# JUnit Testing Strategy

## How we are going to identify JUnit test cases

### Organisation

As each member of the group is developing their code, we will also be thinking of tests to match what we are building appropriately. Tests will be made for methods that change or manipulate game data, for example adding gizmo methods, file I/O methods and moving the ball etc. Classes will have corresponding JUnit test classes with an appropriate set of methods to test each method. The test classes will be contained in own package. The test classes will be initialised with *@BeforeClass* and *@Before* annotations and cleaned up with *@AfterClass* and *@After* annotations. We will be using the JUnit methods such as *assertTrue()*, *assertFalse(),* and *assertEquals()* that will validate the method outputs. The test will be set to verify for a range of valid and invalid inputs to ensure that invalid conditions are handled gracefully, and parameter boundaries are never exploited. Therefore, some tests will be designed to test formal specifications, while the other will test upper and lower boundary values for method inputs. Methods that throw exceptions will also be tested accordingly to ensure they are able to catch any thrown exceptions if the methods were to fail.

### Details

A set of tests will be designed that confirm that certain listeners for user inputs that are available in running mode are not available in the building mode and vice versa. The tests will enforce consistency throughput the game modes. Another set of tests will check the position and velocity of ball to verify that changes induced by collisions and forces modify the initial conditions of the ball as expected. Therefore, the interactions of gizmos with ball can be quantified derived from its position and velocity. The tests for building mode will ensure that gizmos can only occupy empty tiles within the board, and no gizmos can possibly overlap. Triggering actions of the gizmos will be tested to ensure that each gizmo can serve as a trigger to multiple other gizmos and all appropriate actions on connected gizmos are invoked. The tests will also ensure that the flipper motion is synchronised with the ball’s motion. The I/O tests will ensure that appropriate exceptions are thrown and the user is notified when an invalid or corrupted file is attempted to be opened.

## Approaching JUnit testing as a group

When approaching JUnit testing for Gizmoball, as a group we decided that the best approach would be for every group member to design tests for the parts of the code that they are developing as a group member testing code that they are fully familiarised with would be the most effective way at ensuring that the code works in the manner that it was intended too. However, this does not mean that members of the group cannot test code that they did not develop themselves. As the project progresses, members will obviously be familiarising themselves with every class, learning the ins and outs of the entire project. We see this as an opportunity to get different perspectives on current JUnit tests – making sure that the current tests return the correct results and test the code appropriately – and giving other members the freedom to test other code further if they see necessary. We believe this approach will give full proof test cases for every class as essentially each test will be checked over by other members, not just the person who wrote the test.

The plan is for each member to write both black box and white box tests for the segments they are testing. Black box tests to make sure we are getting the expected outputs for the given inputs, and white box tests to make sure the logic behind the code that member developed is working as it should.