Coursework

Team Coursework Exercise 3

Refactoring

COMP23311- Software Engineering I

University of Manchester

School of Computer Science

2018/2019

written by

Gareth Henshall

1 Overview

In this final exercise using the Stendhal code base, you and your team are presented with a single issue, recorded in your GitLab issue tracker. This time, however, the issue describes a larger code change that alters the Stendhal system's non-functional properties, while keeping its functional behaviour the same. That is, the issue describes a proposed modification to the code that **does not change the way the game should behave**.

This brings into play a number of new software engineering skills, in addition to those used so far in the course unit. The first difference is that the task is represented as a single GitLab issue, and this issue is too large for any single person to complete alone. **You are therefore required to develop a work breakdown that creates independent chunks of work that can be allocated to smaller groups of team members.**

As in Exercise 2, the scale of the requested changes is large, and likely represents more work than you can sensibly achieve in the time given for the coursework. The refactoring exercise must be completed in multiple stages, with each adding more complexity. For this coursework your team will therefore need to **determine a set of stages in the refactoring process** and **choose the degree of progression through these stages that they can commit to deliver by the deadline.** Some suggestion for the more accessible functionality is given in the GitLab issue.

Finally, it's worth highlighting that the refactoring task you have been given should simplify the process of adding new (related) functionality. A sample of such functionality is given in the GitLab issue, and you will be asked to add a similar piece of functionality in the interview. **You should therefore ensure that the changes you make are flexible enough to accommodate new, as well as existing, functionality**.

This exercise will follow the same broad phases as the previous ones, with some modifications:

- Step 1: a planning phase
- Step 2: a testing and implementation phase.
- Step 3: a code review phase.
- Step 4: an integration and system testing phase.
- Step 5: a release phase.
- Step 6: a preparation phase for the face-to-face interviews.

You can choose how you carry out these steps in your team, perhaps overlapping some of the steps, or carrying out the whole process for individual issues separately.

The exercise will test your ability to:

- use code reading techniques to locate the parts of a large software system implicated in the proposed code changes,
- use work breakdown structures to help you plan for larger scale developments that don't naturally divide into independent chunks of work,
- use basic software estimation techniques to commit to a workload that your team can deliver, and to divide work fairly across the team,
- identify a staged refactoring to carry out a controlled migration of functionality that preserves the functional behaviour with each step,
- make use of a simple Git workflow for coordinating the work of the whole team on a single issue, and to keep your release branch clean,
- write test suites that improve test coverage across new and existing code,
- use code review techniques to help ensure the quality of your team's code, and
- use a continuous integration server to ensure that only clean code reaches your team's release branch.

2 What You Have to Do

Step 1: High Level Planning

Your first task is to examine the new issue that has been raised in your team's repository. Draw up a work-breakdown structure (WBS) for the whole task, and use it to decide which of the current classes you will attempt to migrate to the new XML structure for this coursework exercise. Use the work breakdown structure to create defensible estimates for how much effort will be required to complete the task, and check this against the time the contributing team members intend to spend on the work. If the estimates are much higher than the likely available effort, then you will need to cut down the scope of the work, while still making sure you migrate a playable subset of the overall issue functionality.

Bearing in mind the time you have available in which to complete the coursework (the team study sessions and your personal study time for 2 weeks) and the skills you have in your team, you should use these estimates to decide:

- which of the classes you are going to migrate to the new structure (and in which stage of development),

- what additional tests will need to be added to provide adequate code coverage for the areas implicated in the migration,
- how the proposed code modifications should be broken down to divide between team members,
- what new Git issues will need to be created,
- which team member will be the lead for each issue,
- which team members will work on the implementation of each issue, and
- which team members will act as code reviewers for each issue.

All these decisions should be documented in your issue tracker and GitLab project wiki.

Once you are satisfied with your work breakdown structure, you should create issues in your issue tracker describing the components of the work that are to be completed by the deadline. Although issue trackers were traditionally used by end-users, for reporting problems with software, it is also now common for software teams to use issue trackers to plan and track their progress more generally. In this context, an issue can be raised by developers themselves whenever a discrete piece of work must be done in connection with the code base, and not just when a problem is found with the code created so far. New issues you create should be sufficiently clear that an outsider reading them would understand what work is expected to be done as part of resolving the issue. The issues you create should be associated with the Coursework 3 milestone. **At least one issue you create should related to test coverage.**

Lead team members should be assigned as the responsible person for the issue in the issue tracker. This person is responsible for the detailed planning for their issue, for scoping it down, for coordinating the implementation work amongst the sub-group assigned to it, and for documenting the work as needed for marking.

Sub-team members for each issue must be recorded on the issue tracker, using a sentence of the form:

> Sub-team member: <link to GitLab profile of team member>

GitLab profile links have the form: `https://gitlab.cs.man.ac.uk/name`, where `name` is the string used to identify a team member in GitLab. Multiple sub-team members may be recorded in separate comments, or in a single comment with newlines between the sentences. You do not need to record the team lead as a sub-team member.

Code reviewers for issues must also be recorded on the issue tracker, using a sentence of the form:

> Code review by: <link to GitLab profile of team member>

Multiple code reviewers may be recorded for a given issue, if wished, but code should be reviewed by someone outside the sub-team that created the code.

You should document your estimate for the **effort** required to implement each feature in the issue tracker. For this exercise, we are going to use GitLab's Time Tracking facility. Information about how to do this can be found at:

`https://gitlab.cs.man.ac.uk/help/workflow/time_tracking.md`

You should document the WBSs you create for each issue at this stage on your team's GitLab wiki. To assist us in marking your work efficiently, please use the page slug "client-slash-actions" for the top-level page you create to describe your planning. This will allow us to access the page from a link of the form:

> `https://gitlab.cs.man.ac.uk/comp23311_2017/stendhal_S1Team<your_team>/wikis/client-slash-actions`

Create a page for each of the issues you open when doing this task. You should ensure that all wiki pages you create for this coursework are linked to from the top-level "client-slash-actions" page. Wiki pages for each issue should include the basic WBS you create in order to estimate the difficulty of the issue.

Step 2: Testing and Implementation

Once features have been allocated a team lead, sub-teams can start work on the implementation of their issues. The team lead should begin by creating a detailed WBS for the work on the issue's wiki page, and by allocating parts of the work to the sub-team members. Care should be taken to find a breakdown of the work that allows sub-team members to work in parallel on different parts of the feature.

The team lead should set a due date for the delivery of the feature to the team, using the issue tracker's Due Date facility.

The team lead should update the estimate for the issue in the issue tracker if, based on the more detailed planning carried out, the sub-team feel the original estimate was inaccurate.

We ask you to use the same Git workflow as you used in the first two exercises, to allow your sub-team to work collaboratively on each issue without interfering with the work of the rest of your team. As in the second exercise, we are also asking you to control the size and risk of the work, by scoping which files will be migrated (and in what order), and to code in a test-first manner. We are also asking you to manage the quality of the commits your team makes, and to use a test coverage tool to assess whether you have written sufficient tests or not. Details of all these are given below.

Choosing a Sensible Starting Commit

It is important that all the team start their feature branches from a clean commit. Otherwise, problems in earlier commits (such as failing tests) will be carried forward into the new feature branches, and may result in lost marks for exercise 3.

Teams with a clean build for their development branch in Jenkins should be okay to start the work for the next coursework exercise from this point. Teams with unstable or failed builds on their development branch will need to start by getting the `master` branch to a clean state. If the problems were caused by merging feature branches with unstable or failed merges into `master`, then you will need to revert those merge commits (or the actual commits, in the case of a fast-forward merge).

Once a suitable starting commit has been identified or created, one member of your team should create a tag pointing to that commit. The tag should be called:

```
COMP23311/EX3/starting_point
```

We have created a Jenkins build for this tag, to help you choose a good starting point for your work for this exercise. Push the tag to your candidate starting commit and then manually request a build. (You need to request manual builds whenever you push tags or branches without code changes. Jenkins won't trigger a rebuild unless there are new commits to fetch and compile.)

If you place the tag at the wrong commit, and need to change it, you will need to delete it and recreate it. Instructions for deleting tags can be found at the end of this document.

It is expected that your development branch will start at this commit for exercise 3. All feature branches for exercise 3 should begin at this tagged commit or a later commit.

Git Workflow

As before, you should use feature branches to keep the work on each feature separate from the main development branch, until you are satisfied that the code is ready to be integrated into the main game code. Please continue to follow the Stendhal team's practice of using the `master` branch as the development branch.

In this exercise, the branch names are not given for you – you have to choose your own branch names to suit the task in hand. For consistency, you should ensure all your branch names are prefixed with "COMP23311/EX3". For example,

```
COMP23311/EX3/slashaction_test_coverage
```

would be an acceptable branch name for a branch that focused on test coverage.

Best Practice Commit Messages

For this exercise, you are again asked to follow best practice in terms of the commit messages you use. A reminder of the required format can be found in the Blackboard documents for the second team coursework.

Please note that all team members must use their own GitLab account to commit and push their work, and that commits must have your University e-mail address as the e-mail of the commit author. This is set using `git config`, as described on the School wiki:

`https://wiki.cs.manchester.ac.uk/index.php/Gitlab/Git_Setup`

Please make sure you set the `user.name` and `user.email` variables on all machines you intend to commit from.

Tests and Test Coverage

As in exercise 1 and 2, you are asked to write tests to describe the behaviour of the code change you are making. Each sub-team will need to develop tests that thoroughly test the current (client-side) functionality of the actions to be modified. These tests should represent all valid variants of slash action use.

When writing tests for more extensive functionality changes, you may wonder when you have written enough tests and when you can stop. This is a critical question for software developers, since writing tests is not free (neither in terms of developer time, nor the time required to run the tests). For this coursework, we ask you to use test coverage to help you

manage your tests, and determine whether you need to write more.

Now that you have had some prior experience with the test coverage tool, we are expecting you to increase coverage for the area of code you are working on. For this task you will therefore need to use the coverage reports to find the current instruction coverage for the areas you expect to be affected by the changes you will make when working on this issue.

For this exercise you are asked to achieve an instruction code coverage of 60% or higher for the following parts of the code prior to the release:

- All classes within `games.stendhal.client.actions`.
- All new classes/packages you create.

You should report the initial coverage figure for `games.stendhal.client.actions` on the Wiki page for the test coverage issue(s) you have created. Likewise, you should report relevant test coverage figures once the release is finalised on the wiki page, too.

Documenting Your Work

You should use the issue tracker and/or the team wiki to document any problems you encounter while implementing that require you to make changes to your plan or your team structure. Discussion while implementing using other tools (such as Facebook or Slack) is obviously fine but remember that we can only mark what we can see. Key decisions regarding who does what work, and changes to the planned scope for features that are running late, must be documented on your GitLab repository if we are to be able to take them into account when marking.

An important piece of documentation to keep is the record of how long you have spent on the feature so far. You should record this in your team's issue tracker on GitLab, using the "time tracking" feature. Information on how to use this is available from the side bar of each issue. Note that you will need to record all the time spent by all sub-team members. You will probably find it easier to update the time spent as you go along, rather than trying to remember and adding the total in at the end.

Step 3: Code Review

For this exercise, we are asking you again use code reviews to improve code quality. Your goal is to ensure that no code is merged into the development branch without having an independent team member look over it and check for errors or code quality problems. You will lose marks if unreviewed code is merged into the development branch.

Both test code and production code changes must be reviewed. You can choose whether to review individual commits as you proceed, or to review the whole code for a branch before merging.

Reviews must be given using the GitLab commit comment feature, or through merge requests. Reviews given verbally, or recorded in some tool other than GitLab, will receive no marks (for obvious reasons).

Every contributing team member must perform at least one review in GitLab across the exercise.

A guide to performing code review can be found under `Course Content→Week 7` on Blackboard.

Step 4: Integrate Completed Features into the Development Branch

When you are satisfied that a feature branch contains code that is fit to be integrated into the game, you should merge the feature branch into the development branch. For this exercise, **we ask that you use non-fast-forward merges for all merges of your feature branch into the development branch**.

Once your feature branch has been merged into the development branch, and the resulting commit produces a clean stable build, you can consider the issue to be closed and should update its status in the issue tracker. Don't forget to add the time required to carry out the merge to the issue's time tracker.

In-complete issues should be left open, even if some commits have been made for them. The team needs to know that this feature was not completed, so they can return to it in a future release should customer interest in the functionality proposed make the effort worthwhile.

Step 5: Prepare the Release

The final technical step is to prepare the release. Although several team members may contribute commits for this process, a single team member should take responsibility for making sure the release is created correctly. This team member should create an issue for this task called:

```
Prepare release 1.30uom
```

This issue should be associated with the coursework 3 milestone, and assigned to whichever

team member is taking responsibility for carrying out the release task.

Choose the commit on the development branch which will form the basis for the release. This commit should include all the changes for the features that have been completed by the team during the coursework and have been merged successfully with the development branch. These are the issues that will be included in the release.

Once again, you will need to update:

- The version number of the software is updated (to 1.30uom).
- The `doc/AUTHORS.txt` file (if any additional authors have contributed).
- The description of the changes included in the release in the `doc/CHANGES.txt` file.

You can make these changes directly on the development branch (or you can use a feature branch and merge with the development branch when complete, if you wish).

When you have created a commit that contains all the code and documentation you want to release, you should mark this by adding a *tag* at that commit. The tag **must** be called:

```
VERSION_01_RELEASE_30_UOM
```

This is the version of the code that we will consider to be your released code, when we are marking. So, it is important that you place it at the right place. You can create the tag locally and push it, or you can use the GitLab web interface to create the tag once the release commits have been pushed.

Once you have completed a clean release, you can close the release preparation issue. If you are not able to create a clean release by the deadline, you should leave the release preparation issue open.

Step 6: Prepare for the Marking Interview

You are done with the technical work at this stage, but there is one more task to do: prepare your team for the marking interview in week 12. In this interview, your TA will ask you to demonstrate changes made in the released code, show that functionality of modified code remains unchanged, and will discuss with you how you have organised your work to balance load across the team and what monitoring steps you've taken to keep the work of the team on track for the deadline.

During the interview, your TA will also provide server-side code for a (fairly simple) new action, and you will be expected to modify your client files during the marking session to add this new action by adding entries to your XML configuration file.

To prove to yourselves that you will easily add a new action, you may find it helpful to add a simple example action prior to the marking session. A server-side class will be provided on Blackboard in Week 10 for this purpose.

More information about the timing, location and format of this interview can be found on Blackboard, under `Assessment→Team Coursework→Exercise 3`.

Note that all team members must attend the marking interview, and any team member may be asked to demonstrate and talk about the issue they were responsible for. **Any team member who fails to attend and who has not registered a legitimate excuse with SSO or a member of the course team will receive an automatic 50% penalty on their mark for the exercise.** Team members who attend the interview will be unaffected by this penalty.

3 Submission of Your Team's Work

All submission of work for this coursework exercise is through your team's GitLab repository. All you have to do is make sure the contents of your issue tracker, wiki and Git repository are ready to be marked by the deadline. There is nothing else you have to submit.

Once the deadline is passed, an automated process will remove your developer access rights to your team repository, and replace them with reporter access. This will allow you to see your code, in order to prepare for the interview, but you will not be able to push further code changes or references to your Git repository after this point.

4 Coursework Extensions

Since this coursework is a team exercise, no extensions will be given. Team members who experience substantial difficulties in completing their work due to illness or other legitimate reasons will need to complete a Mitigating Circumstances form so that this can be taken into account later. The marking process is sufficiently flexible to take into account non-contributing team members without significantly affecting the team mark.

If you are not going to be able to carry out the work for your issue by the deadline set for your team, you *must* inform the other team members in plenty of time. This will allow them

to make decisions about what to include in the release, so they don't lose time dealing with the fact that your work has not been done.

5 Non-Contributing Team Members

Every team member is expected to contribute some meaningful code to the team's repository. Commits on feature branches should be made by the members of the sub-team recorded as responsible for the feature in the issue tracker.

> Any student who has not made at least one meaningful commit to their team repository, pushed from their School GitLab account, during the period covered by the exercise, will automatically receive a mark of 0 for the whole exercise.
>
> This applies even if you decide to work in pairs on the issues. Sitting and watching someone else make a commit, even if you are telling them what to type, does not count as a commit from you. The commit must be made from your own account and pushed to the repository with your GitLab credentials.

A meaningful commit is one that contributes code changes to either test or production code that moves the team's repository closer to the fix for an issue in some way. Adding white space, rewording comments or copying lines from elsewhere are all examples of code changes that will not be considered to represent a meaningful contribution to the exercise.

6 Partially Contributing Team Members

If a team member contributes something, but does much less than others or contributes their work in a way that causes problems for the rest of the team, the team as a whole can choose to reduce the mark of that student. For this to happen, you must:

- Send an e-mails to the student as soon as the problem is noticed, pointing out the difficulties they are causing for the team, and asking them to say what they can do to resolve matters. CC this e-mail to Sarah, so we have a formal record of the communication.
- Set a deadline for the team's work that is sufficiently far ahead of the actual deadline, so you have time to chase people who don't deliver.

- Before the team interview, send an e-mail to Sarah *and* the offending team member letting them know that the team will propose a reduced mark for them at the interview.
- At the interview, raise the issue with the TA, who will document the circumstances on your marking form, along with details of the proposed mark reduction. If the affected team member agrees, the proposed reduction will be applied at that point.
- If team agreement on the mark reduction cannot be reached, the whole team will need to meet with Sarah/Suzanne to agree a way forward.

Note that this process is not necessary for team members who have not made any commits in your team repository, as they will automatically receive a mark of 0 in this case.

Mark reductions apply to individual team members only. There is no effect on the mark of the rest of the team. Teams are asked to try to resolve problems within the team if possible, before making mark reductions, but this option is there as a measure of last resort for those teams who need it.

7 Plagiarism

This coursework is subject to the University's policies on plagiarism. Details can be found on the School web pages at:

`http://studentnet.cs.manchester.ac.uk/assessment/plagiarism.php?view=ug`

Note that committing the work of other people from your GitLab account counts as plagiarism, and action will be taken when it is detected. Rebasing commits authored by others does not count as plagiarism, providing the original authorship information is retained in the commit.

8 Trouble-Shooting

Finally, a reminder that general trouble-shooting information for working with the Stendhal code base, and tools such as Git and Eclipse is available on the School's wiki, at:

`https://wiki.cs.manchester.ac.uk/index.php/LabHelp:Main_Page`

The best way into these pages is either through the `Error Messages` index or the `Symptoms` index:

https://wiki.cs.manchester.ac.uk/index.php/LabHelp:Errors https://wiki.cs.manchester.ac.uk/index.php/LabHelp:Symptoms