

Metrics Report
Including Analysis of Coverage, Complexity, Cohesion, and
Coupling
For
Swords and Sorcery(S&S)
Version 1.0

Prepared by:
University of Idaho Computer Science 383 Class, Spring 2014

Prepared for:
Dr. Clinton Jeffery

May 9, 2014

Swords and Sorcery Metrics

Table of Contents

1 Introduction

This document includes reports and analysis of several software metrics of our computer adaptation of the Swords & Sorcery board game. The project and its documentation are property of the Spring 2014 Software Engineering class at the University of Idaho and its constituents.













































Through this document, the development team aims to describe the software using several measurements such as Test Coverage, Complexity, Coupling, and Cohesion. Analysis of these metrics and what they mean for the software follow.

2 Test Coverage

The project's 79 automated test cases resulted in 9% instruction coverage and 5% branch coverage of the entire project. While the average is low for the whole project, some elements exhibit higher levels of coverage.









The leader in instruction coverage was the MoveCalculator package, which experienced 53% instruction coverage (and 44% branch coverage). Meanwhile, the most branch coverage was attained by the ssterrain package, at 53% (and 48% of instructions).

The top level coverage data is summarized by the table below.





























Element	Missed Instructions	Cov.	Missed Branches	Cov.
MoveCalculator		53%		44%
ssterrain		48%		53%
sshexmap		33%		29%
Units		25%		12%
systemServer		8%		3%
sscharts		4%		1%
mainswordsorcery		0%		0%
CombatSimulator		0%		0%
Spells		0%		0%
Character		0%		0%
buildmapfilel		0%		0%
Spells.PL_6		0%		0%
Spells.PL_3		0%		0%
Spells.PL_2		0%		0%
Spells.PL_4		0%		0%
Spells.PL_1		0%		0%
default		0%		0%
Spells.PL_5		0%		0%
unitdetailsdisplay		0%		0%
Spells.PL_7		0%		0%
ObjectCreator		0%		n/a
phase		0%		n/a
Total	50,483 of 55,448	9%	6,570 of 6,929	5%

Ignoring percentages, the package with the most covered instructions was ssterrain with 1,374 covered instructions. Of its 39 classes, only 10 were missed by the automated unit tests. All hex edge types had 100% coverage except for HEFord. While the various terrain types were not covered well, TerrainType itself had 94% instruction coverage.

You'll notice from the table that the sscharts package sticks out sorely as far as coverage goes. This is due largely to the very large and completely untested class CreatUnits, which has 14,363 instructions. This is demonstrated below.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
CreatUnits		0%		0%
ArmyCombatResultsTable		0%		0%
RandomEventTable		0%		0%
Scenario		69%		55%
Total	16,158 of 16,771	4%	3,298 of 3,320	1%

While sscharts gets a bad report due to one untested monster, the systemServer package warrants its bad marks by leaving several significant classes untested, as shown by the following.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
ClientObject.ServerReceivingThread		0%		0%
ClientObject.ServerListenerThread		0%		0%
MessageUtils		0%		0%
NetworkClient.ListenerThread		0%		0%
NetworkClient.ClientReceivingThread		0%		0%
NetworkClient.ClientListenerThread		0%		0%
ClientObject.ListenerThread		0%		0%
NetworkClient		25%		12%
NetworkClient.ClientCommandThread		0%		0%
NetworkServer		39%		27%
ClientDataForm		0%		0%
ClientObject.new Object() {...}		0%		n/a
NetworkClient.new Object() {...}		0%		n/a
ClientObject		0%		0%
Lobby		0%		0%

The full coverage report can be accessed in the sworsorc repository as [sworsorc/.jacocoverage/report.htm](#) for full details.

sshexmap has 16 classes and 9 were missed. The best coverage was with the Hex class with 100% coverage, followed by HexMap with 78% and MapHex with 69%. Of the 140 methods in all the sshexmap classes 97 were missed and the 801 lines, 550 were missed.

Units has 291 classes and 234 were missed. The classes that had 100% coverage were Zeppelin, Bow, RangedLandUnit, LandUnit and FlyingUnit. Of the 156 methods in all the classes of Units 110 were missed and of the 922 lines 716 were missed. The UnitPool had 47% coverage.

Because test coverage in most cases was extremely poor for this project, we have done very little to assure that the componenets of the project perform as they are expected to. This means that very little can be said about the quality of the product. Our tests simply cannot conclude that it performs well. When we analyze the complexity of various components later, we will see that some of our worst coverage overlaps with our most complex methods. It becomes impossible to assure product quality when the components that are most difficult to understand are also unchecked.

3 Complexity

When setting out to to develop a semester project for this course, we knew that it would have to be complex enough to take a multi-person team of developers a semester to create. These complexity metrics serve the dual purpose of assuring the reader that this has occurred while also pointing out exactly how frightening it is that test coverage is so low.

3.1 Lines of Code (LOC)

The project totals 49,862 lines of code. These are broken up between several packages, with the following five having the highest LOC count:

- src: 40903
- model: 10485
- src: 10218
- CombatSimulator: 10218
- utilities: 9777

The following five classes are the longest as well:

- Main: 6597
- CreatUnits: 6575
- NetworkClient: 1281
- CharacterJ: 1222
- CharacterJ: 1222

You'll observe that despite breaking the project up into several modules, there are still some files that total over six thousand lines of code. This could degrade both the readability and maintainability of the code base, as it requires a great deal of effort just to understand the code of one class.

3.2 Class Count

Even with some classes being so voluminous, we still managed to acquire 303 total classes throughout the course of development. Once again, this can have a poor effect on the understandability of the project. The packages with the most classes are listed.

- src: 244
- model: 119
- src: 70
- CombatSimulator: 70
- doc: 59

3.3 Method Count

There are a total of 1614 methods. These are more evenly distributed through classes, as shown by the top five classes by method count. While this is still a lot of methods for a single class to contain, at least there are no outlying monsters in this respect.

- NetworkClient: 42
- CharacterJ: 42
- CharacterJ: 42
- MainMap: 35
- MapHex: 30

3.4 McGabe's Cyclomatic Complexity

While the average cyclomatic complexity of the project's methods was a comfortable 3.33, outliers emerge that are purely terrifying. The unchecked CreatUnits class provides one of these, while the others are equally unchecked methods within Main.

- Main::Create_unit1: 252
- Main::Create_unit2: 252
- Main::Create_unit4: 252
- Main::Create_unit3: 252
- CreatUnits::Create_unit2: 202

These monstrously-complex methods include nested switch statements (thus the high volume of paths possible), and it is unclear what they are meant to do. This has a disastrous effect on readability, comprehension, and maintainability. Furthermore, something this complex and untested is likely to be broken, meaning that maintainability and quality are also affected in that sense.

4 Coupling

Our metrics tools gave the following measurements of coupling.

Loose Class Coupling (LCC) averaged 0.151 with a maximum of 1 and a minimum of 0 for the project. High Class Coupling (HCC) on the other hand also ranged from 0 to 1 and averaged 0.111.

From this data, we can conclude that about 15% of our classes are loosely coupled, meaning they require little knowledge of the definitions provided in other classes. This means that these classes may experience an increase in readability, maintainability, and testability because they can stand alone.

On the other hand, about 11% of our classes were tightly coupled, which means understanding, testing, and executing them requires knowledge of the definitions in other classes. This can be detrimental to maintainability, testability, and readability.

5 Cohesion

The five measures of Lack of Cohesion had the following minimums, maximums, and averages for the project.

1. 32, 299, 43
2. 28, 273, 39
3. 2, 6, 5
4. 2, 6, 4
5. 0.925, 0, 0.553

From these metrics, we can make the following conclusions.

1. There is a minimum of 32 and at most 299 pairs of methods in a single class that do not share attributes. There is an average of 43 methods per class with low-cohesion.
2. There is a minimum of 28 and at most 273 methods after subtracting the number of pairs of methods that share attributes from the number of pairs of methods that do not share attributes.
3. There is a minimum of 2 and maximum of 6 disjoint components per class with an average of 5 disjoint components per class.
4. Similar to Method3 but measuring method invocations instead of node edges.
5. $LCOM5 = \frac{a - kl}{(l - k)}$, where l is the number of attributes, k is the number of methods, and a is the summation of the number of distinct attributes accessed by each method in a class. The average lack of cohesion across all our classes using LCOM5 is 55.3%

Unfortunately, some of these measures of Lack of Cohesion are fairly high, meaning we have fairly poor cohesion between our classes. Unfortunately, a lack of cohesion can mean that the methods and data in classes are unrelated. This has the effect of reducing readability and maintainability of methods. Additionally, since classes with unrelated elements are likely to be constructed in a design-specific manner that will limit their reusability.

6 Appendix

A report of several more metrics can be found as generated by the NetBeans Source Code Metrics plugin at <sworsorc/doc/EndingDocumentation/Metrics/metrics.xlsx>.