CS 1632 - DELIVERABLE 2: Unit Testing and Code Coverage

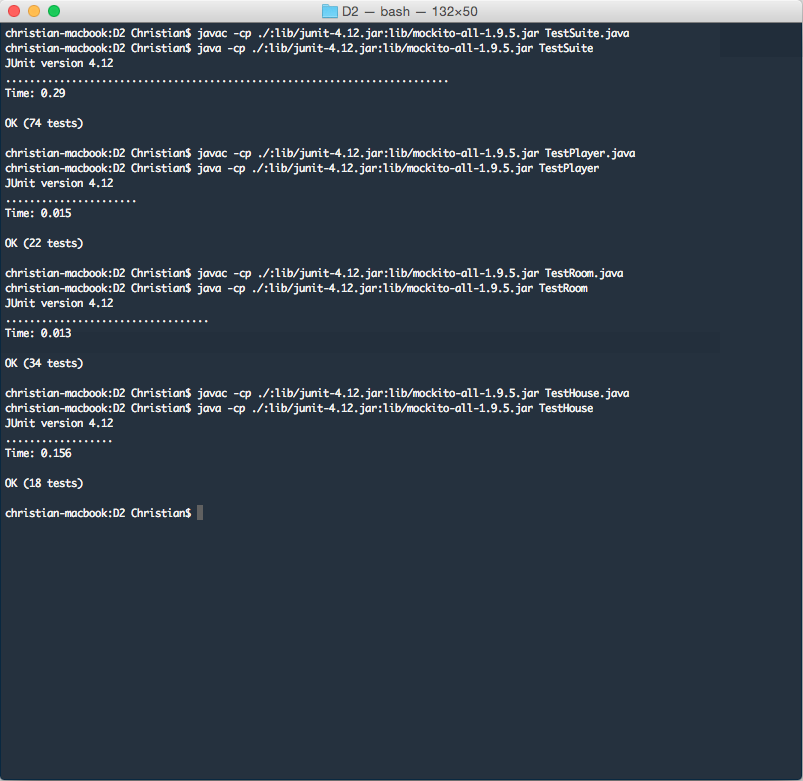
Members: Christian Boni

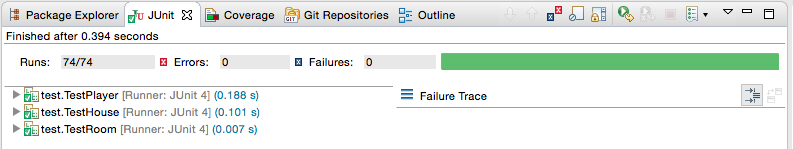
Project: Coffee Maker Quest

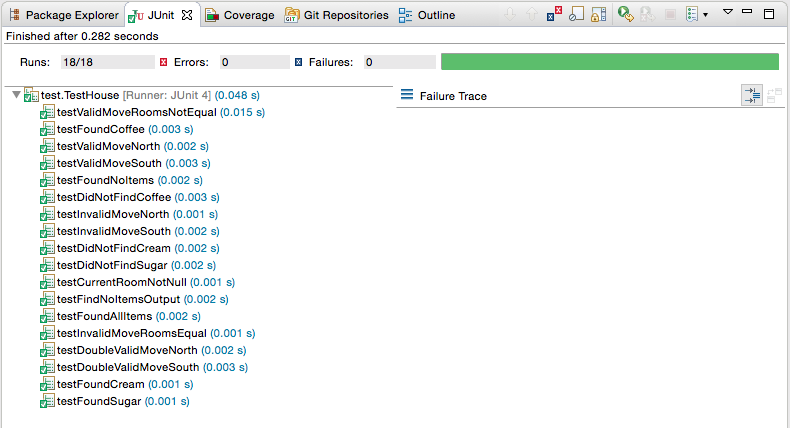
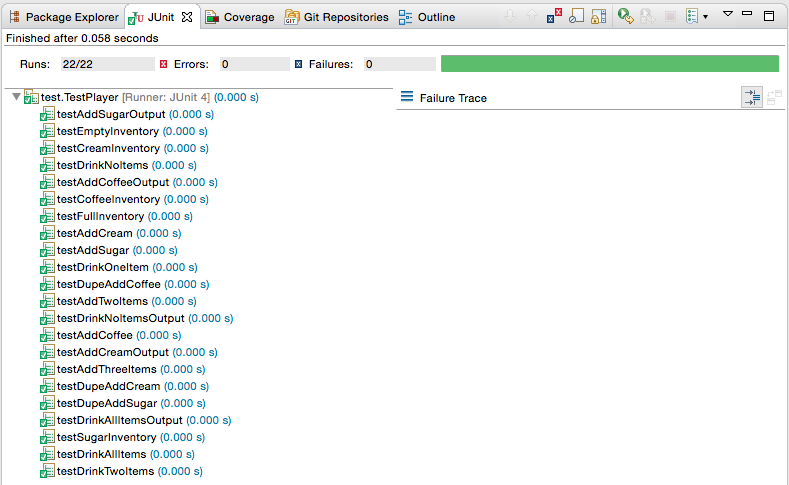
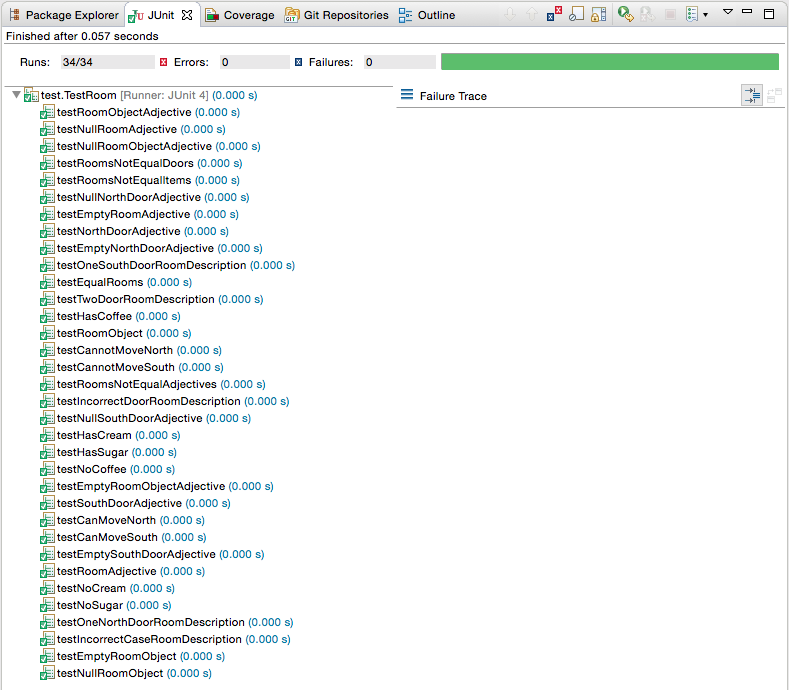
**Conclusions**

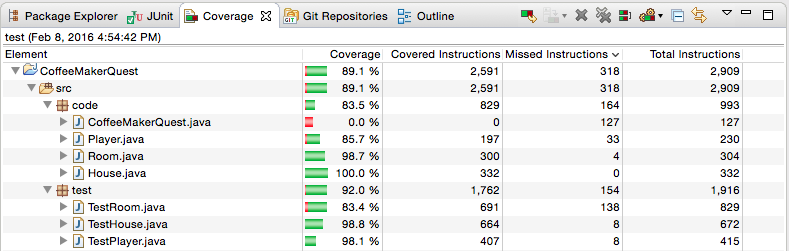
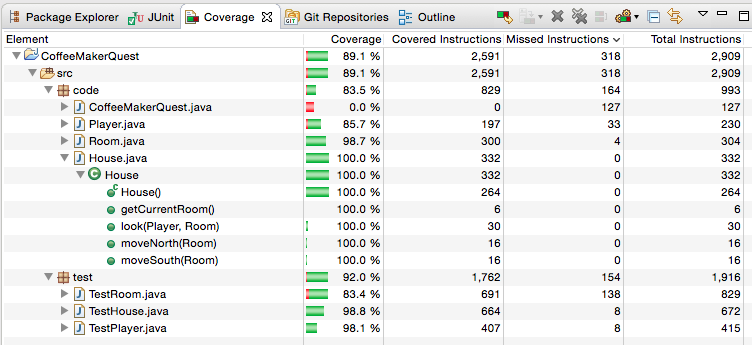
The most difficult aspect of this entire project for me was learning how to write easily testable code. I was given specific requirements and even a working version of Coffee Maker Quest from the last project. While having all of this information for reference greatly assisted in understanding what the code should be doing, it gave no insight on how the game should actually be coded. There are probably thousands if not millions of ways to code a game as simple as Coffee Maker Quest. The problem is in this set of millions of implementations there is a much smaller subset which includes good, easily testable code. As a matter of fact, I had to re-write large portions of the game in the midst of coding it because I realized I was originally writing code that could not be easily tested. It was a little frustrating at first, since previous class projects have taught me that once the functionality is complete the project is complete no matter how badly the resulting code is structured. However, after refactoring the code I began to come across inefficient and careless design choices that I made. Many of which were complicating my code resulting in me having a harder time trying to write effective unit tests for it. Throughout the refactoring process, one of the major things that I realized about writing easily testable code is that all the code should be modularized. It is extremely important to segment the code and to minimize external dependencies within methods and functions. Once I finished up on refactoring the code for Coffee Maker Quest to the best of my abilities, I moved onto writing the jUnit test cases for the game. I was amazed how easy it was to write the jUnit tests after refactoring the code. I simply would look at each public method and think about its expected behavior, typical unexpected behavior, boundary values, and equivalence classes. Lastly, writing jUnit tests for several methods brought my attention to specific edge cases and corner cases that I did not think about when writing the code in the first place. This allowed me to go back and update the code in the game to handle these specific cases.

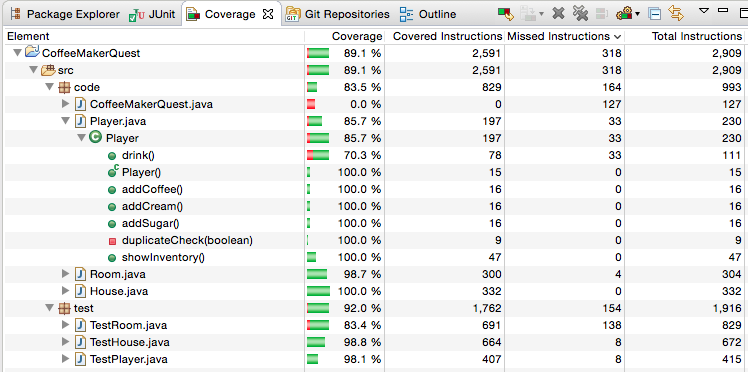
**Unit Tests  
  
All jUnit Test Output from Terminal:**



**All jUnit Test Output from Eclipse:**

**TestHouse jUnit Test Output from Eclipse:** **TestPlayer jUnit Test Output from Eclipse:** **TestRoom jUnit Test Output from Eclipse:**

**Code Coverage  
  
Code Coverage Output for All** **Code Coverage Output for House**

**Code Coverage Output for Player** **Code Coverage Output for Room**