QA Report (Skip-Bro: Group 15)

In our quality assurance report, we want to target 3 main things: the efficiency 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 program.

These 3 main focus points 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 login and logout process, chat message forwarding and additionally, checking that a new game is created correctly are all important parts of this test. We did not wish to focus on testing the functionality of the game itself, but more so the communication 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 *lines 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

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
skipbo.client		36%		17%	328	415	1,084	1,710	101	155	7	19
skipbo.game		32%		24%	95	145	299	448	28	58	0	5
skipbo.server		61%		47%	111	214	226	556	10	60	0	8
Total	7,956 of 13,627	41%	700 of 967	27%	534	774	1,609	2,714	139	273	7	32

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:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
ProtocolExecutor		54%		44%	86	137	186	378	8	20
SBListener		55%		23%	14	22	23	50	0	8
SBServer		75%		58%	7	22	15	58	1	10
SBLobby		95%		92%	2	21	2	26	1	14
Protocol		100%		n/a	0	1	0	12	0	1
Main		100%		66%	2	6	0	14	0	3
NameTakenException		100%		100%	0	3	0	13	0	2
NoCommandException		100%		n/a	0	2	0	5	0	2
Total	1,207 of 3,130	61%	155 of 295	47%	111	214	226	556	10	60

Even though the *ProtocolExecutorTest* is designed for testing our *ProtocolExecutor*, 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.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
putTo()		0%		0%
gameEnding(Game)		0%		0%
display()		0%		0%
check(String)		0%		0%
playerLeavingGame()		0%		0%
sendAll(String, SBListener)		51%		41%
broadcast(String)		0%		0%
logout()		80%		62%
newGame()		89%		61%
chatMessage()		87%		71%
changeTo()		95%		87%
informLogin(String)		91%		75%
ProtocolExecutor()		0%		n/a
getInput()		0%		n/a
setTo()		100%		87%
sendAllExceptOne(String, SBListener)		100%		100%
broadcastExceptOne(String, SBListener)		100%		100%

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.

Though that was not the task, strictly speaking, the ProtocolExecutor was designed more as an integration test than a unit-test. Since we were not working with interfaces and inheritance allowing us to abstract the tasks performed by the *ProtocolExecutor* class from the rest of the program, we had to set up an entire server-client framework for each test method. That is why the code coverage resulting from that single test is so high and pervasive. That does not mean that every covered class is thoroughly tested, though, which is only the case for the actual *ProtocolExecutor* methods.

The *SBListenerTest*, in contrast, is way more selective and really a unit-test. A new class, *testingSBL*, was implemented as an inner class in the test class itself. It is an exact copy of the *SBListener* class, except for some additional class fields and the *ProtocolExecutor* method calls being replaced by the assigning of a *String* to a class field. That way, the testing class mimics the original class exactly, but without involving any other classes, allowing us to test its implementation without needing any external framework.

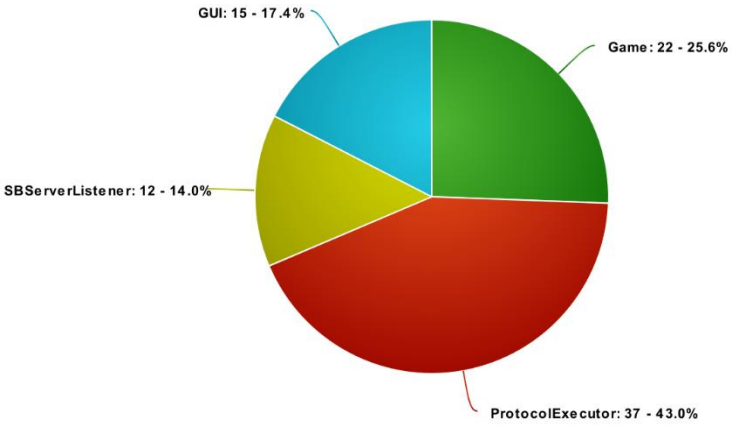
However, since the actual SBListener class is not run in the SBListenerTest, the thoroughness of this test is not reflected in the code coverage metric. Thus, we had to make sure ourselves that the whole class is traversed by the test.

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. Timestamps are also very useful to compare logging statements across server and client.

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. Whenever a bug was found during the gameplay and testing process, we added logging statements to analyze and fix the bug, but removed such statements after the bug



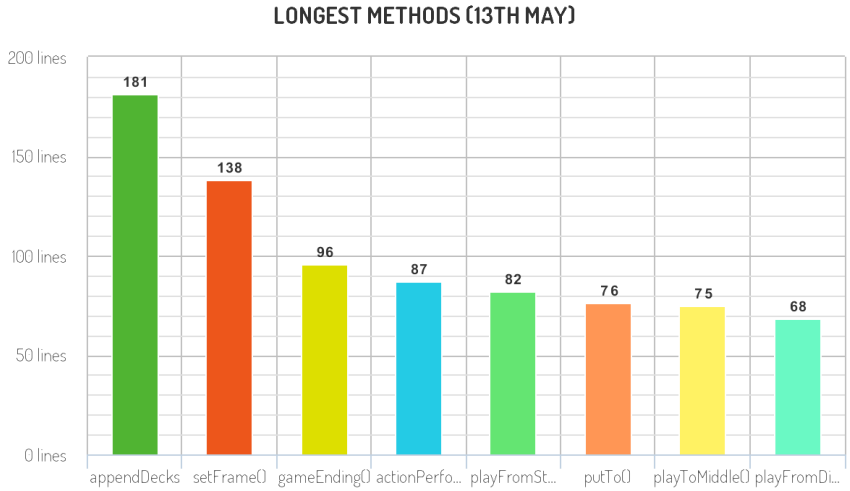
was fixed. The logging statements that are present in our final version are more general ones not targeted at fixing specific bugs, but those that facilitate making the program and game structure easy to follow. Our logging statements are also written to varying levels. For example, a server not being able to be hosted is classified at the fatal level.

Metrics

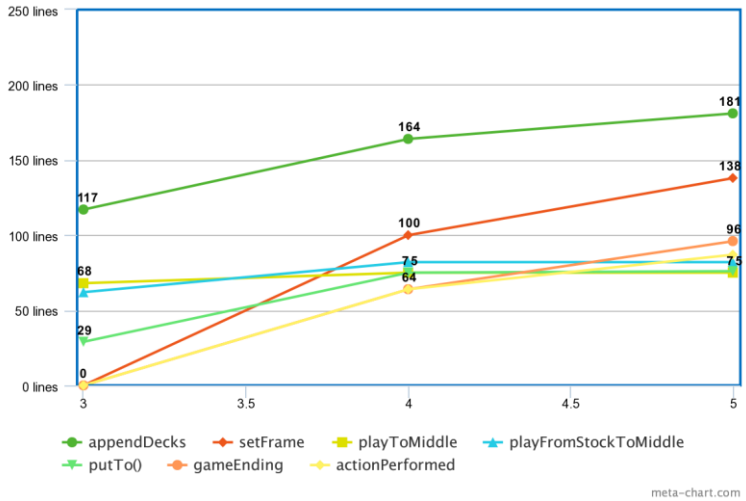
Lines of code per method **-** Our total lines of code inside of methods amounts to: ***4373*** *lines of code.* Our total lines without comments is ***3677.*** The average lines per method across our whole program is **16.63**. This can be explained easily, as our program contains many *get()* methods that have only one to two lines of code, and bring down the average significantly.

The longest method in the code is *appendDecks()* in the class *GameGraphic*. With *181 lines* in the method, it creates all components in the graphical representation of the game. The method creates the default layout from which the game can be played. Below is a chart of the top 10 longest methods in our program.

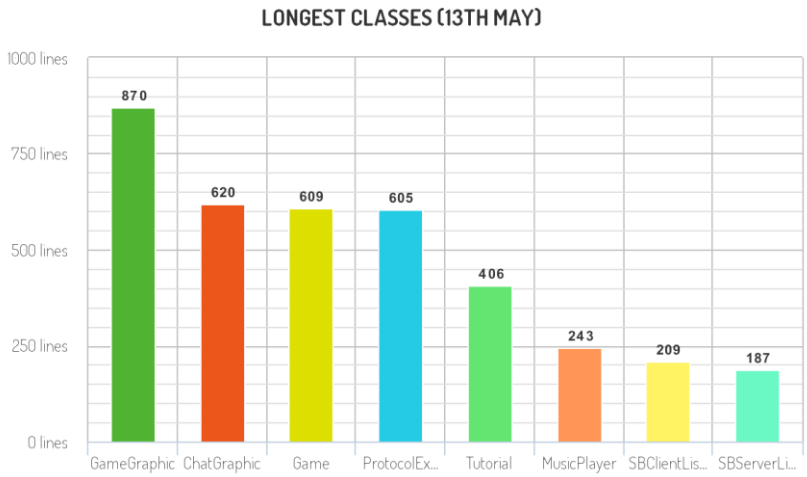
Following is a graph of these methods across Milestone 3, 4 and 5.



The method *set GameGraphic()* was refactored to be *appendDecks()* after MS3. The *playTo()* methods are near the top in MS3, as our GUI was not yet ready at that point. In the interim, GUI specific methods, like *setFrame(), actionPerformed(), and appendDecks()* have overtaken methods like *playToMiddle().* The average was 13 at MS3, 15.84 at MS4 and 16.6 at MS5. Our methods have gotten longer, which also makes sense, as our programs and GUI have gotten more and complicated with time.

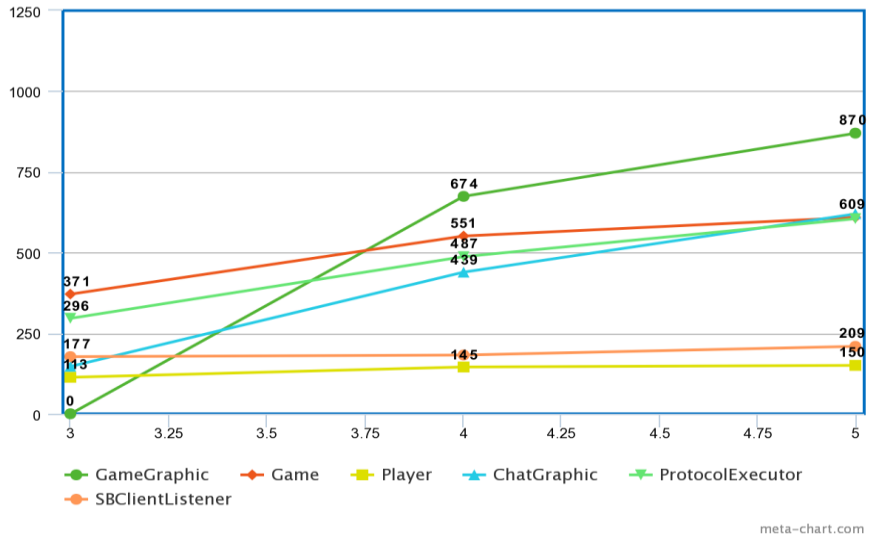


Lines of code per class - The total lines of code in our program (excluding tests) amounts to ***4899*** as of the *13th of May.* The longest class is *GameGraphic (870 lines)* which represents the game GUI. As it has many components to define and arrange on the game layout, it is understandable why this class is the longest in our program. It manages the default layout and manages all card movement and player movement.



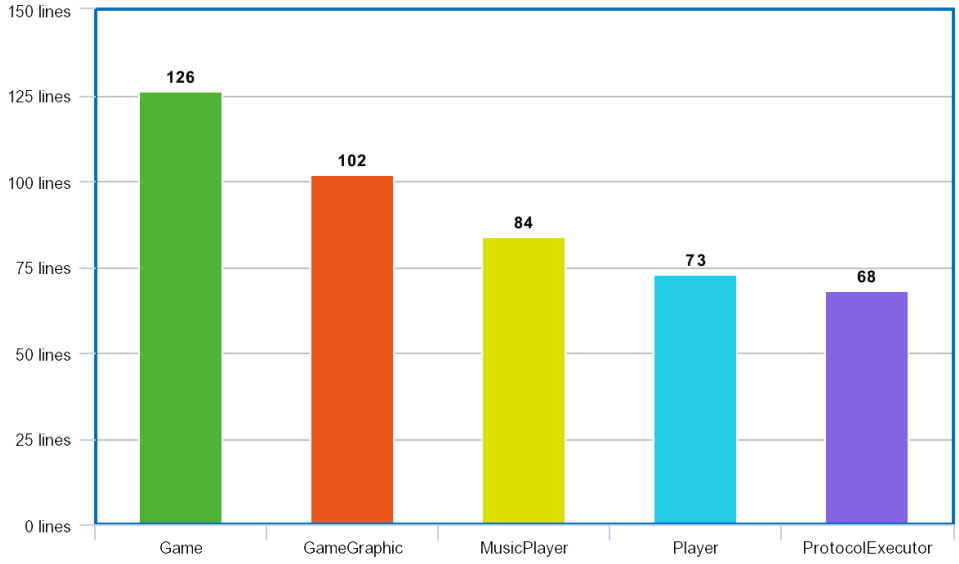
The second longest class is also on the GUI side of things – the class *ChatGraphic*. This class is responsible for all GUI elements of the chat, and all forwarding of messages from the server to the client. As this is another essential part of the program, it has multiple applications, which cannot be short.

Following is a graph describing the progression of these classes across the last 3 milestones. GUI was developed heavily after MS3, and as such, these classes (GameGraphic, ChatGraphic) show a rise in line numbers. The Game methods take a small halt, as the game was ready in MS3 and did not need to be developed much further after that point.



At MS3, the *total lines of code* was at *1895*, at MS4 the number was at *3614*. At MS5, this was at *4899*. That is an increase of *3000* lines from MS3 to MS5.

Javadoc per class - For complicated classes, javadoc lines are going to be longer, as these need more explanation. The game classes need the most explanation, as none of the game methods are self-explanatory. As such, the *Game* class is the one with the most javadoc descriptions – *126*. As most of the game is described in *Game,* as are all the card operations, it is imperative that this class be correctly described. We chose javadoc as a measurement to see what classes are the ones that have needed extra documentation, so that we can keep track of the complicated things.



Out of *734* lines in *Game, 17.17%* are javadoc descriptions. As *GameGraphic* is the longest class in our program, it also has many javadoc descriptions; *102 lines* in *GameGraphic* are for javadoc. There are also many methods for the representation of the GUI that need proper descriptions. *MusicPlayer* is from the *MP3SPI* library, and as such, will have a lot of javadoc comments to describe the functionality. *Player* is an essential part of the game, with multiple imperative methods, and as such, also needs correct and descriptive comments.

Conclusion

At the end of this report, we can see that most of the results are what we would expect them to be. As the GUI was developed between MS3 and MS5, the classes and methods pertaining to that rise up quickly, while all game classes and methods are more stagnant, as the game was implemented before MS3. Javadoc descriptions shows us the classes that needed the most description, which are also as would be expected. *Game* and *GameGraphic* are the least intuitive of most of the classes, and as such, need exact descriptions to be easily understood.

Report by Group 15: Rohan Girish, Guillaume Joyet, Manuela Wildi, Janni Batsilas.