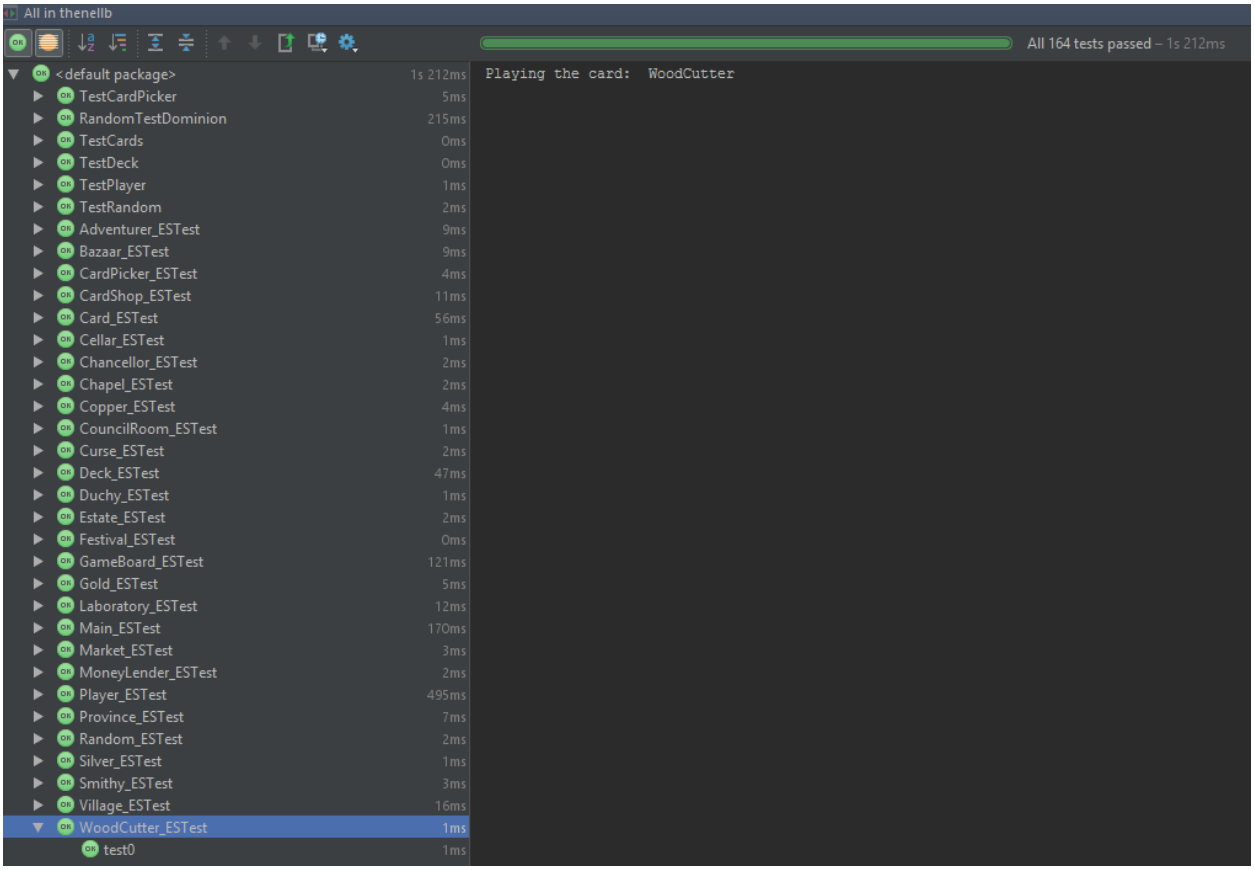
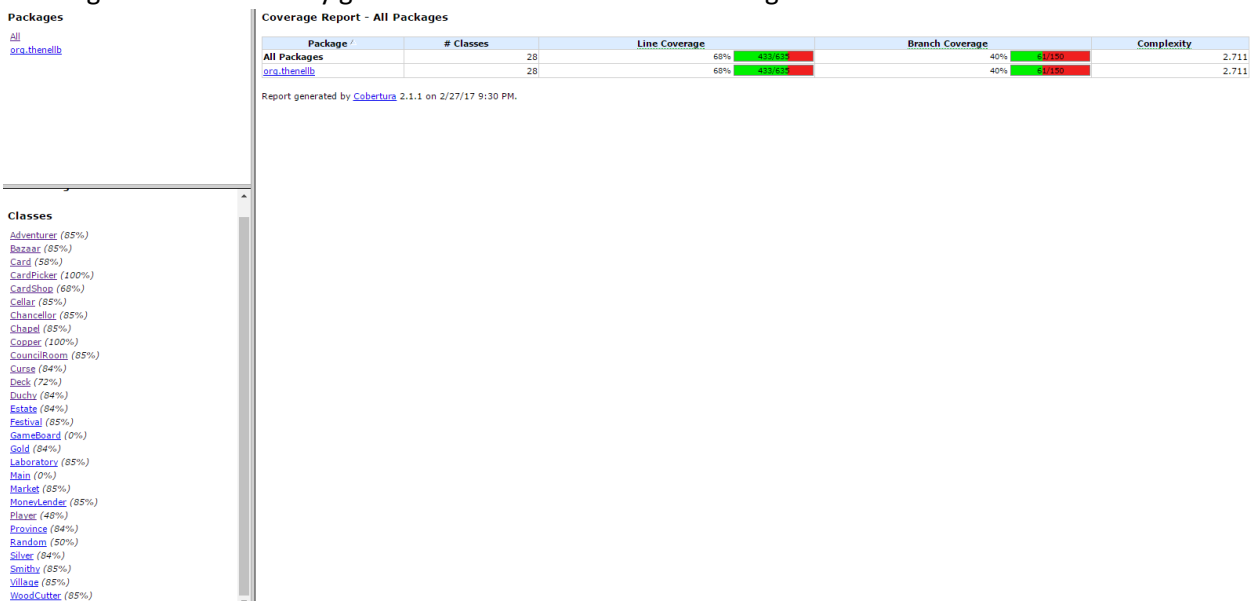
Result of running all 164 tests (both manual and random tests)



Coverage with no randomly generated tests: 68% overall coverage



Packages

All
org.thenelb

Coverage Report - All Packages

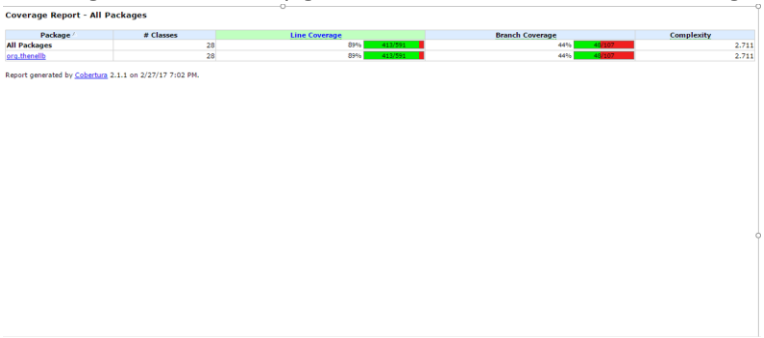
Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	28	68% 433/635	40% 61/150	2.711
org.thenelb	28	68% 433/635	40% 61/150	2.711

Report generated by Cobertura 2.1.1 on 2/27/17 9:30 PM.

Classes

Adventurer (85%)
Bazaar (85%)
Card (58%)
CardPicker (100%)
CardShop (68%)
Cellar (85%)
Chancellor (85%)
Chapel (85%)
Copper (100%)
CouncilRoom (85%)
Curse (84%)
Deck (72%)
Duchy (84%)
Estate (84%)
Festival (85%)
GameBoard (0%)
Gold (84%)
Laboratory (85%)
Main (0%)
Market (85%)
MoneyLender (85%)
Player (48%)
Province (84%)
Random (50%)
Silver (84%)
Smithy (85%)
Village (85%)
WoodCutter (85%)

Coverage with randomly generated tests: 89% overall coverage

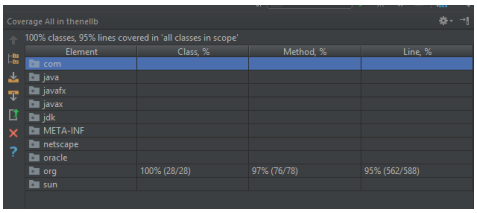


Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	28	89% 413/591	44% 44/107	2.711
org.thenelb	28	89% 413/591	44% 44/107	2.711

Report generated by Cobertura 2.1.1 on 2/27/17 7:02 PM.

Secondary Coverage tester: 100% Class coverage, 95% line coverage, 97% method coverage.



Coverage All in thenelb

100% classes, 95% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
com			
java			
javafx			
javax			
jdk			
META-INF			
netscape			
oracle			
org	100% (28/28)	97% (76/78)	95% (562/588)
sun			

i. Section Tool:

Personally, I believe this second one is a better representation of the coverage because it does not rely on separate dependency libraries. It is a built-in utility of my IDE. In Contrast to Assignment one, this is much better coverage because in Assignment one, I barely had any test functions to begin with. The coverage from Assignment one was below 50% coverage. The tool did run into a few errors which can be attributed to the bugs. There was a total of 7 test failures, which seems

reasonable since there is built in bugs. One issue I had was trying to find the right random test generator tool. While there are many testing tools/utilities out there, a lot of them require extensive setup and dependency imports. I settled on EvoSuite because my IDE has a built-in plugin for it which made setup on my project a breeze. I also had some issues with the Cobertura coverage report tool. It was having issues with my preexisting dependencies and would often times simply not generate a coverage report. I felt like I was rolling dice every time I tried to generate the report. However, the built-in coverage report tool in my IDE worked like a charm every time. I think this might be another reason the coverage reports returned different values of coverage. One thing I did not like about EvoSuite is how long it took to generate the random tests. It stated that it would take almost an hour to generate everything. Thankfully, after a little research, I was able to figure out a way to generate the tests with parallel processing, cutting it down to a mere 15 minutes. The tools I looked at before settling with EvoSuite are below:

- CodePro Analytix (deprecated)
- CoView
- AgitarOne Automated Junit Generation
- Jtest
- Junit Factory
- Randoop
- Palus

While I did look at a lot of different tools, many of them required to buy a license and/or didn't allow students to use it.

ii. Section Random Tester:

In order to make the game flow automatically, it required me to replace all user prompts with predetermined numbers. The way I was able to do this was create a random int class that would return an int between x and y. This became very useful because I could call the same function over and over again with multiple different inputs depending on the need at the time. I also had to add in additional loops to make sure the RNG didn't truly mess anything up (including undiscovered bugs). Overall, writing the game to play itself helped with coverage immensely. I no longer had to try to error handle users entering in bad input purposely trying to break the program.

iii. Section Bugs:

The IDE I was using had a built-in debugger which made debugging very easy. It let me insert break points and would show me relevant data and values line by line, which immensely increased how fast I could find and fix the bugs.