Unit testing strategy for Gizmoball

During the development of the preliminary release, as a team, we have realized that the code base of this project will steadily increase in both size and complexity as we keep adding new features. Therefore, we agreed to do unit testing as this practice will boost our confidence when either writing new code or editing previously written code. This is especially important since the code base is maintained by multiple developers. As a bonus, this will also help us confidentely experiment with extra features without breaking the basic functionality, in order to achieve bonus marks.

Test cases identification

We will focus on writing unit tests for the domain specific code, that is code that implements "business logic" (in this case, Gizmoball core functionality) as this is very likely to change often.

We are planning to identify test cases mostly based on the `model` side of our `MVC` design, as these classes are responsible to model the physical aspects of the game. Moreover, we decided to write good method specifications for the complex methods, as we can easily create relevant uses cases based on them, including *edge cases*.

We are going to have one `JUnit` test class for each of the `model` classes that we are planning to test. The test classes will be placed in the `test` package which is going to have the same structure as the `src` package, for consistency.

What we'll test

- `equals()` and `hashCode()` methods for `Gizmo` and `GameModel` classes as we're heavily relying on them in hash-based collections, and in writing other unit tests
- `toString()` methods for `Gizmo` and `GameModel` classes, because they will be used in converting the `model` to a text file
- all the methods exposed in the `model` interfaces since they will be used by `views` and `controllers`
- *file saving and loading* functionality
- `Gizmo` uniqueness, deletion and rotation
- the triggering system
- the movement according to time - *location*, *forces*; especially for edge values
- collisions detection
- whether checked exceptions are being thrown accordingly

What we won't test

- library code; we assume that the MIT physics package works according to its specification; the same with other resources when they're available
- code that deals with UI, like `java-fx` toolkit and components; for that, we'll perform *validation testing*
- trivial code, like *getters* and *setters*

Approach

We will use the `JUnit 5` framework for Java as it provides some nice features like *Parametrized tests*, *grouped assertions, powerful exception expecting mechanism etc*. Since we're all contributing to the source code of Gizmoball, we have decided that all of us will be responsible for unit testing as well. However, as individuals, we should try to test the code written by a different team member and not our own as we might be biased towards our own implementation which may cause us to miss some test cases, making it more difficult to find potential bugs.

As a lot of the code is already written, we won't be able to adhere to the `Test Driven Development` practice. However, we plan to write tests incrementally, most likely after each piece of meaningful new code is added. A mix of *black box* and *white box* testing techniques will be useful in finding bugs more effectively. We can start with *black box* test cases against a method's specifications and then switch to *white box* testing in order to get some more insights about the method's behaviour.

Guidelines that we're going to follow

- smart use of `setUp()`, `tearDown()`, `beforeEach()` and `beforeAll()` methods
- use the following naming convention for tests: `MethodName_StateUnderTest_ExpectedBehavior`; for example: `rotate_Square_PositionIsTheSame`
- 100% line coverage is a must, but only run the coverage on what we mark as business critical code
- only one *assertion* per test, unless the `JUnit 5` *grouped assertions* are used (they are placed in a lambda and the framework reports all the assertions that failed) in which case they should be related
- use the overloaded version of the `assert` methods, in order to pass in a message that describes what went wrong
- run the whole test suite frequently, especially when adding a new feature
- set up a Continuous Integration pipeline on *gitlab* to run the test suite whenever we push code to the `master` branch