COMP23420 exam performance feedback (JS)

Question text in plain text, original marking scheme in **bold**, additional comments in ***bold italic.***

Note: this is the feedback for John's part only, but some of Liping's questions are retained for reference. See Liping's feedback for her questions.

Question 1

(3 marks)

d). An important principle in agile software development is *Physicality* - physical interactions are considered better than virtual interactions. Explain why this is. (2 marks)

Physical interactions are richer - we can synchronise sight and sound and monitor facial expressions, body language etc. [1]. When interacting virtually (e.g. by email) people often adopt, consciously or otherwise, persona that are less helpful than when you meet them ftf. [1]

Marks for answers which include the something along the lines of the first point above and one other good point.

Mostly this was answered well.

e). Give three examples of agile practices which are consistent with the principle of Physicality, and for each state whether your team did it or not, and why.

e.g. stand-up meetings, programming in pairs, retrospectives, diagramming at a whiteboard. Marks only for answers which address clearly the last part of the question.

(3 marks)

Most answers did address the last part, but many people failed to give three distinct practices (e.g. I didn't count stand-up meetings and just "meetings" as two different practices. A lot of people said they did stand-up meetings, which I allowed of course, even though I saw zero evidence f this in the lab! A common answer was that meetings were held sitting down rather than standing up because that way you can see the computer screen. I allowed this, although it rather misses the point of stand-ups which is to get away from the computers for a short time and discuss how things are going, what needs to be done next etc.

f). Why is it not a good idea for a programmer to write some code and then test it? (1 mark)

Because a programmer expects their code to work and may not Test to Fail sufficiently.

Most people got this right.

g). State two alternatives approaches to testing which overcome the problem of part h.. (2 marks)

Traditional approach: have a separate testing teams. [1]
Agile approach, TDD - write the tests *before* the code [1]

Most people got the second of these, but a lot missed the first. I gave the mark for any answer which clearly implied independent testing. For example I allowed having one person in a pair doing the programming and although writing the tests, although that is not really correct - normally in pair programming both members are involved in writing the code and therefore both are likely to have an emotional attachment to it.

h) Give an example *not* given in the lectures of a JUnit fixture. Hint: you need a JUnit annotation and a small piece of Java code. Minor errors in the code will not be penalised. (2 marks)

e.g. something like this will connect to a database before running a series of tests.

```
@BeforeClass
public static void setUp() {
  connectToDatabase();
}
```

This was intended to separate the people who had really been paying attention from the rest and it certainly did that. Most people wrote a test, not a fixture, and an alarming number of people didn't get that anywhere near right. I gave one mark for a test that clearly had a fixture at the start of it. Of those who did have the right idea, almost all included that "teardown" part (@After or @AfterClass) as part of the fixture. I allowed this although it is not correct (only the setup art it the fixture - it sets up the fixed conditions under which the tests will run) because it looked like I hadn't made this clear in the lecture.

A note about how Questions 2 and 3 were written. Liping wrote the first half, and then I wrote the second half in a way which was intended as far as possible to follow on and make for a coherent question. I've left in those parts of Liping's halves which are relevant to mine.

Question 2

Answer this question OR question 3

This was the more popular question, maybe 60/40.

b) Explain, through an example, how the Role pattern works.

(5 marks)

c). Briefly explain the role of GRASP principles in object-oriented software development.
(2 marks)

A good answer is

They provide a set of principles for assigning responsibilities to classes, the most difficult skill in OO software development.

+ one other good point, e.g. Like all patterns they form a language which helps developers to communicate They provide guidance on when to refactor and how..

Note that there were no marks for simply listing GRASP patterns at this point, which a lot of people spent a lot of time doing. That does not answer the question "Briefly explain the role of…" In general more marks were lost in this and subsequent parts due to poor exam technique rather than lack of knowledge of the patterns. Always read the question carefully and answer the question actually asked, not the one you expected/hoped for. What answers to this half did show that many people had a good grasp of GRASP which was pleasing.

d). Explain how the Role pattern which you explained in part b is consistent with **five** different GRASP principles. (5 marks)

High cohesion - factoring out extrinsic behavior makes classes more cohesive, we have a single well-defined entity rather than entity + roles
Low (internal) coupling - the entity is loosely coupled to its roles
Information expert - the role information is localized in the role class hierarchy
Polymorphism - roles are (in the general case) represented as a class hierarchy, and hence…
Protected variations - roles can be added/changed freely without affecting anything else.

A major problem arose with the making here because at least half the people who answered this question didn't actually know the Role pattern (both of us taught it although I didn't use that name - see my footballer/position scenario for one example). A lot of people therefore got zero for part b and I didn't think it fair to double-penalise them. I therefore gave marks for any coherent explanation of GRASP principles in whatever people said for part b. Since many answers t part b were vague and/or confused, this was not easy. I don't think anybody got 5/5 on this basis.

Of those who did get the Role pattern right, the only common mistake was to say that the Role class was a Pure Fabrication. It is not - it's a domain class. The Role pattern simply represents the domain better than the naïve alternative (e/g/ a footballer can play in several positions at different times, not just one for all time). In general, a common mistake is to think that an

abstract class must be a PF. In some cases it is, but if it represents a generalization in the domain, as in this case and many others, it is not.

e). Explain how your implementation of the roster and the controllers' user interface was consistent with the GRASP principles of Information Expert, low external Coupling and Protected Variations (or if it was not, explain why not).. (3 marks)

Hopefully you had model-view separation, possibly with a controller. Answers will of course depend on exactly how you did it, but I'd expect something along the lines of:
Information expert: e.g. the UI has the information required to display a roster, a roster-generator class has the information required to generate a roster.
Low external coupling: (low coupling between subsystems as opposed to internal, coupling within them) the coupling between the roster and UI subsystems is via a clean and simple interface (or via a controller). Answers which explain low external coupling with other systems are also good.
Protected variations - it should be possible to display the roster in different ways without affecting the roster code at all.

Not e that is explicitly says low *external* coupling, so to get that mark you needed to talk about coupling between subsystems (which most people did), not between individual classes. This was generally answered well. Interestingly, many people said that although their design had low external coupling and followed the Information Expert, principle, it didn't have PV - making changes had been difficult. Which just goes to show how difficult software design is (and how important it is to refactor).

Question 3

Answer this question OR question 2

You are designing a point-of-sale software system for a convenience store. You want to apply some of the business application patterns you have learned from this course to your design.

a) Design a UML class diagram to show the relationships between the following three business domain objects: Sale, Product and Line Product. Your diagram should show: (1) the multiplicities of these relationships; (2) essential attributes of these objects and essential operations performed by these objects.

(4 marks)

b) Explain the business application design patterns used in your design.

(2 marks)

c) Extend your design by showing cash and card payment and explain new patterns used in this extension.

(4 marks)

d). Other than sheer size, what are the main characteristics of a piece of software which affect how easy or difficult it is to test? Illustrate your answer by briefly explaining how difficult or otherwise it would be to test 1. The subsystem you designed for part c, and 2. The whole sales system of which it is a part. (5 marks).

A good answer is:

The better the design, the easier it is to test. [1]
The more parts of the system can be tested in isolation the easier it is to test (conversely, the more interactions between subsystems the harder it is. [1]
Concurrent and/or reactive systems are much harder to test than sequential ones, especially if they are safety critical. [1]
The subsystem of part c is a very easy case, as I has no concurrency etc., it can be tested in isolation and the classes can be tested separately. [1]
The system overall is a fairly hard case as it does have concurrency etc. and while not safety-critical is mission-critical. [1]

Sensible variations on the above are acceptable, but the question is about characteristics of the software, so practices such as programming in pairs are not relevant in this part (they are in the next).

For the first part of the question, many people fixated on the first part, in some cases giving a list of GRASP principles, and missed the other parts completely. For the second part, many people failed to read the question correctly. The question did not ask you to compare the testing issues of the design in part a) with that in part c). It asked you to compare part c with "The whole sales system of which this is a part". A very small part, when you consider that this is only part of the business logic layer of such a system. There will be user interfaces, database issues, training/usability issues (part of System testing) additional functionality such as logging, and management functions of various kinds, concurrency issues etc. This will be far harder to test than the subsystem of part c).

A number of people pointed out something I'd missed, though. The card payment option requires interaction with an external system, which makes testing of this part a great deal harder.

e). Estimate how many bugs there are likely to be left in your IBMS implementation. Explain your reasoning, take into account the estimates of bug density given in the lectures and the factors relevant to bug density relevant for your project. (5 marks)

The relevant estimates are: 1-10 bugs per hundred lines, of which 90% may be found by debugging (which they did do) of which 90% may be found by "routine" testing (which they probably only partially did). Factors affecting bug density include programmer experience and whether they practiced programming in pairs.

First they have to estimate how many lines of code they have. Let's use 5,000 as an example. That's 5-50 bugs after debugging. In theory that's 0.5-5 after testing, but it's unlikely that they did the amount of testing required to get that order of magnitude. If they have a clean design, tested really thoroughly, and programmed in pairs, there's a chance their code could be bug free. Much more likely, given the relative inexperience of most members of most teams, and the strong time constraints on the amount of testing done, and nasty edge-cases in the data, there will be several bugs remaining, where "several" could easily be 10 and might be as many as 50. A good answer will combine the estimates given in the lectures with the reality of their experience.

Many people didn't remember the estimates of bug density correctly (or didn't mention them at all). Many people came up with silly numbers (if there are hundreds of bugs in your code, nothing will work; almost all teams came up with solutions which worked fairly well). Strangely, there was little intersection between these two sets - people tended either to have a reasoned argument leading to silly numbers, or a poor (or non-existent) argument leading to sensible numbers! The upshot of this was the most people got some marks for this part (typically 3/5) but very few people got full marks.