Software Testing

Y1481702

April 4, 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. This IDE gives powerful features, including plugins for generating code metrics and displaying coverage of unit testing. JUnit has been used to create automated unit testing for the software. Git version control has been used to manage the code for these JUnit tests. For a long term project, this would be particularly advantageous as any automated tests could be updated and versioned alongside any future code changes.

1.2 Test Coverage

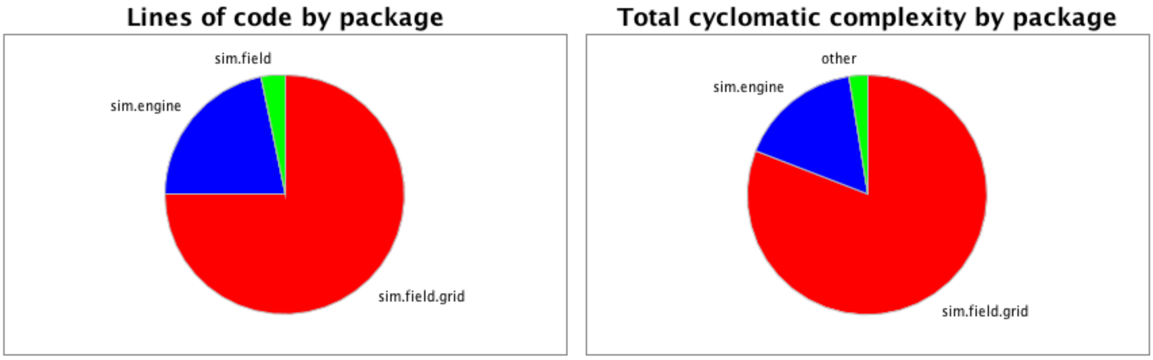
1.2.1 System Overview

In order to design appropriate test cases an understanding of all levels of the system was needed. As resources here are significantly limited, with only 8 testcases are allowed, it will be necessary to ensure they are used in the most effective way. As stated in the project brief, the testing only needs to cover the following 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.

Each of the Fig. 1a shows the proportional sizes of each of these three modules.

It has been shown across software projects that some modules of code may be significantly more error prone than others. The Pareto principle is said to hold with software bugs, with 80% of software bugs being found within 20% of the code[2, pp. 124]. We can use a variety of metrics about our software to predict which parts of the code that these bugs may be hiding in[3].



(a) Lines of Code (b) Cyclomatic Complexity

Figure 1: Charts showing various metrics of MASON

High cyclomatic complexity (Fig. 1b/Fig. 3c) can be a good indicator of the most bug-ridden sections of applications. The causality here can be easily understood: complexity creates confusion which results in developers misunderstanding the software they are trying to build. This misunderstanding can provoke a large number of bugs in the code. The graphs show that `sim.field.grid` has a slightly higher share of the code's complexity than lines of code, this should be investigated in further detail.

Fig. 3d shows how the packages in the test scope interact with other packages in the system. In particular, `sim.engine` is relied on by a large number of packages. Code coverage has provided a good understanding of which parts of the [code] are regularly used while running the software. How often different source files have changes is also a particularly useful metric. Files which are subject to frequent change have had more opportunity to develop additional bugs.

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

1.2.2 Testing Goals

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 any significant undiscovered bugs in commonly used code.

Due to limited resources, the tests will aim to cover the parts of the software which are likely to be used more often by the target users. MASON is bundled with some demo simulations that give example usages for the library. Running these simulations with code coverage detection turned on has given a good overview of which parts of the code are used regularly (Fig. 6). Core simulation code that is used routinely by all simulations regardless of their customisations should be tested the most rigourously. Both the documentation and code coverage checks indicate that this type of code can be found within `sim.engine`.

1.2.3 Expected Behaviour

The system behaviour has been tested against the inferred requirements of the client, a private biological research institute.
The expected behaviour of the software has been determined using the extensive MASON documentation[5], first-party example implementations as well as a number of open-source examples[6].

1.2.4 Unit Testing

Unit Testing allows the testing of small sections of source code. In this case, unit tests will help us to isolate the select packages that are within the scope of testing from the rest of the system.

Unit testing should focus on core code that is a dependency for other modules, code that regularly gathers bugs and code that is changed by a number of different developers. It should not cover trivial code, such as accessors and mutators, code with non-deterministic results or UI code.[7]
Mention Pareto principle again! Aim for 60-70% code coverage of business logic, ~20% of the overall application.

1.2.5 System Testing

1.2.6 Integration Testing

1.2.7 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[8].

1.2.8 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 Cases 1-3

As previously stated, our initial unit testing will focus on the simulation scheduler which provides the core functionality for MASON. This functionality is contained within the Schedule class as shown in Fig. 2. While this class initially appears to be quite large, at closer inspection, many of its methods are simply overloading others, due to Java's lack of real support for default function arguments.

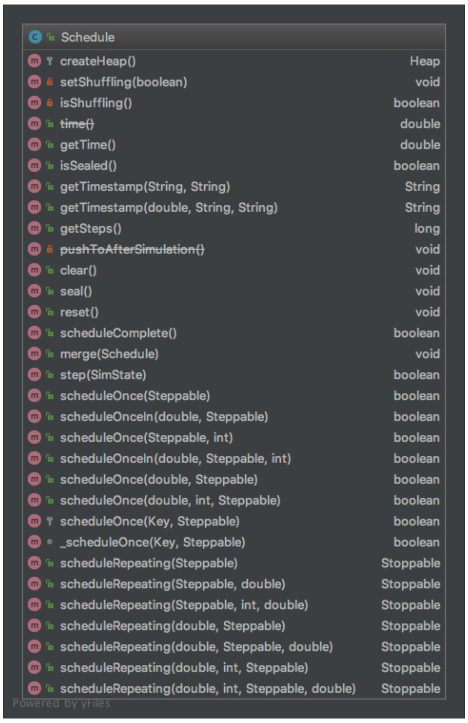


Figure 2: Schedule Class

By removing trivial methods (accessors, mutators, overloading methods), we can reduce the class down to:

- `getTimestamp`: Returns a given time in string format.
- `merge`: Merge a given schedule into this one.
- `step`: Moves the schedule forward by one implementation, skipping empty timesteps.
- `_scheduleOnce`: Adds the specified item to the schedule, to occur at the specified timestep.
- `scheduleRepeating`: Adds the specified item to the schedule, to repeat continously at specified interval.

By constructing a test that ensure the simulation `step` method works correctly, we will also indirectly test that the `_scheduleOnce` method works as expected during our setup process.

_scheduleOnce is a private method, so this will be called indirectly by one of its overloading methods: scheduleOnce(double, Steppable). This is **Test Case 1**.

Test Case 2 is similar, but will utilise the scheduleRepeating to ensure that the Schedule correctly manages recurring events.

Test Case 3 ...?

2.2 Test Case 4

2.3 Test Case 5

2.4 Test Case 6

2.5 Test Case 7

2.6 Test Case 8

3 Test Results

Test Case	Passed	Level	Description
1	✓	Unit	
2	✗	Unit	
3	✓	Unit	
4	✓	Unit	
5	✓	Unit	
6	✓	Integration	
7	✓	System	
8	✓	System	

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. I can assure that the core functionality, including the simulation scheduler works.

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] R. S. Pressman, *Software Engineering: A Practitioner’s Approach*, 7th ed. Boston ; London: McGraw-Hill Higher Education, 2010.

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

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

[5] 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]

[6] 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

[7] 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]

[8] 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`	2,961
`sim.field`	444
`sim.field.grid`	10,248
Total	13,653
Average	4,551

(a) Lines of Code

Package	**Class Count**
`sim.engine`	25
`sim.field`	6
`sim.field.grid`	14
Total	45
Average	15

(b) Class Count

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

(c) Cyclomatic Complexity

Package	**Dependencies**	**Dependants**
`sim.engine`	2	54
`sim.field`	1	24
`sim.field.grid`	2	20
Average	1.67	32.67

(d) Package Dependency

Figure 3: Code Metrics for the relevant MASON libraries

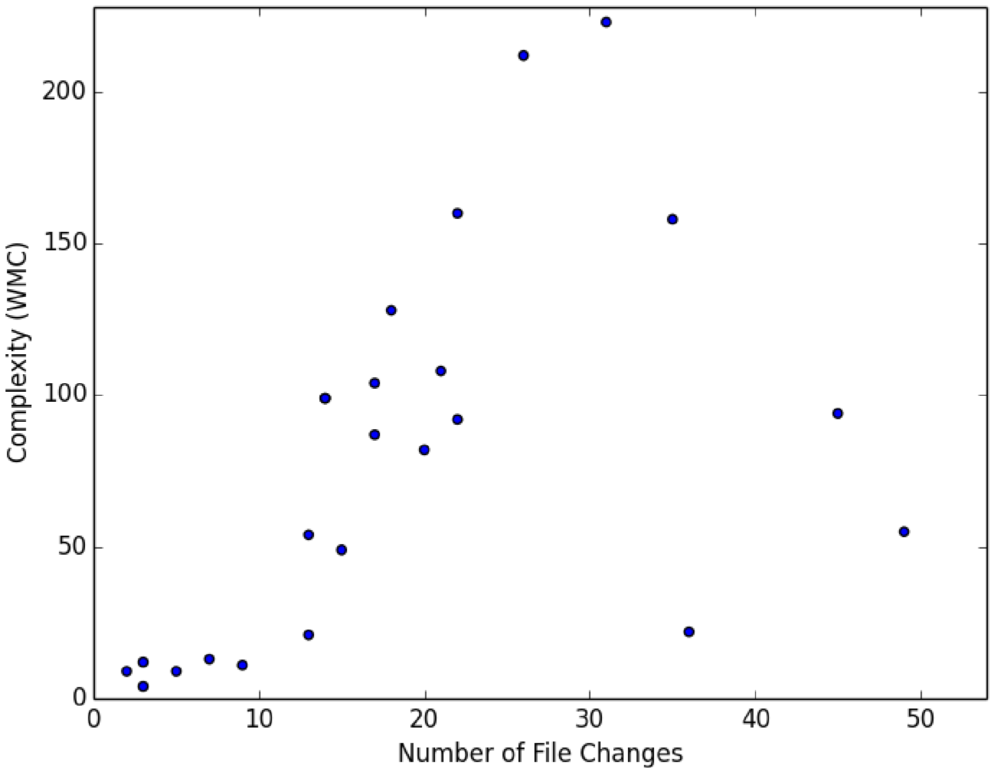


Figure 4: Cyclomatic Complexity and Number of File Revisions for classes in Test Scope

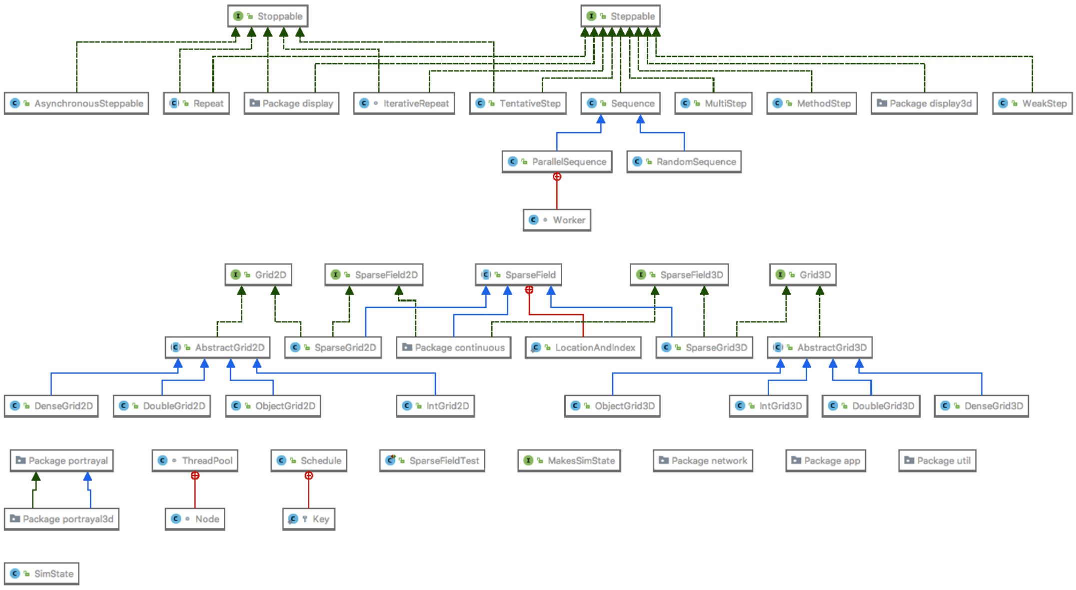


Figure 5: Reverse Engineered UML diagram of Class Hierarchy

Package	**Class**	**HeatBugs Coverage (%)**		**Virus Infection Coverage (%)**	
		Method	Line	Method	Line
`sim.engine`	AsynchronousSteppable				
	IterativeRepeat	60	70	60	70
	MethodStep				
	MultiStep				
	ParallelSequence	38	59		
	RandomSequence				
	Repeat				
	Schedule	41	51	41	50
	Sequence	11	12		
	SimState	31	18	31	18
	TentativeStep				
	ThreadPool	66	76		
	WeakStep				
`sim.field`	SparseField	21	34	21	35
`sim.field.grid`	AbstractGrid2D	10	1		
	AbstractGrid3D				
	DenseGrid2D				
	DenseGrid3D				
	DoubleGrid2D	11	7		
	DoubleGrid3D				
	IntGrid2D				
	IntGrid3D				
	ObjectGrid2D				
	ObjectGrid3D				
	SparseGrid2D	10	2		
	SparseGrid3D				

Figure 6: Code Coverage from Demo Simulation Runs