Software Testing

Y1481702

April 16, 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. This report documents testing of the latest version (19.0) of MASON, which has been obtained from the MASON website, with the aim to discover whether this software is dependable.

1.1.1 Tools

MASON Version 19.0 was run on macOS High Sierra (10.13.4) using the Java Runtime Environment and Development Kit version 1.8.0_131.

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. Specifically, the MetricsReloaded plugin[2] has been used for generating the statistics in Fig. 4.

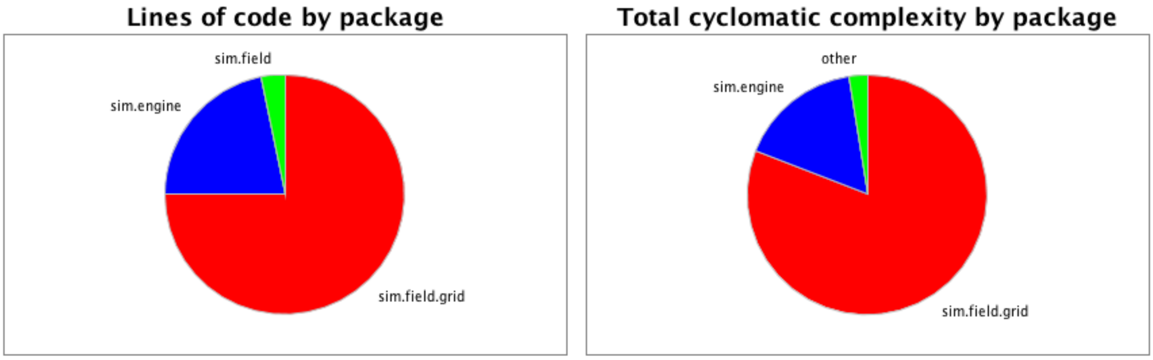
JUnit has been used to create automated testing for the software, this includes both unit and integration testing. Git version control has been used to manage the code for these JUnit tests. For a long term project, this would be particularly advantageous as any automated tests could be updated and versioned 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 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.



(a) Lines of Code

(b) Cyclomatic Complexity

Figure 1: Charts showing various metrics of MASON

Fig. 1a shows the proportional sizes of each of these three modules, while Fig. 10 shows their code components.

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[3, 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[4]. Unfortunately, a number of these metrics, including a history of bugs are not publicly available.

High cyclomatic complexity (Fig. 1b/Fig. 4c) 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. If we investigate this metric in further detail, it becomes clear that the majority of the more complex classes are in the `sim.field.grid` package. This metric has been plotted against the number of changes that have been committed to the class' source file. Files which are subject to frequent change have had more opportunity to develop additional bugs, therefore files with high complexity and frequent changes are particularly worrying and should be looked at.

Fig. 4d shows how the packages in the test scope interact with other packages in the system. In particular, `sim.engine` is directly relied upon by a large number of other packages. Additionally, code coverage (Fig. 11) has provided a good understanding of which parts of the software are regularly used while running simulations. Again `sim.engine` appears to be heavily used by the demo simulations.

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. The testing will focus on showing that the system fits our non-functional requirements of quality, reliability and, to a lesser extent, having acceptable documentation. 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[5, 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. In this case, our target users are the staff of a private biological research institute. The brief has not provided any specific use cases for the software, so this aim will not be trivial to accomplish. Previous biological implementations which utilised MASON[7] have been used to gain an insight into which MASON features are important to biological simulations.

Additionally I have assumed that MASON's bundled demo simulations give typical use cases 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. 11). Core simulation code that is used routinely by all simulations regardless of their customisations should be tested the most rigourously. Both the documentation[6, pp.85] and code coverage checks indicate that this type of code can be found within `sim.engine`.

1.2.3 Expected Behaviour

The expected behaviour of the software has been determined using the extensive MASON documentation[6] and first-party example implementations. Many of these examples have been implemented using the ContinuousGrid classes which are outside of the test scope.
The MASON documentation has conflicts, likely caused by parts of the document becoming out of date as the software is updated. One example of this is that in the tutorial section on Page 19, it describes different behaviour for the `-time` command line argument to that described in the formal documentation on page 91. In this case, and in general, the behaviour in the main documentation has been assumed correct and the tutorials assumed to be out of date. If a more ambiguous case had presented itself, exploratory testing could have been used to determine the expected behaviour.

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.[8] In this case, we will target our unit testing at ensuring the core software functionality is correct, specifically that simulations can be created and run.

As previously mentioned, it has been shown that approximately 20% of the code, contains 80% of our program's bugs. Our unit testing will aim for 60-70% code coverage of core code and ~20% of the overall application, as recommended in [8].

1.2.5 Integration Testing

Java is a heavily object-oriented (OO) language. This will affect our testing as in this type of software, much of the complexity is moved from algorithms in methods, to the connection of software components[9, pp.236]. We will therefore need a greater focus on integration testing than unit testing. Four out of the eight available tests will be devoted to integration testing.

1.2.6 System Testing

While testing the system as a whole will not help us find low-level bugs, it will give a high-level view of whether the bulk of the functionality appears to work as expected. Some system testing of the application will be conducted using the command-line as a black-box test. The MASON UI functionality is provided by the `sim.display` package, which is outside of the test scope.

System testing will also help us to verify the non-functional requirements that have been previously determined. In particular, our system testing will attempt to cover the requirement of acceptable documentation.

Acceptance Testing is a particularly useful form of system testing, which can be used for verifying non-functional requirements. It 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. However, it would helpful to discover if it is appropriate for the target audience. A particularly important non-functional requirement of our software is that it is fit for use, fit for purpose and can be operated effectively by our end users. 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[10]. Indeed the MASON documentation itself states this[6, pp.8].

In order for acceptance testing to be meaningful, it should be performed on potential users. Unfortunately, this means that this type of testing is out of reach for this report as target users are unavailable.

1.2.7 Mutation Testing

Mutation testing is often used to determine the effectiveness of a test-set at discovering bugs in the system. As we already have a very limited number of tests, mutation testing will not be used.

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 Case 1

Code coverage runs (Fig. 11) have shown that the `Schedule` class is frequently used during simulations. In particular, one of the most used methods is `getTimestamp`. As such, it should be tested to show it is performing correctly.

For this test case, a new `Schedule` object is instantiated. No setup is done, and no items added to the schedule. The `getTimestamp` method is called with `3.0`, `"Before"` and `"Completed"` given as each of the three arguments. The method is expected to return the string `"3"`.

2.2 Test Case 2

The `AbstractGrid` classes are among those with the highest cyclomatic complexity. The other grid classes inherit code from these abstract classes, except SparseGrid2D which duplicates a great deal of code instead (Fig. 8), implying this code is highly important. `AbstractGrid2D` also has the highest number of historic code changes out of the `sim.field.grid` package. These two metrics are worrying given that the simulation runs show that this class is routinely used to some extent. Unit Test cases have been derived to target the methods in this class with the highest complexity (Fig. 9), specifically `getHexagonalLocations`.

`AbstractGrid2D` cannot be directly instantiated for this test, so `DoubleGrid2D` will be used as it extends the class without overriding `getHexagonalLocations`. A new `DoubleGrid2D` object is instantiated with width and height both equal to 150. The method is called with the origin x and y both equal to 120 and dist equal to 20. The grid mode is set to bounded, origin is set to be included and empty `IntBags` are passed to the method. The expected output has been independently calculated using a hexagon point algorithm.

These independently calculated point values have been manually verified as correct using a 2D graphing software.

2.3 Test Case 3

Another method of `AbstractGrid2D` which has a high cyclomatic complexity is `getRadialLocations`. This method will also be tested to ensure its behaviour is expected. In particular testing at boundaries is most likely to show ambiguities in the specifications[5, pp.25], thus revealing bugs. In the case of this method, a boundary case is checking that the values correctly wrap around when the grid is in toroidal mode. A new `DoubleGrid2D` is created with width 200 and height 160. The `getRadialLocations` method is invoked with an origin of (150, 150) and a radius of 20 so that the circle should wrap around the end of the grid in the y-direction. In this case, include origin is set to true and empty `IntBag`s are passed into the method.

Assertions are made to ensure that the size of the bags for x and y are both equal to 1345. The bags are then compared against the expected values as calculated independently by the midpoint circle algorithm. These independently calculated point values have been manually verified as correct using a 2D graphing software. In order to check that the toroidal wrapping works correctly, the remainder operator is applied to these circle algorithm results with the height/width of the field.

2.4 Test Cases 4-5

As previously stated, our initial integration testing will focus on the simulation scheduler which provides the core functionality for MASON. This scheduler works alongside the `SimState` class and custom implementations of the `Steppable` interface to provide the functionality that simulates the passing of time.

While this integration between classes may appear to be complex at first glance, with numerous methods available (Fig. 2), many of its methods are simply overloads, due to Java's lack of real support for default function arguments. Testing a single example of `scheduleOnce` function, will provide a reasonable assurance that the functionality works, as code is shared with each overloaded instance calling the private `_scheduleOnce` method.

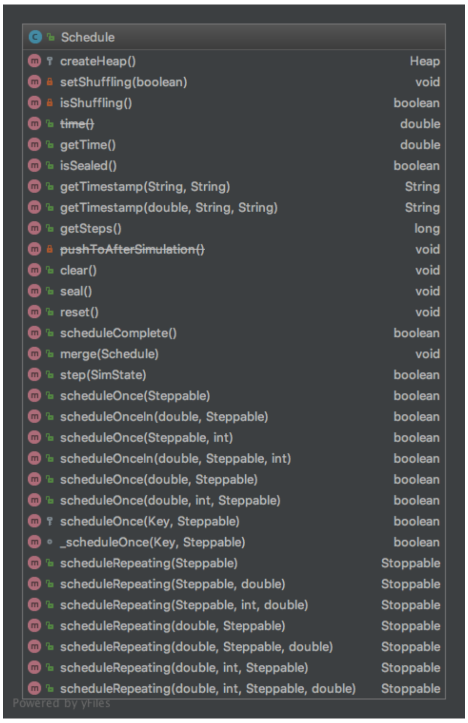


Figure 2: Schedule Class

Test Case 4 will add a `Steppable` object to the `Schedule` and ensure that the `step` method work as expected. A by-product of this test, is the assurance that the `_scheduleOnce` method also works as expected. As `_scheduleOnce` is a private method, it should not be used by an integration test. Therefore, it will be called indirectly by one of its overloading methods: `scheduleOnce(Steppable)`. Other overloading methods could be used instead with no difference on the test case as the scheduler skips any empty timesteps. In order to facilitate this test, a simple `Increment` class (Fig. 5) has been created.

The scheduler will be stepped a number of times (10) to ensure that the `Increment` value is correctly stepped. The expected behaviour is that the `Increment` class' step method will be called once, at the first simulation step. Future steps of the simulation should result in no effect.

Test Case 5 is similar, but will utilise the `scheduleRepeating(Steppable)` to ensure that the Schedule correctly manages recurring events. As mentioned, the scheduler skips any empty timesteps, so providing a different start time or time interval will make little difference to the test. Again, the scheduler will be stepped a number of times (10) to ensure that the value is stepped multiple times as expected. Here, the expected is that the `Increment` class' step method is called each time (10) the simulation's schedule is stepped.

2.5 Test Case 6-7

A simple simulation case will be created and tested to show that `sim.engine` can be integrated with `sim.field.grid`. The aim is to show that simulations can be *created* using MASON. The test cases will be run as a grey-box test, assuming only knowledge given by public methods and interfaces of each package. This is how end-users would create simulations using MASON. As the end-user is using the system as a framework in their own programming, this case is on a blurred line between integration and system testing.

A very simple ABM simulation which models the collision of two objects has been created. The `Steppable` object class has been created with a position and direction as shown in Fig. 6. Each time the objects are stepped, they move one square diagonally in their current direction. If two objects collide, the simulation ends. Given this implementation, the relative speeds of the object can be set by adding a time interval when adding the objects to the simulation schedule. We cannot easily test the `doLoop` function for this as it exits the program not allowing our post assertions to be run.

In **Test Case 6**, a `DoubleGrid2D` with both width and height of 20 is created to track the locations of the objects. Two of these objects are created at (0, 0) and (19, 19) both moving towards the center of the grid. The object at (0, 0) is scheduled to move with interval 2.0, while the other will move with interval 3.0. The simulation is expected to last 24 timesteps, with cars colliding as they both occupy cell (11, 11). As the simulation does not get killed until the timestep is ended, car is expected to move to cell (12, 12) after the collision. Assertions are made to ensure that this calculated behaviour is matched by the simulation. *N.B. The `Schedule.step` method ignores and skips empty timesteps. This method therefore only needs to be called 16 times (or alternatively while `Schedule.scheduleComplete` is false.)*

Test Case 7 uses a similar setup to the previous test. The aim of this test case is to ensure that the the grid objects can detect their neighbours in simulation steps. The step function is enhanced to detect collisions when the objects are 3 cells apart- before they might occur. This is done by calling `DoubleGrid2D.getRadialNeighbours` with the current position of the object, the origin should not be included. At this point the agents randomly change one of their directions to avoid each other. The grid is made toroidal and the object class movement functions updated accordingly. The simulation should now be infinite, but clearly this is unverifiable. The simulation is stepped and assertions are made that a collision is detected at the the 22nd timestep and that one of the objects changes their direction.

2.6 Test Case 8

This test case will verify that the tool is able to run from the command-line, correctly reacting to user configurations. The case will be run as a black-box test, assuming no knowledge of simulation classes. The aim is to show that simulations can be *executed* using MASON. The HeatBugs example simulation will be used as it utilises Field classes that are in scope (as opposed to classes in `sim.field.continuous`). The aim of this test is to verify that the main documentation for MASON is correct, despite a conflict in the tutorial section. Well-written and correct documentation is a particularly important functional requirement for MASON.

The simulation will be run until a specific simulation time. The command run is `java sim.app.heatbugs.HeatBugs -until 200000`. The expected behaviour is for the system to output several lines of text showing the Job Number, Current Steps, Simulation Time and Rate of Steps to the console. *N.B. The documentation here is conflicting. On Page 91, the `-time` argument specifies that it will 'print a timestamp every T simulation steps' whereas page 19 states that the argument means the simulation will 'run for a limited time', behaviour that 19 attributes to the `-until` argument.*

3 Test Results

Case	Pass	Level	Expected	Actual	Details
1	✗	Unit	The entered timestamp (`double 3`) should be returned as a string.	The before simulation string argument is returned.	After a review of the code, it's clear that the time passed into the method is ignored in favour of the current simulation time.

Case	Pass	Level	Expected	Actual	Details
2	✓	Unit	A hexagonal set of points is created, such that the points match the expected values which have been independently calculated.	*As Expected*	N/A
3	✗	Unit	The circular points that are generated should wrap around to 0 when they become greater than the height of the coordinate system.	The points wrap around, but go past the origin and become negative.	This is a **major** bug as it may significantly impact the results of simulations.
4	✓	Integration (Class)	`i` variable is incremented once, in first simulation step.	*As Expected*	N/A
5	✓	Integration (Class)	`i` variable is incremented once for each simulation step.	*As Expected*	N/A
6	✓	Integration (Package)	Objects move through the grid and collide in cell (11, 11) at the 24th step.	*As Expected*	N/A
7	✓	Integration (Package)	A collision is detected and avoided at timestep 22.	*As Expected*	N/A
8	✓	System	The simulation runs and stops after 200,000 steps.	*As Expected*	Documentation is conflicting, but it was possible to discern the correct behaviour.

4 Test Summary Report

A wide spectrum of testing has been performed including all levels of the software. This testing has aimed to cover the core (most-commonly used) functionality of the system as thoroughly as possible. The aim of this is to assert that the main requirement

Package	Class	Method	Line
`sim.engine`	18	18	18
`sim.field`	0	0	0
`sim.field.grid`	25	4	4

Figure 3: Code Coverage (%) from Unit and Integration Testing

of the end-users (creating agent-based
simulations) can be achieved using this library. While this testing has been limited to select classes, they do achieve a good level of code coverage over the essential functionality. For example, *Test Case 2* achieves line and condition coverage of 81% and 73% respectively over its target method `getRadialLocations`. This code coverage is similar to that achieved by the MASON demo projects which, in are absense of any clearer specifications, are our expected use cases. *Test Case 1*, which covers this core functionality found a medium level bug in the `Schedule.getTimestamp` method. This bug is unlikely to significantly effect the output of our simulation's progression, but may affect the ability to track whether a simulation has begun or ended.

Testing has also been done on parts of the code with the highest complexity, this is where bugs are more likely to be found. Despite the limited resources, a *major* bug in the code was detected here (*Test Case 2*). The bug affects the calculation of y values for neighboring points when wrapping around the edge of a toroidal grid. This type of grid has previously been used in MASON biological simulations[7] so it can assumed as a dependency of future simulations created by the end users. While this is limited to a specific branch of code and may not be run regularly, if it is reached it may significantly change the results produced by the simulation. This is highly underdesirable. As this code is *also* duplicated in a different class, it effectively means that two bugs have been found with this test. The presense of these bugs is an indicator that the software may not have been tested as rigorously as could be expected.

Additionally, *Test Case 8* covers a case which has conflicting definitions in the documentation as discussed in Section 2.6. Conflicting documentation is likely confuse the end-user of the software. This is likely to be particularly damaging in this case as the end-users only have a basic level of programming. Even with high-quality documentation MASON would already likely be a steep learning curve for the users.

The core functionality MASON package does seems to broadly function as expected. MASON can be used to produce and run agent-based simulations, but a number of bugs in the system may affect the correctness of results. It is likely that users with only basic training would be unable to debug any incorrect results caused by these bugs, providing they are even spotted. As such, I believe is is worth reviewing other simulation libraries to see if they could provide a better experience for our end users than MASON. If no better options are found, I would recommend that a reasonable amount of acceptance testing should be performed with a group of end users from the company before MASON is adopted. This will help to verify if the software is acceptable for the users' Java programming abilities, as research[6, 10] has suggested otherwise.

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] B. Leijedekkers. (2016, Sep) Metricsreloaded. [Online]. Available: https://plugins.jetbrains.com/plugin/93-metricsreloaded [Accessed: Apr. 10, 2018]

[3] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 8th ed. Boston ; London: McGraw-Hill Higher Education, 2010.

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

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

[6] 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]

[7] 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

[8] 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]

[9] P. Ammann and J. Offuit, *Introduction to Software Testing.* Cambridge University Press, 2008.

[10] 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.

[11] *Basic MASON Model and its Relationship to GUI Controllers*, George Mason University. [Online]. Available: https://cs.gmu.edu/ eclab/projects/mason/docs/SimulatorLayout.pdf [Accessed: Apr. 10, 2018]

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 4: Code Metrics for the relevant MASON libraries

```
class Increment implements Steppable{
        private int i = 0;

        @Override
        public void step(SimState state) {
                i++;
        }
}
```

Figure 5: Simple Increment class used for Unit Testing

```java
public class Car implements Steppable {
        private int x;
        private int y;
        private boolean x_reverse;
        private boolean y_reverse;

        public Car(int x, int y, boolean x_reverse, boolean y_reverse){
                this.x = x;
                this.y = y;

                this.x_reverse = x_reverse;
                this.y_reverse = y_reverse;
        }

        @Override
        public void step(SimState state) {
                Collision world = (Collision)state;
                world.grid.set(x, y, null);

                x += x_reverse ? -1: 1;
                y += y_reverse ? -1: 1;

                if(world.grid.get(x, y) != null){
                        world.kill();
                }

                world.grid.set(x, y, this);
        }
}
```

Figure 6: Simple moving object class. Accessors and mutators have been omitted.

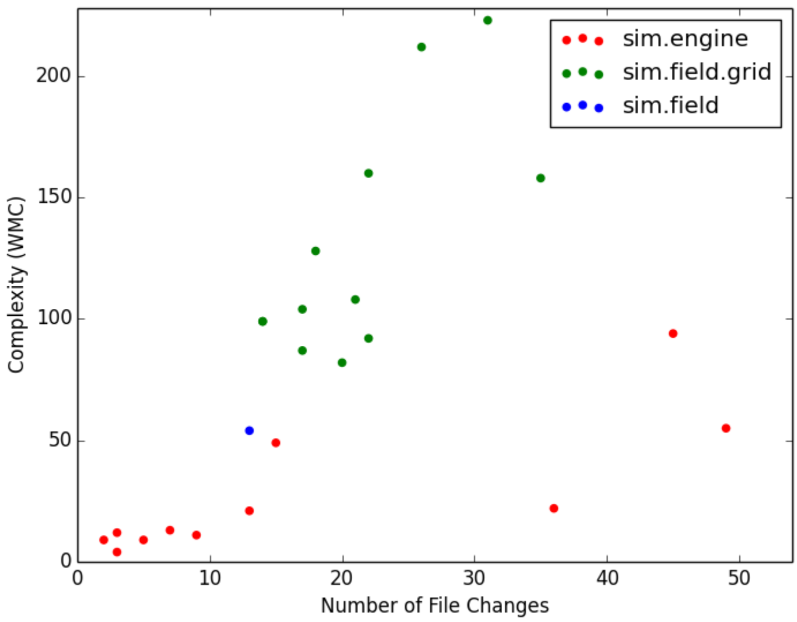


Figure 7: Cyclomatic Complexity and Number of File Revisions for classes in test scope

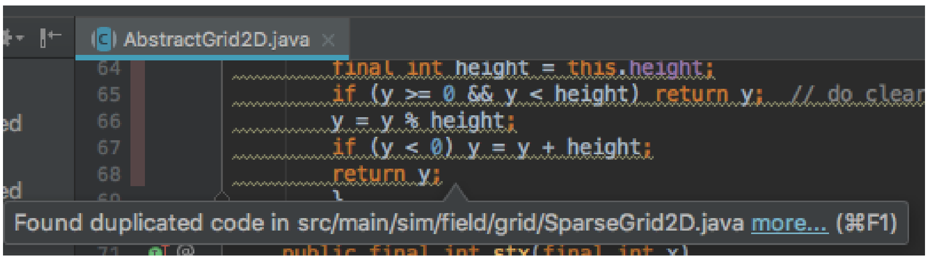
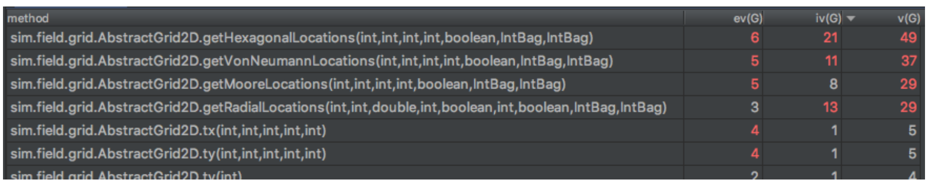


Figure 8: Screenshot showing IntelliJ's detection of duplicated code between Abstract-Grid2D and SparseGrid2D



method	ev(G)	iv(G)	v(G)
sim.field.grid.AbstractGrid2D.getHexagonalLocations(int,int,int,int,boolean,IntBag,IntBag)	6	21	49
sim.field.grid.AbstractGrid2D.getVonNeumannLocations(int,int,int,int,boolean,IntBag,IntBag)	5	11	37
sim.field.grid.AbstractGrid2D.getMooreLocations(int,int,int,int,boolean,IntBag,IntBag)	5	8	29
sim.field.grid.AbstractGrid2D.getRadialLocations(int,int,double,int,boolean,int,boolean,IntBag,IntBag)	3	13	29
sim.field.grid.AbstractGrid2D.tx(int,int,int,int,int)	4	1	5
sim.field.grid.AbstractGrid2D.ty(int,int,int,int,int)	4	1	5
sim.field.grid.AbstractGrid2D.ty(int)	2	1	4

Figure 9: `AbstractGrid2D` methods with greatest cyclomatic complexity

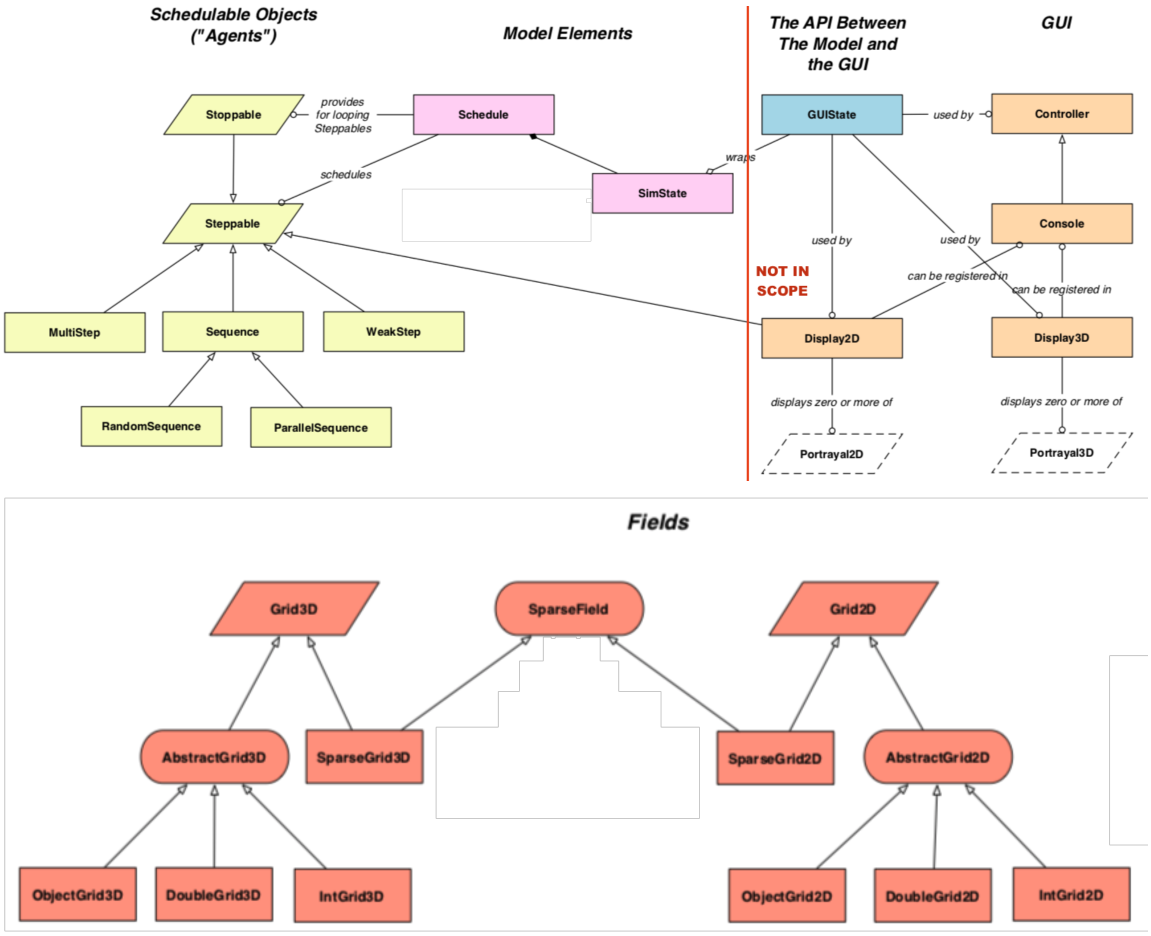


Figure 10: UML diagrams of hierarchy for classes in the test scope[11]

Package	Class	HeatBugs Coverage (%)		Tutorial 4 Coverage (%)		Mouse Traps Coverage (%)	
		Method	Line	Method	Line	Method	Line
`sim.engine`	AsynchronousSteppable						
	IterativeRepeat	60	70	60	71		
	MethodStep						
	MultiStep						
	ParallelSequence	38	59				
	RandomSequence						
	Repeat						
	Schedule	41	51	38	49	35	46
	Sequence	11	12				
	SimState	31	18	55	48	31	18
	TentativeStep						
	ThreadPool	66	76				
	WeakStep						
`sim.field`	SparseField	21	34	24	37		
`sim.field.grid`	AbstractGrid2D	10	1	5	0	5	0
	AbstractGrid3D						
	DenseGrid2D						
	DenseGrid3D						
	DoubleGrid2D	11	7	7	5		
	DoubleGrid3D						
	IntGrid2D					17	10
	IntGrid3D						
	ObjectGrid2D						
	ObjectGrid3D						
	SparseGrid2D	10	2	4	1		
	SparseGrid3D						

Figure 11: Code Coverage from Demo Simulation Runs