Test 2

Group 2 - Vikingz

Damian Boruch

Tommy Burholt

Elliott Bryce

James Butterfield

Ren Herring

Zhenggao Zhang

Sharlotte Koren

Testing

Testing Methods

As our main method of software testing we decided to go with unit tests for multiple reasons. The main reasons being isolation, automation and granularity. Isolation meant we were able to test out functions independently of other parts of the game and focus on the current function's behavior. Automation meant once we wrote the tests once, we were able to execute them as many times as we liked and tailor our code to suit the test, and granularity, which allowed us to go through our code with a fine toothed comb and test the smallest part of the software, meaning that no errors would go unnoticed. We are also planning on writing all of our unit tests in the headless environment, alongside with 'Mockito', as it will allow us to simulate most parts of the game without running a physical window, giving us the benefit of being able to test non graphical components with ease and without the hassle of having to run the game over again.

We found testing to be particularly valuable as it allowed us to catch tiny bugs that may have never appeared during actual game play due to the niche conditions that had to arise for the bug to have been detectable, and yet we managed to fix them with unit testing, facilitating an almost TDD (Test Driven Development) approach. We set of attempting to write a test for each function of each class in the entire project, this would be perfect as this way we could also check the correctness of our functional requirements however we quickly realized that this may not be feasible due to the amount of code, and at times pointless due to code duplication meant that we would be writing essentially the same tests over again. There also exist classes that are used as static data stores, such as the "York.java" class which only exists to store the map data, and therefore do not contain actual logic that can be tested as such.

Moving on, this brings us to the biggest caveat of unit testing which is that it's time consuming. Achieving a large coverage of the code is very recourse expensive, especially given the large code base we have amassed, in the sense that at times the time spent writing tests to discover bugs, would have been better spent reading over the code and fixing it directly. To keep things organised, we decided to keep the same package structure as the core project with the exception that the in the headless tests each file will have the suffix test, i.e. 'Timer.java' becomes 'TimeTest.java'.

Test Results

Unit Testing Observations

Throughout the testing process, we aimed to test every function to achieve the largest coverage possible, which meant going through the entire code base and through this we came across a lot of redundant code that we could simply erase to remove clutter. This not only made the code more legible, but also improved our unit test creation efficiency due to there being less unnecessary code to test. During this process we also realised that even though we were covering each function, we weren't always hitting all of the edge cases of the function itself. For example when testing an ‘equals’ function, we were originally only testing whether an object A is equal to object B, and never what happens if object A isn't equal to object B, and what happens when either of the objects is null, after realising this, we amended our methods to test for all edge cases possible too, which increased our overall output coverage for possible code branches.

For example, whilst developing the leaderboard it occurred to us that we will need a tuple type to contain the values of a leaderboard entry. The main two uses for this type would be to return entries and compare entries to calculate which scores were in the top five of the leaderboard. However, after we implemented the sorting logic, it only occurred to us that the tuple “equals” method isn't working as intended after we had conducted various unit tests, which then led us to fixing the equals method for this type, as aforementioned.

As we began creating unit tests for the building classes we realised that since most of the functionality for the buildings was contained in the ‘Building’ class and not the actual building classes, meant that if we were to create a separate unit tests file for each building we would create a lot of duplicate tests, for no real benefit. Instead we went for a more joint approach of testing all of the classes that inherit from the ‘Building’ class such as the ‘Accomodation’ class etc. directly from the ‘BuildingsTest’ class. This not only removed unnecessary duplicate unit tests,which lessened unit test compile overhead, but also allowed us to keep the building logic testing together for easier management. It also allowed us to keep the same test coverage for a fraction of the code.

As we continued to write test cases, we quickly realized that it will indeed be quite difficult to reach a large coverage due to multiple factors. The main one being, after we exhausted all of our simpler standalone classes, where the unit tests could be atomic, and not be reliant on any other parts of the codebase, we started to come across pieces of code that was so heavily entangled with other pieces of code that to test a singular function, we would first need to simulate half of the game to get to a point where enough data has been initialised to run that function. This not only made writing the test much more difficult, but also time consuming, and frankly unfeasibly at this scale. This process has taught us a lot about the importance of well structured code, and the ability to write code with the anticipation of having to run a single function without having to initialise half of the codebase beforehand. To further tackle this dilemma we began relying on Mockito to ‘mock’ certain components of the game before we could write a unit test, but even then it was causing us some issues.

For instance here, despite running the application in headless mode and using mockito to mock the sprite batch used for the leaderboard screen, we were unable to get it to compile. This doesn't necessarily mean that our code was failing, as the tests weren't not passing, but simply not running at all. After having encountered the same error on multiple occasions, and having spent a large amount of time trying to fix it, we decided that time would be better spent focusing on writing and perfecting other tests that didn't have this requirement of having to run as a part of the game, but rather as standalone components.

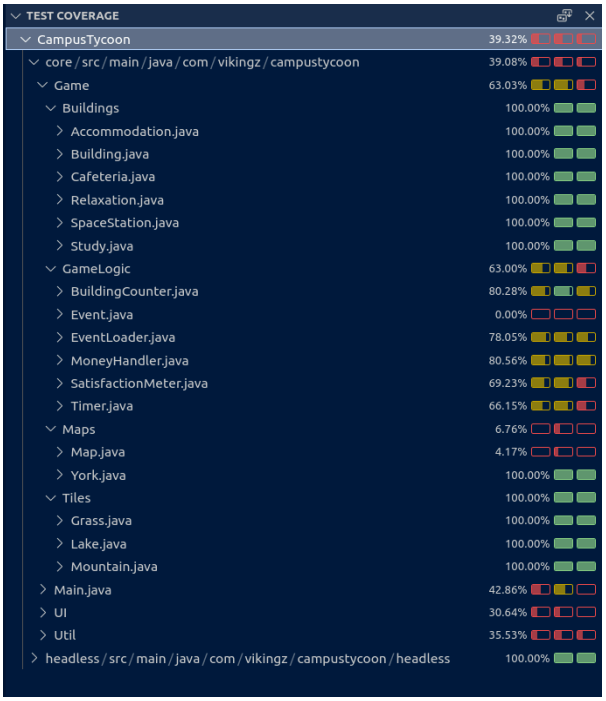
Test Statistics and Outcomes

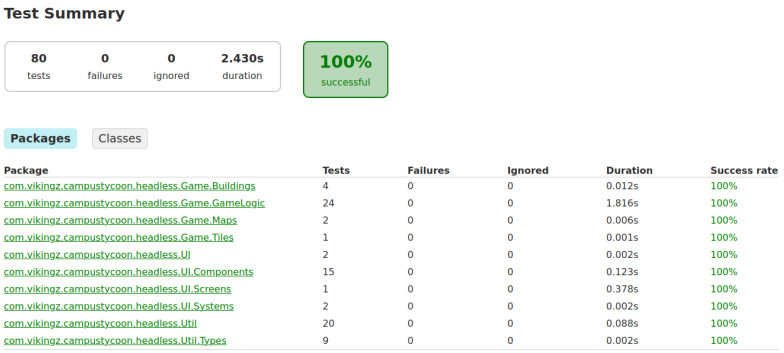
In the end we were able to achieve about 40% coverage of the code base in unit tests. This is a little less than we originally anticipated, however it still brought alot of improvements to our code overall. For all the code that we were able to write unit tests for, i.e. tests that would compile properly, we created 80 unit tests, all of which passed giving us a 100% success rate. This gave us confidence that our code is working as expected, as well as the fact that we can be sure that if someone were to edit our code and break it, our unit tests would lead us straight to the issue.

As for the parts of the code that we attempted to create tests for, but were unable due to the tests not compiling, we learnt that the reason they were not compiling is since we were testing from a headless configuration, a lot of the code was not getting the correct rendering dependencies due to the fact that there was no window present. This was particularly the case in all of the screen classes. After we noticed this, we attempted to go back and change our code so that the update game logic would not be dependant on the rendering logic, however due to the technical debt amassed, and the fact that out render and update logic were so interlaced, the idea to go back and separate out those 2 components was unfeasible. To decouple the code in this way would mean to go back and redesign our top level architecture, which would have been impossible at this stage of the project. If we were to take on this project again, we would have kept this in mind from the very beginning.

headless

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.vikingz.campustycoon.Util		24%		26%	122	173	378	526	61	95	6	14
com.vikingz.campustycoon.UI.Screens		1%		0%	77	82	283	291	58	63	3	7
com.vikingz.campustycoon.UI.Components		50%		10%	114	190	257	504	92	167	4	23
com.vikingz.campustycoon.Game.GameLogic		51%		42%	51	104	126	267	25	62	1	6
com.vikingz.campustycoon.UI		10%		0%	34	39	93	111	17	22	0	2
com.vikingz.campustycoon.Game.Maps		5%		0%	11	13	51	54	4	6	0	2
com.vikingz.campustycoon.UI.Systems		19%		10%	13	18	32	42	8	13	1	3
com.vikingz.campustycoon		49%		0%	3	5	7	14	1	3	0	1
com.vikingz.campustycoon.Game.Buildings		100%		n/a	0	23	0	78	0	23	0	6
com.vikingz.campustycoon.Util.Types		100%		95%	1	19	0	37	0	9	0	3
com.vikingz.campustycoon.headless		100%		n/a	0	4	0	7	0	4	0	1
com.vikingz.campustycoon.Game.Tiles		100%		n/a	0	6	0	6	0	6	0	3
Total	5,122 of 7,957	35%	275 of 367	25%	426	676	1,227	1,937	266	473	15	71





The full jacoco | gradle report as well as a more detailed coverage report and various tests screenshots can be found on our website under the code section:

Testing Screenshots:
https://eng1-vikingz.github.io/campustycoon-website/testingPage.html

Jacoco Report:
https://eng1-vikingz.github.io/campustycoon-website/jacoco-report/html/index.htm

Gradle Report:
https://eng1-vikingz.github.io/campustycoon-website/gradle-report/test/index.html