Designing a testing framework for the Riot Watcher project

Richard C. Smith, Daniel L. Feliciano, Brenard S. Casey
College of Charleston

Introduction

Software testing is a very important aspect of software engineering in order to produce quality, maintainable code and to diagnose any potential problems or issues that may arise due to the addition of new functions or features. By developing test cases and a framework to drive them, a software development team can be assured that the code they produce is operational and safe for all involved.

However, one of the problems that many software developers face is the sheer number of test cases that must be designed, written, and executed. These tests can end up in the hundreds or thousands, or perhaps even larger, as the software program grows in complexity and components are added. A solution to this is building an automated framework that takes the work out of executing the test cases by requiring developers to create the test cases, but allowing the system to handle the rest of the work.

The purpose of our project was to find an open source project to build an example automated testing framework for. This project and framework had to be designed and built to run on a Linux operating system, specifically Ubuntu.

Our team began the semester developing a framework for the open source project titled Pokemon-Go map, found on Github. We made great progress on identifying test cases and becoming familiar with the codebase for this open source project and were developing our framework at a steady pace. However, about one third into development, Niantic games blocked all access that third party applications had into their servers and databases. Essentially, what this meant, was that development for this project was ended, and therefore we were out of luck for continual development on our project.

Once we were faced with this obstacle, we began to identify alternative projects that we could work on. Since each member of our group was familiar with the Python programming language, we knew that we wanted to stick with a Python program to develop our test cases and framework for. We identified two potential solutions: A scrabble cheating application, or a Riot Games API Python Wrapper entitled the Riot Watcher. We discussed each project and began identifying potential test cases for both, but finally decided on the Python Wrapper due to the sheer number of methods and the familiarity each of us had with the source content of League of Legends (which the API's data dealt with).

After this, we began developing test cases and the test driver for this system.

The Riot Games API

The Riot Games API is an information collection service for the massively multiplayer online battle arena (MOBA) game, League of Legends.

It is a restful API that was designed to allow developers access to information on players, games, items, champions, and other information. It relies on a user to have an established account on the game (at least level 30) and has a limit on the amount of calls per minute that a user can accomplish. Users can retrieve different data by making simple calls. Once a user makes a call, the information is returned as a JSON string that has to be interpreted by the user in order to access.

Our chosen project is a Python wrapper that allows users to make calls to the API by calling Python methods rather than the regular API calls. It allows those familiar with Python to have an easier way to access the API.

Testing Framework

Our team developed our testing framework using the Python programming language. We first began our framework by creating a Test Case class/object, which included the information located in our test case template: the test ID, the requirement being tested, the component being tested, the method being tested, and the input. Additionally, our test case object was instantiated with an empty 'result' attribute, to collect and store whether a test passed or failed after the tests were run.

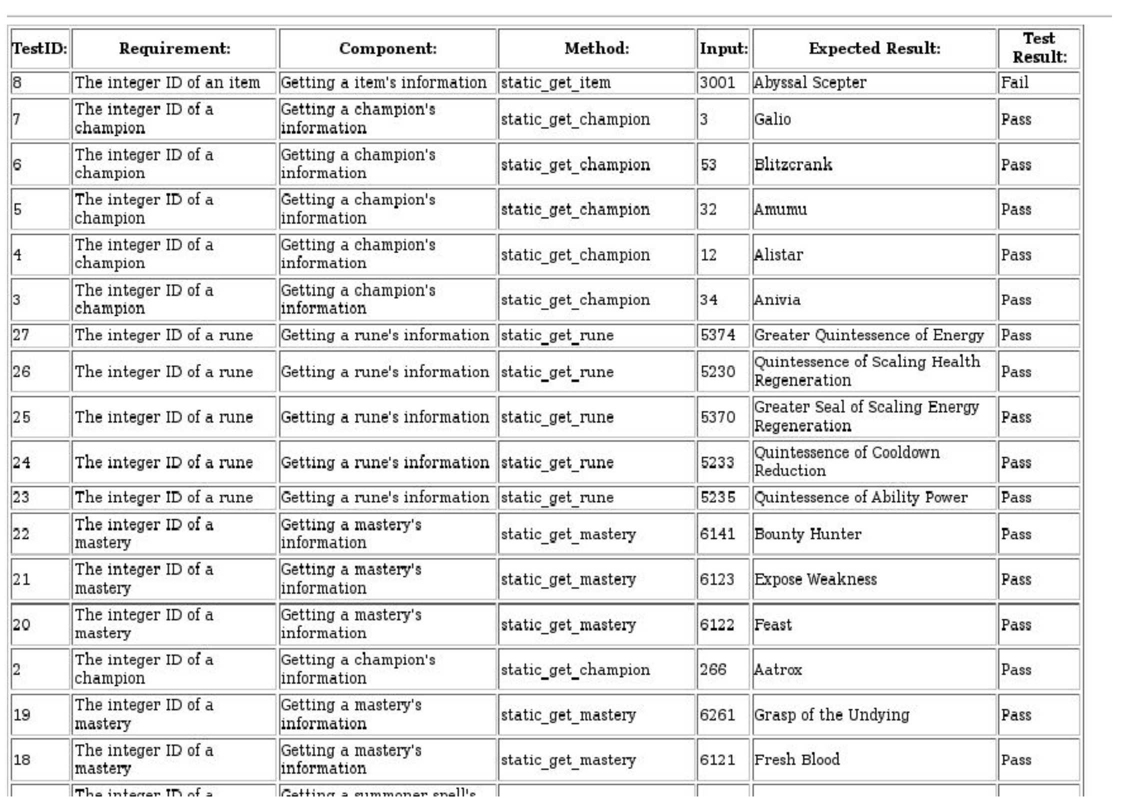
Each test case was defined and stored in a text file, and these text files were read incrementally and stored as python objects. After the reading of the test cases were complete, the methods that we were testing were called and the results collected and outputted to an HTML document, which was then opened in a web browser and displayed to the user.

The methods we chose to develop our twenty five test cases around were as follows:

Static_get_champion
Static_get_item
Static_get_rune
Static_get_mastery
Static_get_summoner_spell

These methods were chosen due to the familiarity each member of our team had with the data that they would return.

Our Test Results report



TestID:	Requirement:	Component:	Method:	Input:	Expected Result:	Test Result:
8	The integer ID of an item	Getting a item's information	static_get_item	3001	Abyssal Scepter	Fail
7	The integer ID of a champion	Getting a champion's information	static_get_champion	3	Galio	Pass
6	The integer ID of a champion	Getting a champion's information	static_get_champion	53	Blitzcrank	Pass
5	The integer ID of a champion	Getting a champion's information	static_get_champion	32	Amumu	Pass
4	The integer ID of a champion	Getting a champion's information	static_get_champion	12	Alistar	Pass
3	The integer ID of a champion	Getting a champion's information	static_get_champion	34	Anivia	Pass
27	The integer ID of a rune	Getting a rune's information	static_get_rune	5374	Greater Quintessence of Energy	Pass
26	The integer ID of a rune	Getting a rune's information	static_get_rune	5230	Quintessence of Scaling Health Regeneration	Pass
25	The integer ID of a rune	Getting a rune's information	static_get_rune	5370	Greater Seal of Scaling Energy Regeneration	Pass
24	The integer ID of a rune	Getting a rune's information	static_get_rune	5233	Quintessence of Cooldown Reduction	Pass
23	The integer ID of a rune	Getting a rune's information	static_get_rune	5235	Quintessence of Ability Power	Pass
22	The integer ID of a mastery	Getting a mastery's information	static_get_mastery	6141	Bounty Hunter	Pass
21	The integer ID of a mastery	Getting a mastery's information	static_get_mastery	6123	Expose Weakness	Pass
20	The integer ID of a mastery	Getting a mastery's information	static_get_mastery	6122	Feast	Pass
2	The integer ID of a champion	Getting a champion's information	static_get_champion	266	Aatrox	Pass
19	The integer ID of a mastery	Getting a mastery's information	static_get_mastery	6261	Grasp of the Undying	Pass
18	The integer ID of a mastery	Getting a mastery's information	static_get_mastery	6121	Fresh Blood	Pass
	The integer ID of a	Getting a summoner spell's				

Conclusions

Our team learned a lot throughout the course of the semester. While we encountered a few issues and setbacks, particularly with our first project being shut down, we overcame the obstacles and came out on top. Before the semester, none of the members of our team had any experience in designing test cases or testing frameworks, and by the end of the project we are comfortable standing behind the quality of our work and the framework we designed, which could easily be adapted to work with a different project by changing a few lines of code.

Our framework is especially important because our project did not have a framework or established test cases designed for it before we began. While the developer wrote some sample test cases, they were very basic and did not extend very deep or very wide into the project to show that the program is working. Because of the sheer number of users (69 forks, 40 stars) of the wrapper, we believe that it is very worthwhile and rewarding that we designed this framework, and intend to submit it to the developer of the wrapper so that he and other users, both present and future, can continue to use it.

Bibliography

- **The Riot Watcher project can be found at: https://github.com/pseudonym117/Riot-Watcher**
- **Information on the Riot Games API can be found at: https://developer.riotgames.com/**
- **Our team's github repository, containing our testing framework, can be found at: https://github.com/CSCI-362-02-2016/Syntax-Error**