Vincent Mackenna
CS 472 1001
2 January 2023

Dynamic Analysis Lab

All IntelliJ screenshots and code snippets are at the end of this document.

IntelliJ Coverage Report				
Step	**Test Function Added For**	**Class %**	**Method %**	**Line %**
0	N/A	3% (4/110)	1% (10/624)	1% (28/2274)
1	isAlive()	16% (18/110)	9% (60/624)	8% (190/2306)
2	addPoints()	16% (18/110)	10% (64/624)	8% (196/2306)
3	setAlive()**	16% (18/110)	10% (68/624)	9% (218/2306)
4	registerPlayer()	36% (44/112)	24% (148/610)	23% (556/2322)

NOTE: I know we are not meant to perform testing on standard getter/setter functions; however, setAlive() is a non-standard getter function.

Q1) Are the coverage results from JaCoCo similar to the ones you got from IntelliJ in the last task? Why so or why not?

The coverage results were somewhat similar, but they still differed a notable amount. The respective algorithms which IntelliJ and JaCoCo use to analyze code coverage must be different because the IntelliJ and JaCoCo would often show different coverage percentages for the exact same code snippets.

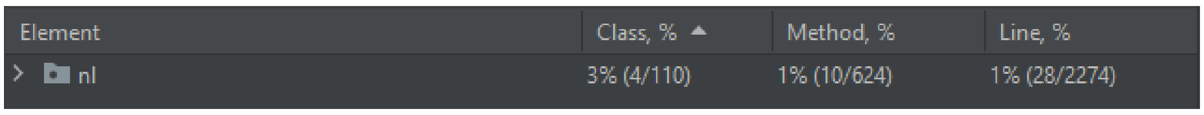
Q2) Did you find the source code visualization from JaCoCo on uncovered branches to be helpful?

I found the source code visualization from JaCoCo to be extremely helpful.

Q3) Which visualization did you prefer and why? IntelliJ's coverage window or JaCoCo's report?

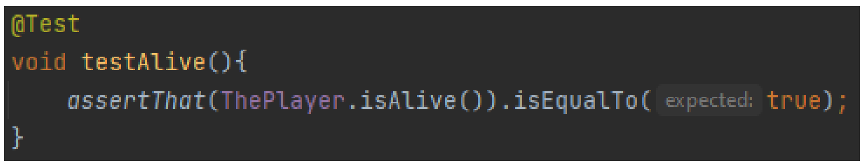
Both tools provide very similar information, but the IntelliJ's coverage window provides that information with better formatting and with more features. The visualizations are sufficient and you can browse the project file hierarchy in the coverage window and click on any file you want to edit so you can make immediate changes inside the app and run the coverage analysis again. This process cannot be done with JaCoCo because it is not directly implemented with the IDE.

Step 0) Results before changes.

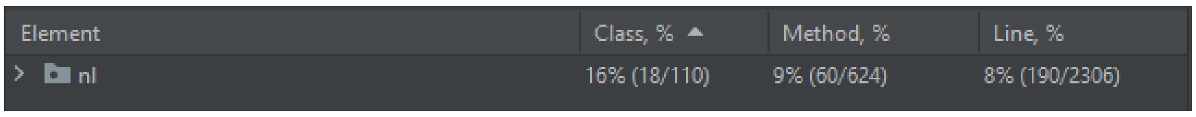


Element	Class, %	Method, %	Line, %
nl	3% (4/110)	1% (10/624)	1% (28/2274)

Step 1) Testing isAlive()

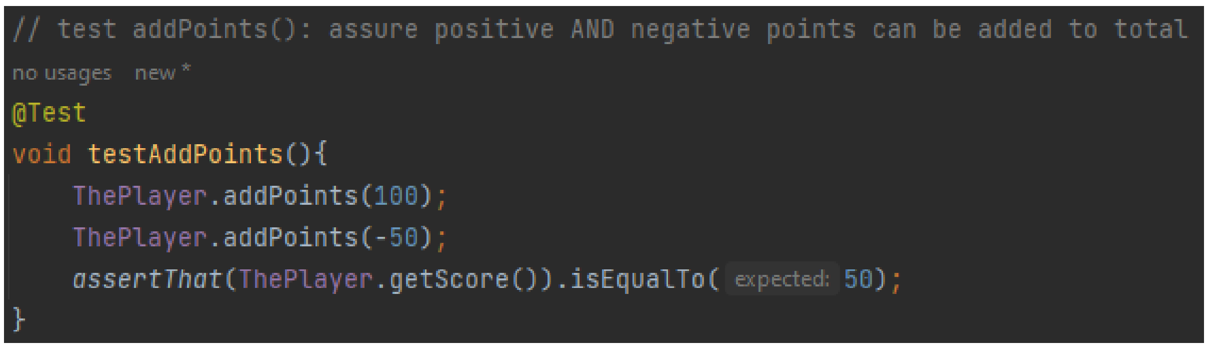


```
@Test
void testAlive(){
    assertThat(ThePlayer.isAlive()).isEqualTo( expected: true);
}
```

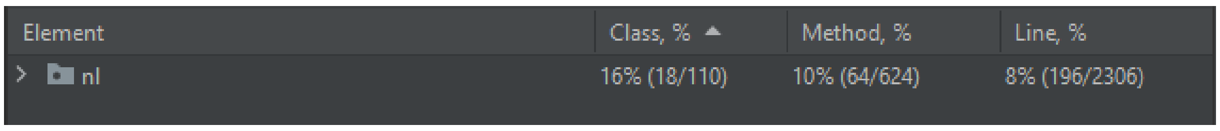


Element	Class, %	Method, %	Line, %
nl	16% (18/110)	9% (60/624)	8% (190/2306)

Step 2) Testing addPoints()

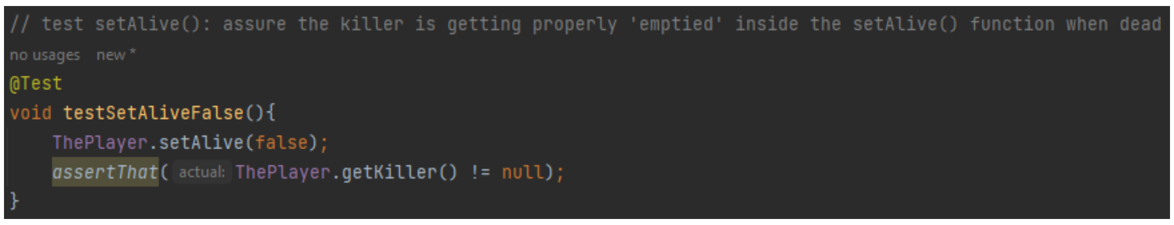


```
// test addPoints(): assure positive AND negative points can be added to total
no usages   new *
@Test
void testAddPoints(){
    ThePlayer.addPoints(100);
    ThePlayer.addPoints(-50);
    assertThat(ThePlayer.getScore()).isEqualTo( expected: 50);
}
```

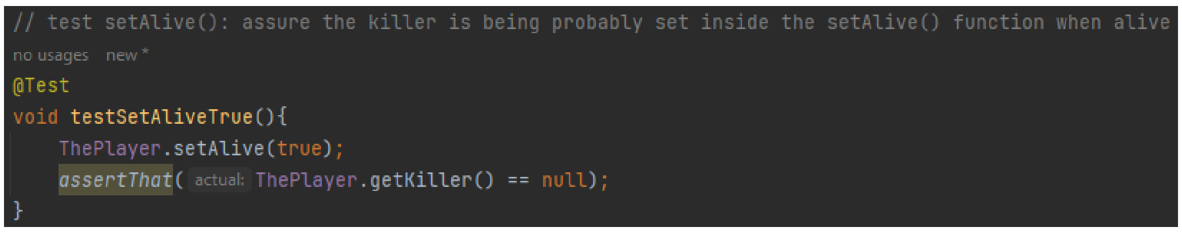


Element	Class, %	Method, %	Line, %
nl	16% (18/110)	10% (64/624)	8% (196/2306)

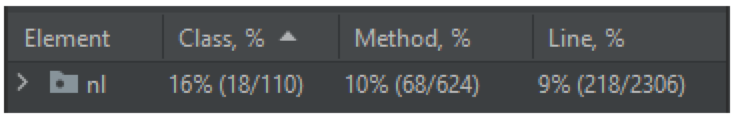
Step 3) Testing setAlive() — non-standard setter function



```
// test setAlive(): assure the killer is getting properly 'emptied' inside the setAlive() function when dead
no usages   new *
@Test
void testSetAliveFalse(){
    ThePlayer.setAlive(false);
    assertThat( actual: ThePlayer.getKiller() != null);
}
```

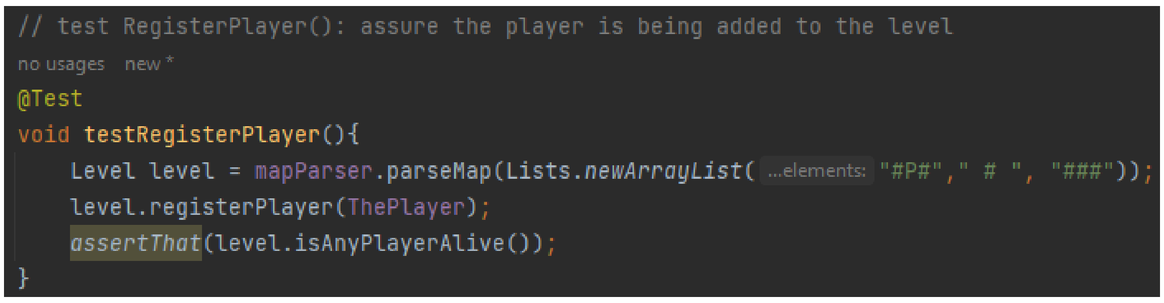


```
// test setAlive(): assure the killer is being probably set inside the setAlive() function when alive
no usages   new *
@Test
void testSetAliveTrue(){
    ThePlayer.setAlive(true);
    assertThat( actual: ThePlayer.getKiller() == null);
}
```

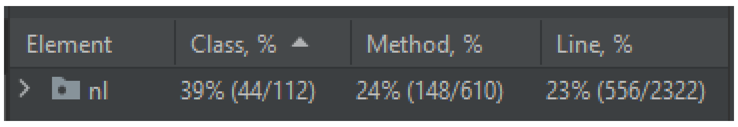


Element	Class, %	Method, %	Line, %
nl	16% (18/110)	10% (68/624)	9% (218/2306)

Step 4) Testing registerPlayer()



```
// test RegisterPlayer(): assure the player is being added to the level
no usages   new *
@Test
void testRegisterPlayer(){
    Level level = mapParser.parseMap(Lists.newArrayList( ...elements: "#P#"," # ", "###"));
    level.registerPlayer(ThePlayer);
    assertThat(level.isAnyPlayerAlive());
}
```



Element	Class, %	Method, %	Line, %
nl	39% (44/112)	24% (148/610)	23% (556/2322)