Software Testing

Y1481702

April 2, 2018

Contents

1 Test Plan

1.1 Introduction

MASON is a software library for creating agent-based simulations in Java. The software fits into the category of shrinkwrap meaning it might be in use in a wide range of real-world production environments. The software is freely-available and open-source, which is a special case of shrinkwrap software. A common trait of open-source software is that tasks 'that are not considered "fun" often don't get done'.[1] For MASON in particular, it is likely that the software has not been thoroughly tested as there is no trace of any automated testing, either on the MASON website, or its GitHub repository.

1.1.1 Tools

The IntelliJ IDE was used to explore the code and develop automated tests. JUnit has been used to create automated unit tests for the software. UI tests?

Git version control has been used to manage the code for the JUnit tests. For a long term project, this would mean that automated tests could be updated alongside any code changes.

1.2 Test Coverage

1.2.1 System Overview

As stated in the project brief, the testing only needs to cover the `sim.engine`, `sim.field` and `sim.field.grid` packages, but not their subpackages.

- `sim.engine` is responsible for the core simulation management, including the agent scheduling.
- `sim.field` provides abstract classes for the representations of space in MASON simulation models, with subpackages managing specific instances of these.
- `sim.field.grid` provides various 2D and 3D grid representations of simulation space.

The expected behaviour of the software has been determined using the extensive Java MASON documentation[2], first-party example implementations as well as a number of open-source examples[3].
The system behaviour has been tested against the inferred requirements of the client, a private biological research institute. The client needs the software to...

There are a number of goals we could define for our testing, such as finding the maximum number of bugs or complying with regulator set demands. The given requirement here is to verify, with a limited number of resources, that the software is dependable. In general, it's better to test with the aim of showing a product fails, if we cannot do so then the product

is reliable enough[4, pp. 20]. As such, the main requirement for our testing is to find a significant number of undiscovered bugs.

In order to design appropriate test cases an understanding of all levels of the system was needed. As resources here are significantly limited (only 8 testcases are allowed) it will be necessary to ensure they are used in the most effective way.
It has been shown across software projects that some modules of code may be significantly more error prone than others[cite]. [5] demonstrated how various metrics can help predict which parts of the code that bugs may be hiding in.
IntelliJ's Diagram feature helped to show how different components of the system are connected. Code coverage has provided a good understanding of which parts of the [code] are regularly used while running the software. Other metrics, such as how often different source files have been changed provide an insight into which files are most subject to change.

Producing the right metrics to help test the application have been particularly difficult in this case. No real history of previous bug tracking GitHub commits are all credited to *eclab* rather than individual developers
Cyclomatic complexity can be a good measure of the most *dangerous* sections of applications.

1.2.2 Unit Testing

Unit Testing should NOT cover: -Trivial Code- getters, setters -Code that has non-deterministic results -Code that deals only with UI but SHOULD cover: -Core code that is accessed by (a lot of) other modules -Code that seems to gather a lot of bugs, how do we determine this? -Code that changes by multiple different developers, github commits may not tell whole story here? [6]
Pareto's law: 80% of the bugs, are found within 20% of the modules..! CITE
Aim for 60-70% code coverage of business logic, ˜20% of the overall application.

1.2.3 System Testing

1.2.4 Integration Testing

1.2.5 Acceptance Testing

Acceptance Testing could be performed to determine if the software can be operated effectively by our end users. Acceptance testing can be useful for understanding the domain of our software better, but as we are not intending to further develop MASON, this is not particularly relevant here. Acceptance Testing should be performed ON?/BY? domain experts?- I am NOT a domain expert..

While acceptance testing will not help us determine that the software is *dependable*, it would help to discover if it is appropriate for the target audience. In particular, our users have

supposedly only received a basic level of Java training. It has been stated that MASON is less-suited to beginner programmers, when compared to other tools, such as NetLogo[7].

1.2.6 Regression Testing

Unfortunately neither the website nor the GitHub repository for MASON provide any previously implemented automated testing. Either this testing has not been done, which is common with freely-distributed software, or it has not been publicly distributed. As such, it will not be possible to run any regression testing as part of the project.

2 Test Case Specifications

2.1 Test Case 1

2.2 Test Case 2

2.3 Test Case 3

2.4 Test Case 4

2.5 Test Case 5

2.6 Test Case 6

2.7 Test Case 7

2.8 Test Case 8

3 Test Results

3.1 Test Case 1

3.2 Test Case 2

3.3 Test Case 3

3.4 Test Case 4

3.5 Test Case 5

3.6 Test Case 6

3.7 Test Case 7

3.8 Test Case 8

4 Test Summary Report

The significant limitations of testing resources reduce the level of confidence with which we can say that the software is free from defects.

5 References

[1] J. Spolsky, "Five Worlds," May 2002. [Online]. Available: https://www.joelonsoftware.com/2002/05/06/five-worlds/ [Accessed: Mar. 15, 2018]

[2] S. Luke, *Multiagent Simulation And the MASON Library*, 19th ed., George Mason University, Jun 2015. [Online]. Available: https://cs.gmu.edu/%7Eeclab/projects/mason/manual.pdf [Accessed: Mar. 28, 2018]

[3] K. Alden, J. Timmis, P. Andrews, H. Veiga-Fernandes, and M. Coles, "Pairing experimentation and computational modeling to understand the role of tissue inducer cells in the development of lymphoid organs," *Frontiers in Immunology*, vol. 3, p. 172, 2012. [Online]. Available: https://www.frontiersin.org/article/10.3389/fimmu.2012.00172

[4] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach.* Wiley Computer Publishing, 2002.

[5] T. Zimmermann, N. Nagappan, and A. Zeller, *Predicting Bugs from History.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 6988.

[6] K. Kapelonis, "Don't Test Blindly: The Right Methods for Unit Testing Your Java Apps," Jan 2013. [Online]. Available: https://zeroturnaround.com/rebellabs/dont-test-blindly-the-right-methods-for-unit-testing-your-java-apps/ [Accessed: Mar. 29, 2018]

[7] S. F. Railsback, S. L. Lytinen, and S. K. Jackson, "Agent-based simulation platforms: Review and development recommendations," *SIMULATION*, vol. 82, no. 9, pp. 609–623, 2006.

6 Appendix

Package	Lines of Code
`sim.engine`	444
`sim.field`	2,961
`sim.field.grid`	10,248
Total	13,653
Average	4,551

(a) Lines of Code

Package	v(G)	
	Average	Total
`sim.engine`	2.43	374
`sim.field`	2.38	57
`sim.field.grid`	3.75	1,829
Total		2,260
Average	3.39	753.33

(b) Cyclomatic Complexity

Package	PDcy	PDpt
`sim.engine`	2	54
`sim.field`	1	24
`sim.field.grid`	2	20
Average	1.67	32.67

(c) Package Dependency

Figure 1: Code Metrics for the relevant MASON libraries

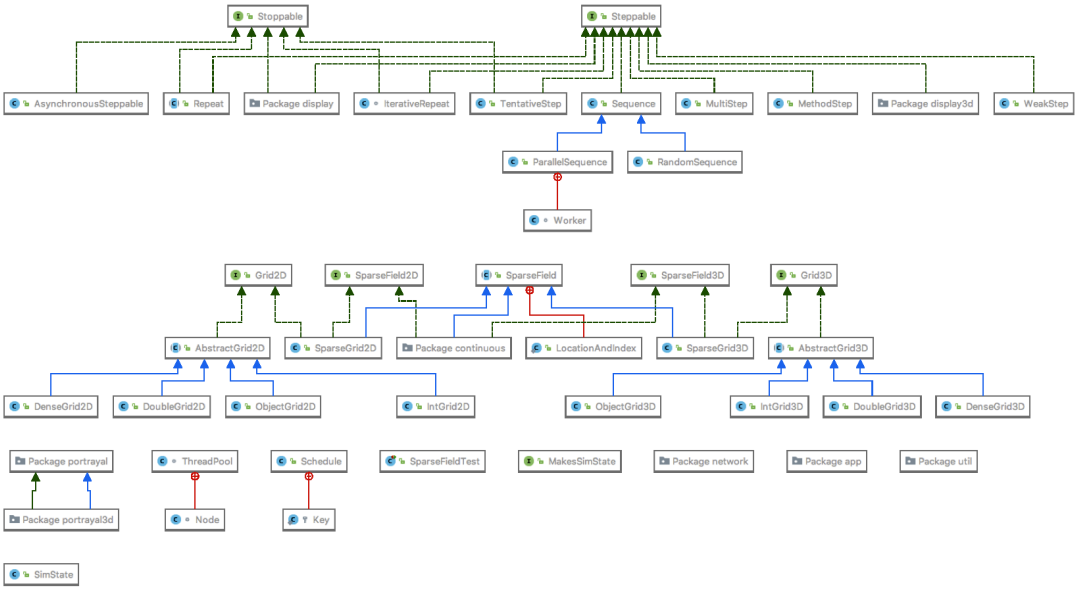

Figure 2: Reverse Engineered UML diagram of Class Hierarchy