UG Exam Performance Feedback
Third Year
2016/2017 Semester 1

COMP33711 Agile Software Engineering Suzanne Embury

Comments Please see the attached report.

Feedback on COMP33711 January 2017 Exam

Suzanne M. Embury
February 2017

Q1.

There was a wide variety of answers to this question, and I accepted any principle provided that a plausible justification was given to link it to the practice. To earn two marks, answers had to go beyond the obvious links, to demonstrate a deeper understanding of the practice and the principle. For example, many people identified principle F at work in the practice of daily stand-ups, and justified it by saying that stand-ups involve the team gathering together for a brief conversation. This earned 1 mark. Others earned the full 2 marks by explaining further that stand-ups must be brief, while conveying the core status information accurately in a low cost way. Face-to-face conversation fits this requirement much better than other forms of communication.

The main cause of lost marks was stating that a link between a practice and a principle exists but not explaining what the link was. For example, some candidates simply wrote out the principle in their answers, and did not add any information beyond what was already present in the appendix. Just stating that task boards ensure frequent delivery of working software does not make it true. (It is not, in fact, true. Task boards do not impose any specific iteration length on a team. They can be used with release schedules of 6 months as easily as with release schedules of 6 days.

Other marks were lost by candidates mixing several practices together in their answer, simply because those practices are typically used together. For example, there were many answers linking the task board practice to principles when the actual link was from the user stories practice to the principle. It is not compulsory to use task boards with stories. They can be used with a variety of card types.

Some candidates lost marks by talking about the practices generally and failing to actually name any of the principles. Others ignored the letters I had conveniently attached to the principles for easy reference and instead slavishly copied the text of the principle into their answer book. In some cases, the copying was performed rather erratically, and I had to guess which principle was meant!

Finally, I will mention 2 common errors in the answers for two of the practices.

The first is the daily stand-up practice, which a worrying number of candidates associated with the final principle (L) on the need for regular reflection by teams. Daily stand-up meetings need to be very quick and to the point. There is no time for in-depth analysis, and no time for reflection on the team's processes. They are *planning* meetings, and are concerned with how the current plan is working and what immediate changes might be needed to the plan. If a problem is raised, the only action in a standup is to identify the team members who will address it, and get them to schedule a meeting after the stand-up, when they can take the time to find the right solution. Principle L is usually implemented through retrospectives, which are longer meetings and held less frequently. (Reflecting on the team's processes every day would be exhausting, and would be unlikely to be productive.)

The second point is that a number of candidates clearly had only a hazy understanding of what the TDD practice involves. It is nothing to do with acceptance tests, or tables. It is not just a commitment to "do some testing" alongside ordinary development. It is not even the practice of writing lots of tests first before implementation starts (that would be test-first development). Naturally, answers based on these misunderstandings rarely contained convincing justifications.

Q2.

This was the most popular question in section B. It had the lowest average mark of all the questions, equally with question 3.

a) This question proved more challenging for candidates than I had anticipated. It was intended as an easy warm-up question, but it also had the effect of revealing a number of basic misunderstandings of some of the agile practices involved.

In part (i) the biggest mistake mad by candidates was to tell the team to get an on-site customer and to order their stories by value. But the question said nothing to indicate that they did not already have an on-site customer. This team had managed to create a story map for the system that filled a whole wall with stories! They are clearly getting a lot of input from their customer representative. The problem was not that they didn't know enough about the customer's needs, but that they had tried to capture it all at once, instead of using epic stories to package some of the less immediately valuable functionality in an easy to manage form.

A note on exam technique is perhaps worth while here. If candidates had said in their answers, "I assume that this team does not have an on-site customer, because…" and then justified that assumption (including explaining how the team could have written so many stories without a customer representative) I would have been happy to accept "get an on-site customer" as the answer. I would see from the justification of the assumption that the candidate understood the basic agile ideas, and so could assign marks. But I could not assign marks for answers that simply said "this team needs an on-site customer", without the question of where all the stories came from in their absence being addressed in some way.

Part (ii) seemed to cause many candidates problems, with some not writing anything at all in answer to it, and others perplexed as to what the problem was. Probably the most common error was (again) to talk about the importance of an on-site customer and to insist that this team got one, so that they did not need to get their requirements agreed by the senior strategy board. Some even suggested that the team drag one of these senior board members onto their team (as their full-time customer representative?) The question was ambiguous as to whether the "senior strategy board" was part of the customer organisation or the developer organisation, so I did my best to accept answers based on either assumption. However, it was not ambiguous as to whether this board was composed of "senior" people or not. It seems unlikely that anyone on a senior strategy board is going to take kindly to being put in a customer representative role, where a (more junior) end-user (end-beneficiary, at least) of the software is normally needed.

A number of candidates seemed to have been confused by the notion of a "real customer", and the idea that several solution options might need to be trialled before the team can make the right decision as to what they should actually build. Again, the "on-site customer" was touted as the solution. "Ditch these slow, expensive trials, and ask the customer what they want" was the gist of many answers. I had hoped to have conveyed the idea in the course unit that customers don't always know what they want, and even if they think they know it may not be what they need. This team was doing everything right, in trying to gather hard evidence, quickly, as to where they should sink their resources longer term. Only the bureaucratic processes involved in acquiring the money needed to buy the equipment and employ any needed personnel (e.g. for data entry) was the problem. (Some candidates should note that not all prototypes are made of paper! There can be genuine expenses involved in putting them out into the world for real users to use.)

Before leaving part (ii), I must mention the significant number of candidates who saw the word "document" in the question, and leapt to that as the cause of the agile disaster unfolding for team B. These candidates inveighed against "lengthy documents", and many indicated that they thought team B had *chosen* to write this massive 2 page tome. To repeat what was said in the course unit, documents *per se* are not the problem. Sometimes, a short and to-the-point document is exactly

what is needed. And sometimes interacting with the world outside the team means following processes designed by other people, that are not convenient for the team. A confident agile team will have no problem in producing short documents in these cases (as is made clear by the final sentence of the Agile Manifesto).

Part (iii) seemed to cause fewer problems in general, although many candidates took the opportunity to again suggest "on-site customer" as the agile practice this team needs (despite having given this answer for both previous parts, and despite the frequent mention in the question of not one but two customers working closely with the team on story writing). This team clearly has an on-site customer, so that can't be the problem. The question describes the customer talking to the developer about what stories they value, so the solution *cannot* be for "the developers to ask the customer about what stories they value".

Finally, for part a), a note on exam technique. The question asked for a diagnosis of the problem and an action the team could take to resolve the problem. That is, the answer needed to contain these two things for full marks. Some candidates provided only one of these two things, and so lost 50% of the marks immediately. It is always worth checking the wording of the question, to make sure you know what you need to provide, before you get started on the answer.

b) This part was answered well in general, although again a large number of candidates were let down by not reading the question properly. The question asked for two things: the events that led to cards moving between columns, and the practice that is being carried out when the event occurs. Many candidates simply described the meaning of each of the columns, and failed to state how or why cards move between columns. I did my best to extract events from the column descriptions wherever I could.

A correct answer this the question will have been organised around column pairs (Backlog -> Next Iteration, Doing -> To Verify, …) whereas incorrect answers were typically based around individual columns, without reference to the columns before and after them.

A common mistake made here was to give the practice of retrospectives as taking place when cards move from column to column. This is a fundamental misunderstanding of what retrospectives are for. They are not for "planning", but for adjusting the team's general processes in order to improve performance in the future. It is very unusual for cards to be moved on a story board during a team retrospective. (Nor are story points used to express the value of a story, as some students wanted to claim throughout the whole exam paper. Story points express story size, as any one who played the Planning Poker game in the lecture will be well aware.)

A further mistake was to give TDD as a practice during which cards are moved across the story board. To quote one exam script: "In a TDD team, a story tends to jump back and forth between the "Doing" and the "To Verify" columns". While this quote displays a strong understanding of what happens in TDD, it is not a correct description of how task/story boards are used alongside TDD. The columns on the board should show the key stages of the project, and should not try and track whether individual unit tests are passing or failing. (There are other practices for that!) Cards move out of the "Doing" column and into "To Verify" when the pair working on them believes them to be complete ("done") and ready for someone else to check over.

c) This part of the question proved a struggle for most candidates, as it was intended to do. There were many odd answers, saying that value of stories was not important during the maintenance phase, since all bug fixes have the same value. This is clearly false. Some bugs cost us much more than others do, and we definitely need to prioritise our effort in the maintenance phase. Others said that there was less urgency during the maintenance phase than during development, which is also not true in general.

A significant number of students got distracted by my example of limiting the number of cards in the “To Verify” column, if the number of testers available is limited. They focussed their whole answer on the problem of having just two testers, with many students making the odd claim that because issues arise more slowly during maintenance there is more time for testing! But this was just an example. The number of cards could have been limited in any of the columns after Backlog in this approach.

Q3.

This was the second most popular question in section B. It had the lowest average mark of all the questions, equally with question 2.

a) This was intended to be an easy starter question, and many students gave 100% correct answers. Unfortunately, a sizeable minority of students lost marks through not reading the question carefully enough. The question asked whether the statement “described” a software value (of the kind we had discussed for use in user stories), but these students interpreted the question as asking whether the statement “had value” (in a very general sense).

To take a specific example from the question, fixing bugs in a piece of software is a useful activity, but it does not describe the kind of value we expect to arise from developing software in an agile context. As we discussed in the lectures, to get value from software, its delivery must result in a beneficial behaviour change - usually saving someone time, or reducing costs, or reducing errors, etc. In this question, I was looking to see whether students could identify descriptions of behaviour changes from descriptions of other aspects of software development.

This is a good example of how revising from slides without actually attending the lectures is a risky strategy. In the lectures, we did a number of coaching games that brought out the specific meaning of “value” for software in an agile context, and specifically the “behaviour change” aspect. Reading over the slides (and listening to the podcast) without actually having done the associated games leaves open the risk that the importance of a term like “value”, and the specific interpretation put on it in the course unit, may not be grasped.

Beyond this, a number of students were skeptical as to the ability of any software program to increase the number of students who came to study at a specific university. A surprising number of candidates also got very worked up about the idea of whether it was possible to make cheating “impossible”, and who exactly would benefit from such a thing if it were. (I was a bit surprised at the number of candidates who implied in their answers that making cheating impossible would *not* benefit students…)

Finally, a small number of candidates lost marks because their answers did not make clear whether they thought the statement described value or not!

b) There were some very elegant answers to this question, but many candidates failed to give enough details to allow me to judge whether their planned MVP was feasible and able to deliver the stated value or not. For example, stating that the app would know when an item was returned to the fridge without saying how the fridge would know that it was the same item that was taken out previously.

Probably the most common mistake after this was to describe an MVP that was not actually “minimal”. For example, the question listed 3 functions that the eventual fridge app might offer. To be minimal, the MVP should have tackled just one of these, but a number of candidates included two, or all three, of the functions in their design. Others included lots of additional features (like

adding the product to an online shopping basket) when their design already gave value by providing useful information at the point of use of the fridge. Another example of non-minimal design was to make the MVP handle all possible products, instead of focussing on a small subset of high value, easy to manage products.

One aspect of the question that was not dealt with well, but which I did not penalise students for, was the problem of multiple users accessing the fridge. Many of you seem to assume that the owner of the fridge is also the only person who puts items in and takes items out, and that that same person buys all the food. This was at variance with the user roles of the stories (often describing family members).

c) It was great to see some of the very elegant stories that candidates gave in their answers to this question. Unfortunately, not everyone managed to reach these heights. A disappointing number of students used “user” and “customer” as the role of their story and (as usual) many of the values given were simple restatements of the functionality using different words.

d) This question was intended to be somewhat challenging and to take candidates out of their comfort zone and beyond what was covered explicitly in the course. There were only a handful of really convincing answers, although the majority of candidates managed to pick up *some* marks at least.

Probably the most notable cause of lost marks was for those candidates who seemed not to understand what a “trial” of software would involve. A number of candidates mistook the word “trial” for “test”, and talked about how the software would be tested for correctness. (These answers described putting a product in the fridge after its use-by date and seeing whether the software would issue a warning, for example.) But many candidates said that the team should use paper prototyping on the software. Paper prototyping is a very useful early-stage technique, but it cannot be used over time with real users to gauge the usefulness and value of a prototype. As we discussed in the lecture on agile testing, paper prototyping has many strengths but one thing it is *not* good at is assessing non-functional properties of a design. The trials of this smart fridge software would be all about non-functional elements. How annoying is it to go through whatever process is needed every time you take something in and out of the fridge? How often are the different queries on the data used? Do all the users of the fridge use the app in the same way? Etc.

Paper prototyping is also not appropriate when you want to test use of a design over a period, as in this case. The trial should show (for example) whether there any change in the amount of food wasted by a user of the app, compared to when they are not using it. Aspects like this can only be assessed over a period of time (certainly longer than a day), and so paper prototyping is not suitable.

Q4.

This question was more popular than it has been in some years, and with good reason as it was a straightforward explanation of specification-by-example with only a few subtleties involved that many students navigated easily. The average mark was an impressive 85% (the highest of all the questions in the exam).

a) This part of the question was generally not answered well, however, with very lack-lustre answers in all but a few cases. Marks were commonly lost through simply not saying enough to

earn 4 marks, while in other cases a lot was said but not many concrete meaningful (and correct) points were made.

A worrying number of candidates made simplistic (and frankly wrong) claims about writing unit and acceptance tests meaning that code was “guaranteed” to be bug free! Many also claimed that the developers would find it easy to write a set of good unit tests (because they knew the code) while an independent test team would have to learn the code first and wouldn’t be able to write such good tests. No one mentioned the problem of psychological set in testing, where we find it difficult to write good tests for our own code whereas someone coming fresh to the code will try more unusual cases and really stretch the code by comparison. It’s not cheaper to do away with an independent test team if the resulting code is full of bugs because it has not been tested thoroughly. Only a handful of candidates mentioned tools to address this, like test coverage.

b) Almost everyone gave a great answer to this part of the question. The only causes of lost marks in terms of table design were in poor naming choices (making up terms like “piping standards” rather than using words from the user story), and abstracting away from the decision about what makes for an extreme temperature. Most candidates were able to give good test cases as well. Marks were lost here for not choosing boundary value cases, and for trying to show multiple aspects of the business rules in one test.

c) There were more answers for this question that lost marks than for part b) but still in general the standard of fixture code provided was excellent, with most candidates scoring full marks or losing only one mark.

The most common errors were failing to deal with the translation between the strings of the FitNesse table and the primitive types of production code (“5mm” is a string, and is returned correctly by FitNesse as such), and in surprising production code design choices. Many of you were very reluctant to use class hierarchies to describe the different insulation types, and there were a worrying number of switch statements. Only two candidates missed the point altogether and wrote the production code logic in the body of the fixture.

Q5.

This question was more popular this year than it has been in some years, though unfortunately the quality of the answers was also less good than has previously been the case. This did not prevent some students was giving first rate answers and this question had the second highest average in Section B. But other candidates lost marks in ways that suggested they did not quite grasp the principles of TDD as strongly as needed to do well in the question.

a) Some very good answers were given to this question, although a number of candidates lost marks through poor exam technique, by not mentioning the Cohn Test Automation Pyramid at all in their answer. There was quite a lot of waffly chat about the value of tests, rather than specific points that would really help a team of experienced coders trying to get to grips with these techniques. An example of a weaker answer was to say that we need a lot of unit tests because they are quick to run. An experienced team will know this; they want to know what benefits a lot of unit tests would bring, to offset the undeniable costs required to create them. Other candidates decided (somehow) that the team must be using TDD. This was not specified in the question, but it seems unlikely that a team struggling to understand why lots of unit tests are of value are doing TDD! And, as usual, a number of candidates merely stated that lots of unit tests were a good idea, rather than attempting to explain why that should be so.

b) (i) There were lots of good answers to this question, with the main cause of lost marks being a failure to include specific values in the tests. A sentence that chats about what software should do ("player should gain 5HP when on yellow tile") is not a *test*. It is a requirement. A test has to have sufficient detail that it can be run repeatedly by different people and exactly the same test will have been run. This means stating the starting HP of the player, the coordinates of their location and of the tiles, etc, etc. Only a few candidates included tests about the pre-existing functionality of the game (being able to place tiles, to work out what kind of tile you were on, etc).

(ii) Most answers to this question were reasonable. A few people did not actually write any code for this question part. In those cases, I marked the code from the first step of part (iii) as if it were the answer to this part, wherever that resulted in a higher mark for the candidate.

The most common mistake here was to forget that the test must give a red result before we can consider this stage done. That means we must be able to run the test, which means the test code must compile. This in turn means that the stub code for the production classes mentioned in the test must be created for the test step.

Another common mistake was the write a test that would not fail (because it depended on functionality that the question described as already existing). An example would be to write a test that checks that stepping onto a yellow tile and then stepping off before a game tick had happened results in no change to the HP. Assuming that the yellow tile stub class is a subclass of a generic Tile class, then this test would run even before any production code had been written.

The correct answer was to write a test that involved *some* healing, in a simple and direct way (since healing is what this increment of functionality is all about).

(iii) Some candidates struggled with this part of the question, with the most common problems being:

- Odd production code designs, that avoided using a hierarchy to represent tile types and instead used lots of string based fields, or that placed responsibility for aspects of the behaviour on the wrong class (like having the Player class manage game ticks…???)
- Refactoring steps that involved making major behaviour changes to the code, rather than sticking to code quality improvement or simple generalisations justified by clear patterns in the code (which cannot emerge over the course of just 2 TDD cycles).