University of Glasgow | School of Computing Science

Level 3 Project Case Study Dissertation

ESE1 Team Project Dissertation

Agnes Ola
Ross James Gardiner
Lorenzo Roccato
Duncan Lowther
Nawaf Al Lawati

6 April 2020

Abstract

Education Use Consent

We hereby give our permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format.

1 Marking sheme - remove before submission

Reflecting on your practice is the hardest part of writing the dissertation, so you are encouraged to talk to the course coordinators and demonstrators to find out what you could include in this section. A good source of examples of incidents for reflection is often the documentation from your retrospectives, because you used the retrospectives to identify areas of your process that could be changed or done better. You should also, try to relate your experiences to other studies available in the software engineering literature (the recommended reading is a good starting point for this).

For example, if you found that you had to drop a feature during an iteration, discuss the reasons why the feature had to be dropped. Had you given yourselves too much work? Was the feature harder to implement than you realised? Had you got your priorities wrong? Then consider looking at the literature (see the recommended reading for PSD3) on project planning and estimation. Was your experience typical of a software project? What steps do other developers advocate for improving estimation?

Alternatively, did you have to make some big design decisions or choice of software platforms early on in the project? What impact did these choices have? Were they the right ones? How might you have improved the decision making process to reduce uncertainty? Did you implement a prototype before proceeding to far with the main implementation? How much effort did this involve? What did you learn about the platform as a result?

The dissertation should be a single PDF document of a maximum of 12 pages, not including front matter and references. You must use the LaTeX template provided for the dissertation and include all the requested meta-data. The LaTeX source for the template, your dissertation and any associated figures should be stored in your version control repository in a clearly indicated directory or branch. It must be possible to build the dissertation PDF from this source, using an automated build script, such as the ant build script provided.

- Presentation/structure - Is the dissertation complete, well organised, clear and literate? Are there examples of spelling mistakes or poor grammar in the dissertation? Is there a clear logical argument and structure to the narrative? Are references used effectively? Are the references complete?
- Description of objectives and achievements. - Are the problem domain, scope and objectives of the project clear? Is it clear what was achieved during the project and what the key technical challenges were?
- Reflection - Are there a number of experiences/critical events discussed in the dissertation? Does each experience answer the following questions: What was the circumstances of the event? What was learned as a result of the experience? What changed in the project team (if anything) as a result of the experience? Is

the experience related to other case studies available in the software engineering literature?

2 Introduction

Software engineering

This paper presents a case study of...

The rest of the case study is structured as follows. Section **??** presents the background of the case study discussed, describing the customer and project context, aims and objectives and project state at the time of writing. Sections **??** through Section **??** discuss issues that arose during the project...

3 Case Study Background

Include details of

3.1 Our client

3.2 Our project

3.3 Our software

4 Reflections

4.1 Windowbuilder themed part, not sure about name yet

Incidents and events: - Moving things about in the GUI disaasociated comments from corresponding code - Made merge impossible - Propagation of early, "temporary" design decisions - Difficult to rename variables - Look into why diff was so confused - Mention abandoned branch merges - Could only complete changes on the last day afer everyone else was done - Perhaps something on deadlocks waiting for others to finish

What we learned: - Make more files - Balance workload - Push earlier releases to see all the cracks in the seams - Acknowledge that there is no such thing as temporary code.

4.2 Agile Testing For Hardware Dependant Software

Kasper considers Agile methods for embedded system software development (Advances in Streamlining Software Delivery on the Web and its Relations to Embedded Systems). One point highlighted by this report is difficulty in establishing whether a bug is the result of faulty hardware or software. Additionally, Kasper describes hardware development life cycles as generally being much longer than software, often elapsing years between a singe iteration. The literature shows testing is one of the most challenging elements of a hardware dependant Agile software project. NextSteps is no exception.

One important characteristic of an Agile code-base is it's frequency to change many times even within a single day. For this development style, an equally rapid software testing procedure is required to verify changes to the code-base as it mutates. Rapid testing is a vital form of validation as it not only detects bugs in software quickly and effectively but also ensures developers continuous confidence in the ever changing code-base.

For continuous integration, the need for a rapid testing facility is addressed with the use of pipelines which verify builds as they are added to the codebase (reference Fowler). Fowler suggests software pipeline testing should be performed in a clone of the software production environment. For our product, the production environment includes the hardware itself and the world it interacts with.

This presents a problem for hardware dependant software such as NextSteps. Given that we are unsure of the technical details of the hardware and exactly how it physically interacts with the world, building a virtual production environment was thought to be a more substantial undertaking than the project work itself.

Our logic classes rely on a connection to hardware that is not accessible from the CI runner - which operates from a remote server. Similarly, the CI runner executes in a headless environment, so instantiating the main GUI application was also not possible. As a result of this, our testing pipeline could only compile the code and instantiate a minor class, `AAARunner.java`.

Our approach to testing may have been lacking in the rigour that is nominally required for rapid deployment, but we learned that our technique did have some merit. For example, compile-time errors pushed to the repository were always caught by the CI runner attempting to build the project. We also configured the CI runner with the target development version of Java. On a few occasions, errors as a result of prior Java versions on developer PCs were also highlighted by our solution.

However, this approach still left much to be desired. In fact, on the run up to the final product release, we noticed that the results produced by our application did not match that of the legacy program. Without the subsequent urgent patching, this would have been considered a critical software failure. Winter describes this as "testing crunch-time"(https://techbeacon.com/app-dev-testing/testers-guide-overcoming-painful-dependencies).

To do better, the team could've made an effort to emulate the hardware as a "Black Box" perhaps emulating some standard data from previous surface tests. This wouldn't account for the response of our product to unexpected data from the hardware, but would at least allow us to emulate the device at some level. Another approach could be to change the pipeline hosting machine to one that isn't head-

less. This could allow the GUI to be instantiated and perhaps tested using the Java `java.awt.Robot` library to simulate user interaction. However, many online resources advise against this(https://techbeacon.com/app-dev-testing/testers-guide-overcoming-painful-dependencies). Another option could be to locate the pipeline server on a machine connected to the hardware. This would be the most accurate way to include the hardware in the synthesised production environment. Finally, in hindsight, the team has learned that static code analysis has many benefits for CI testing and would've been completely feasible within our circumstances. Using this approach allows code to be checked for style compliance, static code metric evaluation and even the detection of some additional bugs. This is something we will take with us into future projects.

4.3 Issue3

5 Conclusions

Explain the wider lessons that you learned about software engineering, based on the specific issues discussed in previous sections. Reflect on the extent to which these lessons could be generalised to other types of software project. Relate the wider lessons to others reported in case studies in the software engineering literature.