secretary as a subclass of Employee . Importantly, the SoftwareHouse class then forms a relationship with the Employee class so that Secretary , Programmer and ProjectLeader objects can be part of the employees set. 	5

(d)	<p>Here, a group of Developers are made team members of a ProjectLeader. Since both Programmer and ProjectLeader are subclasses of Developer, then a ProjectLeader may be a team member of another ProjectLeader.</p>	5
(e)		5

Examiner's Comments

There is strong evidence from the answers given, that the candidates are unable to interpret and infer the knowledge present in the class diagram. For example, when answering part (a) too many candidates included Developer and Employee objects! Further, the object diagram they presented mixed in class diagram elements such as specialisation!

Candidates need practice of working with class, object, collaboration, sequence and use-case diagrams. They also need to demonstrate that they can interpret the information presented by such diagrams.

The candidates were also poor at extending the class diagram as required in parts (c) and (d). For example, the Secretary class required in part (c) was correctly a specialisation of the class Employee but the relation between the SoftwareHouse and the Developer remained so that a Secretary could not be made and employee of the organisation.

Part (b) did not seem to be understood. The question sought to identify that the SoftwareHouse did not need to distinguish between Programmers and ProjectLeaders since they would all be considered some kind of Developer and they would all engage in the same messages.

Question 2

2. a) Carefully distinguish between the terms *subclass* and *superclass*. When combined they give rise to a *class hierarchy*. Why is a class hierarchy important when modelling object oriented systems? **(5 marks)**
- b) What do you understand by the term *abstract class*? **(2 marks)**
- c) Draw a revision to the class diagram given in question 1, clearly distinguishing those classes that have been changed into abstract classes and explaining why they have changed. **(6 marks)**
- d) What do you understand by the term *interface class*? **(2 marks)**
- e) Offer a strong argument why object oriented software should be developed in terms of interfaces **(4 marks)**
- f) Revise the class diagram developed in part 2c) to show where interfaces would be introduced into the scheme. **(6 marks)**

Answer Pointers

Question		Mark
2	This question examines Part 2 (Concepts) of the syllabus	
(a)	A superclass represents a generalisation. A subclass is a specialisation of a superclass and inherits all the features of its superclass. This relationship can be repeated any number of times giving rise to a hierarchy of classes. Class hierarchies support code reuse and support object substitution whereby an object of a subclass can be used where an object of the superclass is required.	5
(b)	An abstract class is used to establish other classes. There is no intention of making object instances of it. It is a way to guarantee that all subclasses share a common set of features.	2
(c)	<p>Since there are no actual examples of Employee or Developer then we consider these as abstract classes.</p> <pre> classDiagram class SoftwareHouse class Employee["«abstract» Employee"] class Developer["«abstract» Developer"] class Programmer class ProjectLeader SoftwareHouse o-- "0..*" Developer : -employees Developer < -- Programmer Developer < -- ProjectLeader Developer < -- Employee </pre>	2 4
(d)	A class in which none of the operations have a defined method and has no state information is referred to as an interface. All operations are deferred to subclasses to implement. Effectively, an interface presents only a specification of its behaviours.	2
(e)	Since an interface has no implementation then different subclasses can be defined with differing implementation strategies. Code written to that interface is then free to adopt any of the concrete subclasses, even where they differ in their implementation.	4

Examiner's Comments

Parts (a) (i) and (ii) elicited poor and very poor responses. Too many candidates seemed not to understand the role of the use-case diagram. The architecture centric question had a very poor response with few able to express its intent.

Part (b) was intended to be an easy question and as a result it was answered reasonably well. However, it was clear from the responses to part (c) that few candidates were familiar with or regression testing. Many did not answer the question and described unit testing in general.

Question 4

4. a) Give definitions of the following:

- i) abstract data type
- ii) modular programming
- iii) structured programming
- iv) typed languages
- v) untyped languages

(15 marks)

b) Choose THREE of the above concepts and discuss how each has contributed to the development of object oriented languages.

(10 marks)

Answer Pointers

Question		Mark
4	This question examines Part 1 (Foundations) of the syllabus	
(a)	(i) A set of data values and associated operations that are precisely specified and are independent of any implementation.	3
	(ii) A style of programming in which large programs are written as several small subprograms. A subprogram is either a procedure or function. Each subprogram performs a single task and is normally short in length. A module is a grouping of related subprograms. A major benefit is the separation of concerns into modules.	3
	(iii) A technique for organizing and coding computer programs in which a hierarchy of modules is used. Normally, each has a single entry and exit point. Control is passed downward through the structure without unconditional branches to higher levels of the structure.	3
	(iv) With a typed language checks that the rules pertaining to the use of a certain type (datatype) are possible. Typically checking is done statically by the compiler. The extent to which the rules are enforced are determined by whether it is strongly or weakly typed. For example, with strong typing if a parameter is specified to be of a certain type then the actual parameter must be of that type. With weak typing a subtype of the formal parameter would be acceptable. Typing brings confidence that during execution there will be no "surprises". However the price to pay is flexibility.	3
	(v) With an untyped language checks made that each call can actually be executed are relaxed. Often any checks made are made at run-time i.e. dynamically. This means that the programmer has a great deal of flexibility in the code that is written. However the price to pay is one of safety.	3
(b)	The ADT leads to the class (a concrete data type) and objects (instances of the type) represented by the abstractions of the problem domain.	3
	Prior to the advent of OO, code was typically organized as a collection of modules, each consisting of a collection of subprograms. Modular programming leads the notion of a set	3

	<p>of related operations for a class.</p> <p>Structured programming leads to method bodies with sequence selection and repetition control structures.</p> <p>The ideas inherent in the use of typed languages lead to polymorphism in OOPs. Some of the constraints of typing are relaxed but some important compiler checks are still possible. For example, checks on the type (class) the actual substitutions made. The same applies to casts. This gives us the balance between type safety and flexibility.</p> <p>Experience with untyped languages leads to the desire to relax typing rules because they are so flexible. For example, in Smalltalk and Java there is the class Object that is the root of all classes.</p>	4
--	--	---

Examiner's Comments

Part (a) of this question was intended to be relatively straightforward and it proved to be so. It was in fact one of the few questions that seemed to be answered well by most of the candidates.

Part (b) by contrast was disappointing with many of the students missing the point of the question.

Question 5

5. a) Briefly explain what is meant by the term, *Design Pattern*. (3 marks)
- b) Explain how an understanding of *Design Patterns* helps the following people:
- i) computing students
 - ii) inexperienced software developers
 - iii) experienced software developers
 - iv) software maintainers
- (12 marks)
- c) Describe a *Design Pattern* with which you are familiar. Your answer should include the motivation for the existence of the *Design Pattern*, its structure, participants and consequences of its use. (10 marks)

Answer Pointers

Question		Mark
5	This question examines Part 3 (Design) of the syllabus	
(a)	A solution to a problem in a context. A design pattern captures in an elegant fashion examples of best-practice as practiced by expert software developers over many years.	3
(b)	(i) Design patterns illustrate how certain types of problem should be approached. This shortens the learning curve and acts as an excellent source of exemplars. Typically the documentation for a Design Pattern has code samples. Again they should what can/should be done. Discussions of the forces that are taken into account are also very useful from an educational point of view.	3
	(ii) Exposure to Design Patterns builds on experience already gained. It may even give confidence that an inexperienced designer was "doing it right" all along. More likely it will widen the developers horizons and make it easier to start or take part in a major project. In this respect the discussion of the advantages/disadvantages/implications/related patterns are particularly useful.	3
	(iii) Design Patterns provide a useful vocabulary for experienced developers. It allows them to communicate at a high level of abstraction without having to worry about unnecessary detail. They have access to a library of properly documented, tried and trusted Design Patterns. It is even possible that they may introduce new Design Patterns and pass on their experience to others.	3

	(iv) Maintenance programmers find it easier to understand what the software does if they understand the Design Patterns on which the software is based. It also helps them make sensible modifications if they appreciate the overall intent and structure of the design.	3
(c)	<p>A typical Design Pattern selected would be Decorator, Iterator, Observer or Singleton. For example:</p> <p>Observer Motivation There is a need to maintain consistency among co-operating objects. However they must not be tightly coupled as they become less re-usable.</p> <p>Structure</p> <pre> classDiagram class Subject { +void attach(Observer observer) +void detach(Observer observer) +void notify() } class ConcreteSubject { -State subjectState +State getState() } class Observer { +void update() } class ConcreteObserver { -State observerState +void update() } Subject < -- ConcreteSubject Observer < -- ConcreteObserver Subject --> "0..*" Observer : -theObservers ConcreteSubject --> ConcreteObserver : -theSubject </pre> <p>Participants Subject – knows its observers and provides an interface for attaching/detaching observers. Observer – defines an updating interface for objects that should be notified of changes. ConcreteSubject – stores state that is of interest and notifies observers of state changes. ConcreteObserver – references a ConcreteSubject, stores its own state that should be consistent with the Subject’s and implements the Observer updating interface.</p> <p>Consequences Subjects and observers can be varied independently. Therefore subjects can be re-used with re-using their observers and vice versa. It lets you add observers without modifying the subject or other observers.</p>	<p>2</p> <p>3</p> <p>2</p> <p>3</p>

Examiner’s Comments

This was a popular question and it was clear that many of the candidates had been well prepared for parts (a) and (c). It was mainly “book work”. However, it was refreshing to find that many candidates managed to use their knowledge of Design Patterns to answer part (b).

Question 6

6. a) A requirement of object oriented systems is to manage a collection of objects.
- i) Describe how collection classes are used to realise this requirement.
 - ii) Give examples of two collection classes with which you are familiar.
 - iii) Explain how a particular collection class might maintain its objects as an ordered collection. **(12 marks)**
- b) Using a suitable example, explain the essential differences between specialisation and delegation. You should use a programming language of your choice to illustrate your answer. **(5 marks)**
- c) Consider the following scenario:

A class **Vector** is a collection class that holds its elements in the order in which each element is added. Each element has a unique index associated with it. Indices start at 1 and increase by 1 each time a new element is added. For example, the first element added has an index of 1, the second 2 and so on. The **Vector** class also has a method **get** that is used to retrieve a particular element from a **Vector**. When supplied with an integer representing an index, the method **get** returns the element with that index. For example, **get(1)** would return the first element in the **Vector**.

You are required to use the **Vector** class in the construction of a **Queue** class. This new class should mimic a queue. As with the **Vector**, it holds its elements in the order in which each are added. However it has a method **front** to return the first element added i.e. the element at the front of the queue. It also has a similar method **back** to return the last element added i.e. the element at the back of the queue. Crucially, it should not be possible to access elements at intermediate positions in the queue.

Discuss whether you should use delegation or specialisation to develop the **Queue** class. Give a detailed explanation of the reasons for your decision. As before, you should use a programming language of your choice to illustrate your answer. **(8 marks)**

Answer Pointers

Question		Mark
6	This question examines Part 2 (Concepts) of the syllabus	
(a)	(i) The collections classes are used to maintain a reference to a collection of objects. They are frequently required by OO applications such as the employees that work for the software house in Q1. The collection classes include the Java container library and the C++ STL containers. These classes remove the burden of maintaining the object collection.	3
	(ii) Java supports the ArrayList and the TreeSet . The former maintains the items in the order they are entered into the collection. The TreeSet collects its items in some determined sort order.	3
	(iii) A container such as the Java TreeSet maintains a sorted collection of objects. This it does, not by knowing how to sort the items. This would restrict the container to a certain sorting scheme. Rather, the container relies on the objects to conform to a Comparable interface so that the objects can be sent the compareTo message from which the ordering is determined.	3
(b)	With specialisation, a child (subclass) inherits all of the features (methods and attributes) of its parent (superclass). Only one object exists. However, with delegation object composition is used. The delegate is held as an attribute (variable) i.e. two objects exist. Therefore a message received by an object can be delegated (relayed) to a delegate. For example using Java, we might have: <pre>public class Parent { public Parent() { // ... }</pre>	

	<pre> public void operation_1() { //... } public void operation_2() { //... } public void operation_3() { //... } // ... } public class Child extends Parent { public void operation_1() { //... } // replaced public void operation_2() { //... super.operation_2(); // redefined } } and public class Delegator { public Delegator(Delegate aDelegate) { theDelegate = aDelegate; } public void operation_1() { //... } public void operation_2() { //... theDelegate.operation_4(); } public void operation_3() { theDelegate.operation_5(); } private Delegate theDelegate; // ... } public class Delegate { public void operation_4() { //... } public void operation_5() { //... } // ... } </pre>	5
(c)	<p>(ii) Specialisation would not be appropriate. The main reason is that it is not possible to disinherit a method. In this case the inherited method <code>get</code> that takes an index as a parameter does not apply to a queue. It would be possible to redefine it but this is not really acceptable. A much better solution is to have a <code>Vector</code> attribute in the <code>Queue</code> class and delegate to it when implementing the <code>front</code> and <code>back</code> methods. For example, using Java we might have:</p> <pre> public class Queue { public Queue() { theVector = new Vector(); } public Object front() { return theVector.get(1); } public Object back() { return theVector.get(theVector.size()); } public void add(Object element) { theVector.add(element); } //... private Vector theVector; } </pre>	4 4

Examiner's Comments

This question was not attempted by many candidates. A similar observation was made at the last sitting. Seems that candidates do not like questions with too many words!

Part (a) invariably produced a collection of methods as the answer! Given that most OO solutions involve collections of objects it was expected that the candidates would be comfortable with the basic container classes needed to answer this question.

On reflection, parts (b) and (c) might have been too challenging for the candidates. However, several did manage to give good answers.