A Report Into the Improvement of a Tabletop Gaming Web Application

Exam No. B024703

February 10, 2017

1 Introduction

This report describes a plan of action to improve a web application designed to assist the playing of a table top game. The code was received in a poor state, featuring errors in code that prevent it functioning properly, situations in which poor design decisions were taken by the original author, and situations where the specifications, as provided by the customer, have not yet been fully met.

The web application is written in Python and uses the Flask web framework.

In this report an approach will be taken that prioritises and improves upon failings in the current code in an agile, iterative fashion, using aspects of the Scrum methodology. The aim is not simply to provide code that is more functional, but that is also extendible and maintainable for further work. Furthermore, as development time will be limited on this project, work will be scheduled so there will be frequent releases of shippable code, as opposed to the waterfall approach which might leave no publishable code when work stops.

The Scrum approach entails dividing the project into broad stories. These stories then have particular tasks associated with them. Work is undertaken in Sprints, short bursts of work in which prioritised tasks are completed from all or some of the broader stories. Sprint planning occurs before each individual sprint, allowing for a structured yet flexible approach, with new

issues to be dealt with as they arise in a sensible way. At the end of each sprint there should be some usable code, i.e. working code that has some way improved upon what was there before.

This report first describes at a high level problems with the current code and divides these into stories, before describing in more detail the specific issues or tasks related to each story. An outline of initial sprint plans showing their order and priority will then be presented, followed by a Risk analysis and management strategy, before a brief conclusion.

It should be noted that this report, other than fixing basic bugs, does not detail plans for improvement of the UI. Such plans will be delivered in a separate report.

2 Problems and Stories

2.1 Developer Tooling and Documentation

At present there is no README file included with the source code describing how to run and do work to improve the code. The code is dependent on Python being installed on the host machine, as well as the Flask framework.

There is currently no source version control. This means that it would be hard to track changes during the development process. This would cause particular issues if there were more than one developer working on the code at once. In this case it would be essential that there was tooling in place that allowed many developers to work simultaneously on different issues in the code, tracking their changes individually and merging at the end of a task.

As well as tooling to track software changes, there should be tooling in place to track tasks being worked on within sprints. This tool should track which developers are working on which tasks, again preparing this project to be worked on by more than one developer.

In its current state there are no unit tests or continuous integration system in place for the project. Continuous integration would allow the code to be tested every time it is committed to the version control system. Unit tests are essential for maintainability, and for all future work tests should be written

first following the 'Test-Driven Development' (TDD) [1] approach. These tests should be combined with a build system that ensures only thoroughly tested code gets presented as complete.

2.2 Code Formatting and Abstraction

Almost all code for the project is, at present, contained within one source file. In this file are very long functions which each handle explicitly data storage, interfacing with the web framework and implementation game rules. All of these aspects should be decoupled from each other and broken into separate source files. This decoupling would make it much easier implement the unit tests described in 2.1 as well as undertake the further work described in the following stories.

In addition to this, the code currently contains very long lines and other poor styling features. Work should be done to ensure basic formatting is done correctly. Some sort of automated linting could also be included as part of the developer tooling, to be done at every build.

2.3 Missing or Inaccurate Features from Game Specification

The specification is long and detailed and contains many features not included in the current work. There are also features in the current work that deviate from the specification. The following is not intended to be a complete list of failings in this regard, as it is intended that part of ongoing work will be discovering more of these deviations. This will become more straightforward as the code quality improves and bugs are removed.

Various terminologies within the game are inaccurately represented. For example, squads are referred to as bands.

The specification also calls for access control, with users having their own squads that they have modification privileges too. Adding access control is a very sophisticated problem. This goes well beyond what can reasonably be achieved in this software project. Addressing this element of the specification may require more of educating the customer as to what is feasible than actual coding.

2.4 Data Store

All data that is persistently stored by the code is currently written to text files from within the functions that implement the game and interface with the framework. As described in 2.2, the data handling code must be abstracted away before any work can be done improving the implementation of the data storage.

At present the data store adds a new file for every new 'squad'. This file contains unformatted plain text information on the members of that squad. This could certainly be improved so that data was stored in a more coherent way that would be interpretable by a human, with or without the help of another program. It may be necessary to use a full database, or it may be decided that a simpler flat file solution is appropriate, with data stored in XML or JSON format. Deciding on this approach is a task in itself, and is discussed further in 3.4.1.

3 Tasks and Estimates

This section outlines specific tasks that must be completed along with estimates for how big those tasks are. Estimation is done through Fibonacci scoring, a popular technique in Agile development practice. Tasks are assigned a number from the Fibonacci sequence that shows their relative length. These numbers are referred to as 'story points'. During early sprints a more accurate picture can then be determined as to the rate at which these numbers are completed. This enables the calculation of a 'velocity' which informs how many tasks of what size to add to later sprints.

Each task is also assigned a code, to make it easy to find in the task tracking tool, and lists the other tasks that it is dependent on.

The section mirrors section 2 in structure, apart from including a new category, "Bugs". This works somewhat more informally than the full stories, as tasks will be added to it as bugs arise and will be prioritised outwith the usual sprint planning process.

The tasks listed here should not be considered a complete list. Indeed, part of the iterative process of development that the Agile/Scrum entails requires the constant discovery of new tasks. There is a very heavy bias in the tasks

presented here towards code quality, as opposed to compliance with the specification. While good code that does the wrong thing is useless, it is appropriate here to improve the quality of some very poor code before further work can be done. Otherwise work would become increasingly inefficient.

3.1 Developer Tooling and Documentation

3.1.1 Write README describing how to install dependencies

Code: tool1 Size: 3

Dependencies: None.

Write a README detailing dependencies. This should say that Python and PIP are system requirements, giving versions. It should further describe how to install Flask using PIP. The README should then detail how to run and test the program.

3.1.2 Upload Sources to GitHub

Code: tool2 Size: 1

Dependencies: None.

Git is an obvious choice for a source control tool. It is completely dominant and the author is familiar with it. A GitHub account already exists for code to be uploaded to.

3.1.3 Link GitHub to CI

Code: tool3 Size: 3

Dependencies: tool2.

Use Travis CI to run a build script every time source is pushed. At first this can just check that the code runs in Python without errors, later it should run a suite of unit tests. Even later it could also run integration/UI tests.

3.1.4 Enter Tasks in Task Tracking Tool

Code: tool4 Size: 2

Dependencies: None.

GitHub also has issue tracking built in. The tasks outlined in this section should all be entered into this.

3.1.5 Write a Suite of Unit Tests

Code: tool5 Size: 2

Dependencies: tool3, form1, form2.

Write a suite of tests and run these in the build script. Also include test coverage report if possible.

3.2 Code Formatting and Abstraction

3.2.1 Abstract All File Writes to Separate Program

Code: form1 Size: 5

Dependencies: None.

All code that handles persistent storage should be within its own source file. This should then be imported into the main source file.

3.2.2 Abstract All Game Rules to Separate Program

Code: form2 Size: 5

Dependencies: None.

All code that implements game rules should be within its own source file. This should then be imported into the main source file.

3.2.3 Wrap Long Lines in Source

Code: form3 Size: 1

Dependencies: None.

Make sure lines do not exceed a certain length in source. Make sure all lines are indented to the correct degree.

3.2.4 Add Linting To Build Script

Code: form4 Size: 2

Dependencies: tool3, tool2.

Find a suitable linting tool for Python and run it on all sources from within the build script.

3.2.5 Make Port etc. Depend on Config File

Code: form5 Size: 3

Dependencies: None.

Write a config file and use it to determine what port to serve to and other variables that may change. Configuration changes should not require edition of the source code.

3.3 Missing or Inaccurate Features from Game Specification

3.3.1 Correct Game Terminologies

Code: game1 Size: 1

Dependencies: none.

Rename:

- 'Wizard' to 'Captain'
- 'Band' to 'Squad'
- 'Apprentice' to 'Ensign'

to more accurately reflect specification.

3.3.2 Make Port Characters and Values Depend on Config File

Code: game2 Size: 3

Dependencies: None.

Write a config file and use it to determine what characters cost and their attributes. It should be possible to change these without editing source code.

3.4 Data Store

3.4.1 Research Most Appropriate Data Storage Solution

Code: data1 Size: 3

Dependencies: form1.

Determine what best option long term for data storage is. Possibilities include SQL database, JSON flat files, XML flat files.

3.5 Bugs

3.5.1 Home Page is Titled 'Holder Page'

Code: bug1 Size: 1

Dependencies: None.

Rename this to something sensible, such as 'Home Page'.

3.5.2 'Internal Server Error' When Adding New Squad

Code: bug2 Size: 3

Dependencies: form1.

Error when writing new file for new squad. This appears to happen when the form is incorrectly filled. Instead there should be an error message clearly explaining to the user what is wrong. For ease, development work on this should happen after data handling code has been abstracted out.

```
IOError: [Errno 21] Is a directory: u'~/project/bands/'
```

4 Work Plan

4.1 Spikes

Figure 1 shows the general structure of an Agile/Scrum project. It does not, however, include spikes. Spikes are a special class of task that can happen before or during a sprint. They involve doing the research required so that other tasks can be completed [2]. An example is *data1*. Research work is needed to determine the future direction of the project.

In this project another task that requires completion before any other work can be done is a spike getting the developer up to speed in both Python and the Flask web framework. This is a non-trivial task and one that all future development work depends on.

4.2 Sprints Planning

Sprint planning should occur before every sprint. While in commercial settings, sprints extend to weeks long with large teams, in this case there will

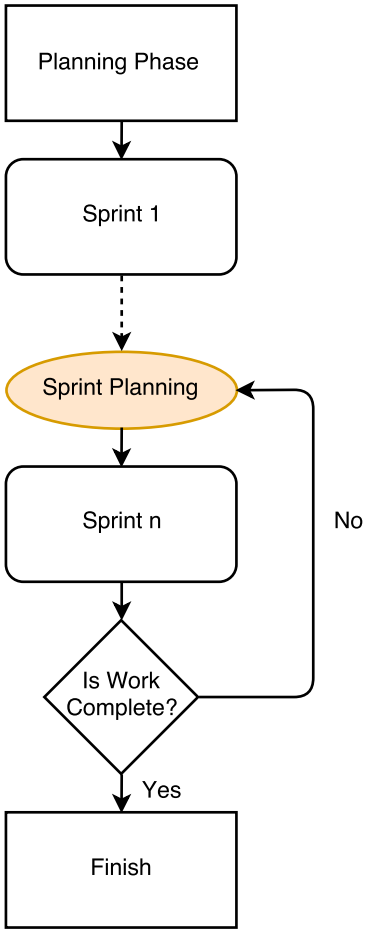

Figure 1: Flow Chart of Agile/Scrum Process

only be one developer working in short bursts. As such the target duration for a sprint will one week of part time work, totalling 8 hours. There will be three week long sprints associated with this project.

The dates for these are as follows:

- Sprint 1: 01/03/2017 to 08/03/2017
- Planning for Sprint 2: 09/03/2017
- Sprint 2: 10/03/2017 to 17/03/2017
- Planning for Sprint 3: 20/03/2017
- Sprint 3: 21/03/2017 to 28/03/2017

leaving time to write a report on the work undertaken, before work must be submitted on 31/03/2017.

This structure is in line with the flow chart in figure 1. Planning occurs in an iterative fashion, which means that more planning happens once the project is underway. This means planning is done when the developer is more experienced in the project and is more likely to make accurate estimates and correct prioritisations. The only sprint that need be planned in detail at this stage is Sprint 1.

4.3 Planning Sprint 1

As there will be much more work to do after Sprint 1, it seems wise to prioritise issues which will make further development easier. This includes tasks such as implementing source control, ensuring different functionalities of the code are properly abstracted away from the framework code.

It was decided but as there is only one developer, and each sprint contains a relatively small amount of work, it would not be appropriate to include a Gantt chart. Instead Table 1 shows, in time order, the tasks that must be completed in Sprint 1.

The first task completed will be *tool4* "Enter Tasks in Task Tracking Tool", as this means it will be easier to track the status of all of the following tasks.

time

Task Code	Story Points
tool4	2
tool1	3
tool2	1
form3	1
form1	5
total	12

Table 1: Table showing sprint 1 order of work.

Then *tool1*, writing a README will be completed. The sources will then be uploaded to GitHub, before basic source text formatting is done. The code will then be refactored into separate functions.

This sprint will leave the code in very good shape for further work to be done. There are 12 story points in total assigned to this sprint. This is 2 per day, with one day off. Progress made during this sprint will be evaluated to determine a more appropriate amount of story points to assign in future sprints.

5 Risk Assessment

This section presents a quantitative approach to risk management. Risk points are calculated by multiplying the expected delay from a risk by the probability of that risk being realised. Along with this, strategies of risk mitigation will be presented. The probabilities and delays presented are for the risks with the recommended mitigations.

5.1 Insufficient Programmer Expertise

Probability: 50% Delay on Schedule: 7 days.

The developer who will work on this currently has little experience with Python and the Flask framework. The spike to learn these skills must happen around other things that the developer is working on before the first sprint starts on 01/03/2017. Mitigating this means ensuring that the developer is aware of the necessity of learning these skills before this date, as well as

lightening the load in Sprint 1, when the developer will be least experienced.

If there is a significant delay there may not be time to complete the full 3 sprints. As working code is presented after each sprint, it would be perfectly possible, if not desirable, to present work as complete after an earlier sprint, if timing considerations made this necessary.

Risk score: 3.5.

5.2 Expectation Management

Probability: 30% Delay on Schedule: 4 days.

It is essential to ensure that the customer is aware of the poor state of the current code. This means that a lot of work needs to be done overcoming technical debt before they will notice any improvements. Mitigating this requires very clear communication with the customer. It also means ensuring earlier sprints involve less of the more noticeable changes to the customer. This prevents build up of further technical debt as well as tempering expectations as to the visible rate of change in the application. If not managed well the customer could expect more work, meaning more time is required.

Risk score: 2.

5.3 Developer Time Limitations

Probability: 15% Delay on Schedule: 7 days.

The developer in question has several other draws on their time. These could become severe at around the time this work is scheduled. However this risk has been mitigated by only scheduling 8 hours work a week from the developer.

Risk score: 1.

6 Conclusion

To conclude, a comprehensive plan of action has been presented. This plan provides significantly more detail as to how the early stages of development should be undertaken, with time allotted for planning later stages in an iterative fashion.

The technical work is heavily weighted towards improving code quality, as opposed to adding features or ensuring more compliance with the specification. This is because the detailed plans described here are targeted more at early stages of development. By resolving technical debts early on, the author hopes that more features may be added later than would otherwise be the case. It would also allow other developers to continue working on a more maintainable source base at a later date.

There could be some question as to whether a Scrum/Agile methodology is appropriate for a project this small. It was however selected for its flexibility, which mitigates some of the risks associated with a developer so inexperienced in Python development. Indeed, risks associated with developer inexperience were shown to be the most significant threat to the project in a quantitative risk analysis.

References

[1] The Agile Alliance *TDD* `https://www.agilealliance.org/glossary/tdd/` Accessed: 01-02-2017

[2] Scaled Agile *Spikes* `http://www.scaledagileframework.com/spikes/` Accessed: 01-02-2017