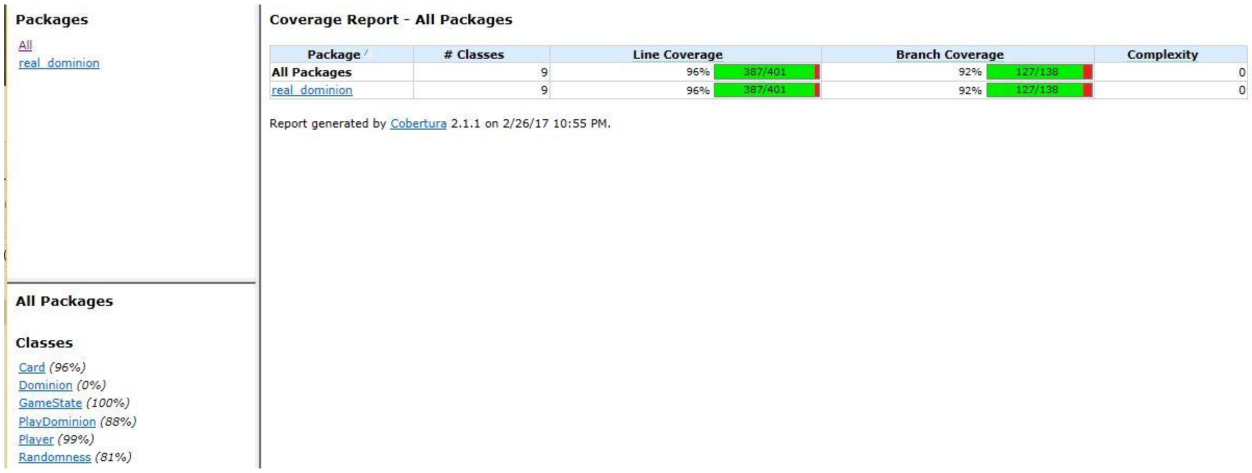
Tyler Jones
Assignment #2 Write-up
2/27/17
CS 362

Compared to Assignment 1, this test generation tool that I used, EvoSuite, actually had higher coverage according to my Cobertura report. In Assignment 1, my tests covered roughly 80% of my Dominion code, this was partially due to unnecessary classes within my code that didn't need to be tested and were just auxiliary functions that could be used for developmental testing, however the coverage was still higher than I believe my coverage from Assignment 1 would have been even if I had accounted for this fact. In this Assignment, as can be seen below, my coverage and branch coverage were both close to 100%, which is satisfactory for this assignment.


Packages
All
real_dominion

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	9	96%	387/401	92%	127/138	0
real_dominion	9	96%	387/401	92%	127/138	0

Report generated by Cobertura 2.1.1 on 2/26/17 10:55 PM.

All Packages

Classes
Card *(96%)*
Dominion *(0%)*
GameState *(100%)*
PlayDominion *(88%)*
Player *(99%)*
Randomness *(81%)*

The tool did miss some bugs that I knew of. However, they weren't functional bugs, they were more semantic bugs that were specifically related to the behavior of my kingdom cards. In my Assignment 1, the 5 intentional errors that I allowed to exist in my program were incomplete or incorrect behavior of my kingdom cards. They didn't alter the game state in an illegal way, but rather they just weren't correct based of the intended behavior of the cards according to the official dominion rules. This was obviously impossible for the test generation software, EvoSuite, to cover. The card either had the behavior that I coded or didn't and EvoSuite has no context about whether the card is doing what it should be doing based off of Dominion's description. It was at this point that it became apparent to me the inherent flaws with testing and test generation software; if you don't know what the end behavior should be, it is difficult to test.

The problems I faced while using this tool were numerous. Firstly, it was an absolute nightmare to install, as were all the suggested test tools. Wherever there was documentation for EvoSuite, it was unfortunately geared towards UNIX commands and I am using windows. Additionally, there was a little bit of Eclipse documentation as EvoSuite actually has a plugin that is supported by Eclipse, however this section of EvoSuite's site was incomplete and stated "Needs to be Updated" so troubleshooting had to be done through third party Stack Overflow threads etc. which proved to be incredibly difficult as well.

My overall experience and findings with this tool were satisfactory. For my longer classes, where I had more lines of code and more methods, EvoSuite seemingly generated longer tests and also more tests. This is expected behavior and wasn't surprising.

The other tools that I looked at that turned out to be absolutely horrendous to install and use were randoop, JCrasher, Palus, Korat, and AVMFramework before finally settling on EvoSuite. I was able to download and find most of my instructions for EvoSuite usage off their main site: http://www.evosuite.org/documentation/tutorial-part-1/.

I ended up making my decision to use EvoSuite based on the fact that it was simply the only one that I had any success with in terms of installation. Once I actually was able to install EvoSuite, and not run into roughly five thousand incomprehensible errors, it actually worked seamlessly. I was very impressed with the amount of code coverage it was able to generate based on each class, and how comprehensive each one was. As I already said, what worked poorly was the installation. Installation from the command line was undoable, and I was lucky that I was able to install it on Eclipse. I think it could be drastically improved, along with all of the other test generation software by GREATLY improving documentation. If these pieces of software are truly open source and useful, I would assume the developers would want to make it accessible to even undergraduates in a Software Engineering class. All of the installations seemed much more geared towards someone who greatly understands maven, the command line (both unix and windows), java, eclipse, and the software that they have created, which is simply not the level we are at yet. It was a huge learning curve, and troubleshooting any error message felt like throwing up a prayer.

ii) For my RandomTestDominion.java file, I was able to improve upon the tests coverage by actually playing an entire game of dominion by incorporating all of my classes together. One thing that the generated tests were missing was using all of the classes together to actually turn into a full game of Dominion, which is of course the end result.

iii) When I was finding a bug in my code, I didn't actually need to use a debugger. Typically I use a debugger when dealing with memory and complex array indexing, because it is easy to get a Segmentation Fault error message. A debugger is useful in these situations because you can step through the code line by line by setting up break points and see what the value of a given memory address is at any given line, and if it lines up with what you are expecting that value to be. For this assignment, I didn't use a debugger, but rather just simply had my code print out the variable in question for me to check if it's right which proved simple enough.