**QA Report**

In our quality assurance report, we want to target 3 main things: the efficacy of our code, number of logging statements used in comparison to the total lines of code and three metrics we have chosen to analyze our code.

These 3 main things will help us understand our code better, thus making it easier to improve the project in the future. The total lines of code will also help us visualize how we’ve progressed through the analyzed points in time.

The *code coverage* will help us determine what parts of our code has and hasn’t been tested yet. The main components we have chosen to test in this part is our *ProtocolExecutor.class* and *SBListener.class.* The *ProtocolExecutor* executes all commands from the client. For example, sending a message to all players and other such similar client/server communications. Testing the logging process of each client, testing the card sizes at the beginning of each game, testing that the cards are being dealt correctly at the beginning is all a focus of our tests for this class. We did not wish to focus on testing the functionality of the game, but more so the communcation between server and client.

The *number of logging* statements will show us how well we’ve used the logger and how often. This is important to see as logging statements help a lot for debugging at every stage, and thus shows us how well logged our program is and how it facilitates easy debugging. The library we used for logging is called Apache Log4j2.

The *3 metrics* we have chosen to analyse here are: lines of code per method, lines of code per class and javadoc lines per class.

The *line of code per method* will help us analyze which of our methods are the longest and thus have taken the most effort. This can help us find methods that can be simplified, as long methods are not great for readability.

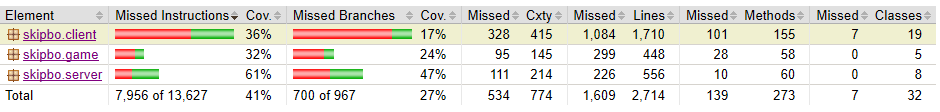
*Lines of code per class* will have a similar function as for LOC per method, but will show us more big picture statistics about our whole program, and not just individual methods.

The *javadoc lines per class* will help us analyze which of our classes has the most descriptions. Very clear and easy to understand classes and methods do not need long javadoc descriptions and thus, by extension, a lot of lines of javadoc will mean that the corresponding class is a more complicated one.

The *total lines of code* metric will show us which classes we have invested the most work in and will also show the average length of classes. This is very useful to have an oversight of all classes and to see if there are classes where things can be shortened/branched off. It is a good way to see how well we’ve stuck to our project plan and if the parts of code were expanded around the time they were supposed to be expanded at.

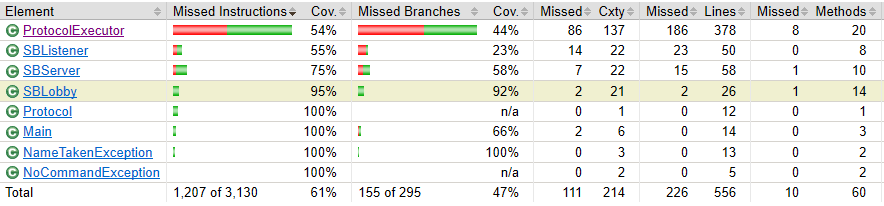
***Results and Discussion***

***Code Coverage***

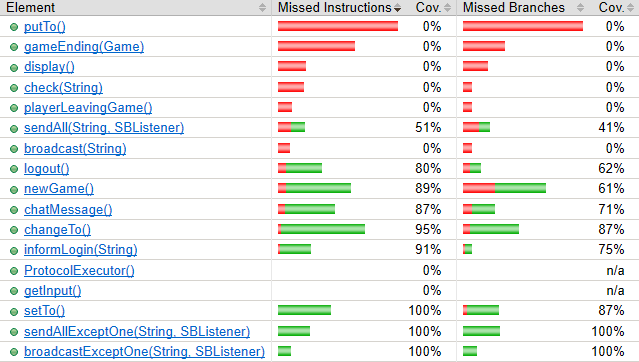


As is visible, from the coverage percentages above, the server is the package that’s tested the most. This is because our *ProtocolExecutorTest* focuses heavily on the *ProtocolExecutor.class*, which in turn tests some parts of the game and a lot of the client/game communication. The 32% of *skipbo.game* package comes from the fact that the *ProtocolExecutorTest* checks that the cards are created at the beginning of the game, but does not check whether further game and card operations are correct.

Looking further into the server package, we can see the following test coverage results:



Even though the *ProtocolExecutorTest* is designed for testing our *Executor*, it only covers 54% of the class, as the Executor has many other functions apart from the tests we designed. Looking further into the tested methods of *ProtocolExecutor,* it becomes clear where the focus of our tests lay. The focus was on testing the client/server functionality – namely, logging in, sending messages and changing name. These things are covered correctly, as shown in the statistics below.



As is clearly visible, a lot of the game functionality, like the method *putTo()* and *gameEnding(),*  are not tested at all. The focus was on methods like *chatMessage(), changeTo() and setTo()*, which are covered nearly completely. The tests were all succesful, and we concluded that our server/client communication was working flawlessly.

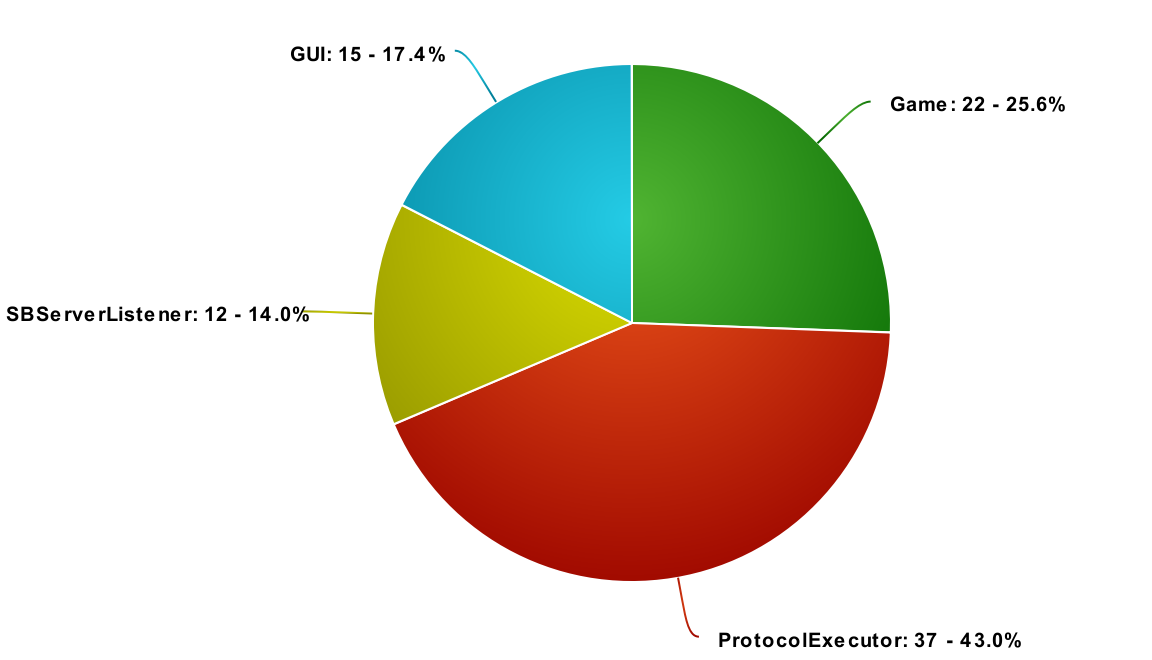
The *ProtocolExecutorTest* was constructed more to be an integration test than a unit test. This is also a reason why such large coverage percentages are seen across our whole program. [...]

The *SBListenerTest* does not feature heavily in the code coverage, as we wrote an additional *testSBL* class to simulate the SBListener, and as such, large parts of our program are not called upon to pass the test. The *SBListenerTest* is written as an actual unit test, unlike the *ProtocolExecutorTest.*

***Logging Statements***

Logging statements were an essential part of our program, as there are many chained calls in our game. In case of failure or an exception, it is quite tough to realize where the mistake lies in such chained programs, so the *Log4j2* library was a massive help. It is easy to follow the structure of executions by the timestamps and get to the root of any exception or failure. As of the **25th** of April, we had ***66*** logging statements in our *game* and *server* packages, and ***37*** in our *client* package. In total, this amounts to ***103*** logging statements across our entire program.

As of the ***12th*** of May, only a few more logging statements had been added, making the tally ***111*** logging statements across the entire programs.



servLog (25.) – 66   
Client – 37

total: 103

today: clLog 41 (fatal: 3, debug: 19 ) , sLog 68 (f: 2, debug: 44)

***Metrics***

The data points that we have chosen are from the 5th of April, the 20th of April and the 26th of April. The 5th marked the end of MS3, the 20th was right after most of our GUI was completely implemented, and the 26th was the day before the MS4 deadline.

***Discussion code coverage:***

***Protocolexecutor doesnt test game side of things, and as such, only has 50% coverage and not a large part. Big coverage because of integration tests and not unit tests for PET, but SBLTest is a unit test. testSBL is inside SBL, and thus not covered by SBLTest as these project files don’t actually get called up.***