COMP23420 semester 2 exam performance feedback.

Original marking scheme in **bold**, additional comments **in bold italic**.

Feedback on question 1.1 - 1.5 and question 2 from Liping, 1.6-1.10 and question 3 from John except where otherwise noted.

Question 1

1.1. 1.2 Overall students performed well and proved to understand the theoretical concepts presented in the course (i.e difference between domain object model and design class diagram, micro vs macro development process)

1.3 Most students have failed to differentiate between OOA artefacts and OOD artefacts.

1.4 Majority of the students knew the architecture for their IBMS project, however, only a handful of them were able to relate the different layers of the architecture to specific components in their project.

1.5 Most student gave a description instead of stating the purpose of requirements elicitation. To differentiate between FRs and NFRs, most students have used examples and they have demonstrated their understanding of the requirements engineering concepts.

1.6. Consider the following design classes for the part of a restaurant management system concerned with staff members:

public abstract class StaffMember

public class Waiter extends StaffMember
public class Chef extends StaffMember
public class Manager extends StaffMember

Assuming that it is necessary to store information about staff members, what is the potential problem with this design? (2 marks)

This is using inheritance to model roles. The problem with this is that roles change over time and one person may have multiple roles. [1] This means that a single real-world object can correspond to multiple software objects, leading to redunandancy and potential inconsistency [1]

I also allowed: Adding new roles requires new subclasses to be added.

Mostly this was well answered, although a number of people had a serious misunderstanding of abstract classes - thinking that they can't store data so it has to be duplicated in the subclasses - while in fact part of the point of inheritance is to allow common attributes and methods to be stored in a superclass.

1.7. Briefly explain how different design can avoid this problem and suggest another advantage of this alternative design. (2 marks)

Have a single StaffMember class with a list of roles associated with it. Then each staff member is represented by exactly one StaffMember object.. [1] It then becomes easier to add more, and more fine-grain roles, e.g. StockManager, PublicityCoordinator.

A number of people suggested using a list of enums, which is a good idea (particularly in Java where enums are actually classes). Alternatively the roles can be subclasses of a Role class.

Many of the answers were vague, talking about adding attributes to StaffMember without being clear what these attributes would do. There were some ugly solutions, e.g. storing all the possible roles in StaffMember and having an array of Booleans to indicate which roles a person has. While this could be made to work it indicates a lack of understanding of what attributes are for.

A number of solution were so unclearly stated that I couldn't give any marks for them. When you're asked to explain a design it's essential to do so clearly, whether in an exam or in real SW development.

1.8 In the naïve approach to testing a programmer writes the code and then tests it. Briefly explain two other approaches and their advantages over the naïve approach. (2 marks)

Traditional: have specialist testers who are motivate to break the software (testing to fail) while the programmer's motivation is to show that it works. [1] Agile: test-driven development, writing the tests before the code. Ensures only functionality which is really required is implemented [1] (Other advantages of TDD also get the mark)

A lot of people gave answers relating to other aspects of testing (e.g. black box vs. white box) so not answering the specific question.

1.9 Briefly explain, using an example not given in the course, the difference between *Integration Testing* and *System Testing*, and why it matters. (2 marks)

Integration testing tests that all the components of the system work together *in the development environment.* While system testing does so *in the deployment environment.* [1] This matters because the environments may be very different, e.g

in the restaurant system mentioned above, the waiters may use PDAs with limited capability. [1] (Any reasonable example gets the mark).

A lot of people said that system testing was testing the whole system rather than part of it. One of the reasons I specifically talked about this was that you might think that's what it means but actually it doesn't.

1.10 "Physicality" is considered important in Agile software development. Give two specific examples of Agile practices which promote physicality.

In a retrospective one of the tasks of the facilitator is to monitor the body language of the team to ensure that everyone is actively engaged. [1] User stories are written on physical cards which encourage people to get together and play with them. [1] (Other examples such as stand-ups also ok).

Most people were ok on this.

Question 2

There were 83 students who answered Question 2 and their performance to this question is summarised below.

2.1 Given that the question is aided by a use case diagram, I had expected all students to do well. However, only a few (less than 10) students got it completely right. Most students still cannot identify all the actors.

2.2 The answers to this question vary widely from excellent to no clue. I have marked this question very generously in order not to fail too many students. The results show that there are about 1/3 students who don't know how to write a use case specification. I suspect that students who did well on this question are the ones who wrote the use case specification in their team projects. I believe there is a strong correlation in project and exam performance.

John adds: I agree, and it also shows that everybody doing what they are best at might be a way of maximising lab marks but is not a good overall strategy.

This is also a straightforward question, but still some students have no clue what domain objects are.

The majority of the students did poorly in this question. I believe there are two reasons for this: 1) Some students still don't know object-oriented design and 2) some students found the drawing tool in the online exam package very difficult to use and consequently produced unrecognisable diagram.

John comments: I see no sign of 2) but if it does occur it can only be due to failure to practice with the tool, as drawing simple class diagrams with it is trivially easy.

Question 3

3.1 Explain the GRASP principles of High Cohesion and Low Coupling and how they are related. Your answer should distinguish between two different kinds of coupling. (5 marks)

[Bookwork] High cohesion means that each class represents a single well-defined entity. [1] Low coupling means that classes should not be overly dependent on each other, but it is not just the number of classes which depend on each other but the nature of those dependences – they should be as simple and stable as possible. [1] ***Internal*** **coupling is between classes within a subsystem. [1]** ***External*** **coupling is between those classes and the rest of the system. [1] They are related because cohesive classes have simple stable interfaces and hence lead naturally to low coupling [1] (The converse is also true and gets the mark).**

Also a mark (but only one) for explaining clearly why HC/LC is good (important but not explicitly asked for in the question).

A number of people said that internal coupling is coupling within a class (as opposed to a subsystem). This makes no sense as coupling by definition is a relationship between classes.

3.2. Constructor Ltd makes construction kits for children. Kits consist of parts of various different kinds. They can be broadly classified as Simple parts, such as cogs and levers, and Compound parts which are pre-built from simple parts. Simple parts have properties such as colour and, of course, cost. Your team is building a software system to enable the company to build and sell kits more efficiently.

3.2a A junior software developer joins your team. You don't know anything about him so you set him a test – you ask him to come up with design classes corresponding to the description of parts etc. above. He comes back with a design that contains a single class, called Parts. He explains:

"I used just one class so it's nice and cohesive, and to minimise coupling. I also found an opportunity for code reuse, so here I have a list which contains parts for a kit, or simple parts for a compound part or features for a simple part. This instance variable TYPE tells you which type of things a particular instance represents."

Explain in terms of the GRASP principles of High Cohesion, Low Coupling and Protected Variations why this is a very bad design. (3 marks)

Clearly the class is not cohesive as it represents three different things. [1] It has high external coupling as code using the class will need to do so through a complex and unstable interface. [1] It therefore has poor protection against variation as any changes will affect code using the class. [1]

There were a lot of muddled answers to this part. Although many people were able to define internal and external coupling in the bookwork above, many failed to apply it correctly to this example. Internal coupling is not a useful concept here as the entire subsystem is contained within one class. There were also a lot of misunderstandings of PV. Also, a lot of people wasted time by suggesting alternatives, which is not what this part of the question is about.

3.2 b Draw a class diagram which shows a better design based on the GRASP principle of Polymorphism. (5 marks)

The design should be based on an abstract class Part. A Kit contains Parts. SimplePart and CompoundPart inherit from Part. A SimplePart contains Features. A Componund part contains SimpleParts (Not Parts so this is not quite the Composite design pattern, but solutions which do this will not be penalised). Can also have Cog and Lever inheriting from SimplePart. Contains relationship may be represented either as association or as List attributes. in the containing classes. Marks deducted for notation abuse or extraneous detail.

Indeed many marks were deducted for notation abuse. There are plenty of examples of correct UML notation (particularly inheritance and contains relationships) on the lecture slides. In general answers to this part were very poor. Many did not even recognise the need for an abstract Part class. In general the answers to this question show clearly the difference between reading the lecture slides and actually attending the lecture.

3.2c Explain how the use of Polymorphism has yielded a design which conforms to several other GRASP principles. (4 marks)

Each class now represents a single well-defined entity so the design is cohesive. [1] Internal coupling is low, e.g. the Kit class depends only on the abstract Part class which will have a simple and stable interface[1] External coupling is also good as external code depends only on Kit (and probably Part). [1] Protection against variation is also provided as e.g. adding different simple parts or changing the implementation of one of them does not affect other code. [1]

Marks also for other sensible points, e.g. Information expert – CompoundPart has the information to deal with lists of Parts etc.

A detailed discussion of polymorphism is not appropriate here as the question specifically asks about other GRASP principles. In particular reciting the three rules for inheritance is a complete waste of time.

Note the Pure Fabrication is not involved here as Part is a domain concept.

3.2d Explain how a method could be implemented to calculate the total cost of a Kit from the costs of the parts it contains. You do not need to write the code, just explain clearly.(3 marks.)

Part would have an abstract method getCost(). Kit would call this for each part in the kit and add them up. [1]. Each SimplePart would store its cost as an and SimplePart would implement getCost() by returning the stored value. [1] CompondPart would get the cost of each of its component parts and add them up.[1]

Solutions which treated SimpleParts and CompondParts correctly but didn't make use of inheritance got a maximum of 2/3. Solutions which stored the cost as an instance variable in the Part class were accepted provided it was clear where this value came from for CompondParts.