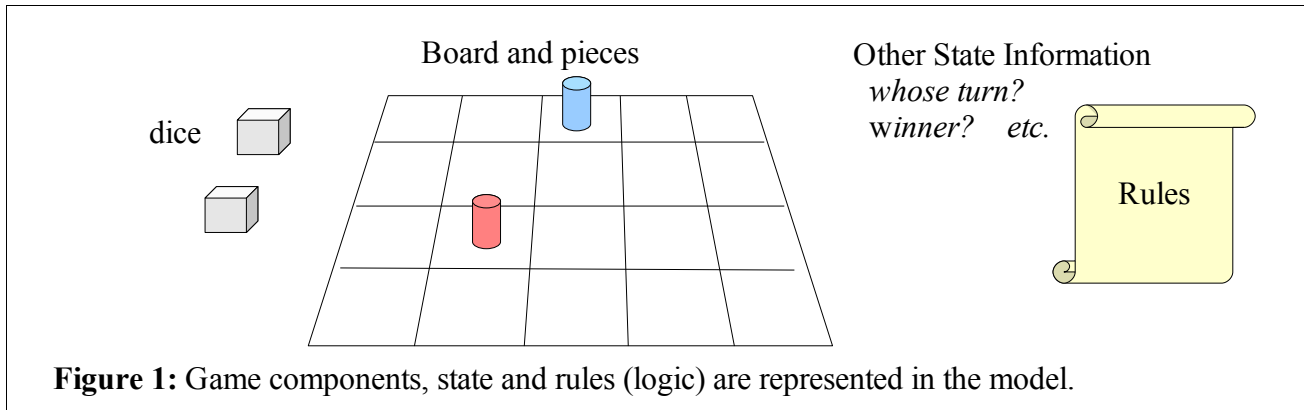


Test Driven Development with an Introduction to JUnit

What is in the model? Consider the basics of what the model represents and what it is supposed to do. We're not talking about a great deal of detail at this point, but an understanding of the concepts from the point of view of a domain expert (as opposed to computer code). For example, our race game model, might be thought of as shown in Figure 1. Just as with computer code, there may be multiple approaches and some fine tuning of concepts through iteration and refactoring.



TDD (Test Driven Development) means tests are written before the actual code that is being tested. But how is this done for code which doesn't exist yet? Just as you run a piece of code and check its output for correctness knowing the expected values, you can write tests that do the same. The tests are written assuming the production code already exists. What should it do? How should it be called? These questions drive the design of the code. Follow this example by adding the new source files to your project. Make sure to make the adds and commits to your repository.

To begin, consider what happens when the application starts. The model should be initialized to a well defined start state (e.g. the initial board configuration). To display the game state accurately for the user, the view should query the model for its state information (positions of players, dice status etc.). Test Driven Development is motivated by what is necessary now (how to get the required game state information from the model) and how to test that it is correct. Tests rely upon comparing information obtained from the model against expected information.

```
package tddExample;
import static org.junit.Assert.*;
import org.junit.Test;

public class RaceGameModelTest {
    public static final int PLAY_A_INIT_POS = 0;
    public static final int PLAY_B_INIT_POS = 14;

    @Test
    public void testInitialPlayerPos() {
        RaceGameModel model = new RaceGameModel();
        int playAPos = model.getPlayerPos(RaceGameModel.PLAYER_A);
        assertEquals("Player A initial pos", PLAY_A_INIT_POS, playAPos);
        int playBPos = model.getPlayerPos(RaceGameModel.PLAYER_B);
        assertEquals("Player B initial pos", PLAY_B_INIT_POS, playBPos);
    }
}
```

Figure 2: A JUnit test case to confirm the initial player positions.

So to make the test pass we are forced to think about how the model maintains the positions of the players. For now we assume that we can use an integer variable for each player. Next, we need to consider how those values are initialized. Finally, we can use those position values from within *getPlayerPos* so that we can return the values that make sense. Figure 5 shows the revised model code and Figure 6 show the “green bar” in the test runner after running it. Note the green circle with white arrow icon can be used to get a pop-up and run recent launch configurations.

Typically after passing the test, the code should be examined for possible refactoring. Refactor and confirm that the tests still work. Now the code can be committed. For harder tasks, you might commit the code even with the “red bar”. Try to assure that at least the code compiles. Make a habit of “cleaning” and running the tests when you update with fresh code from the repository.

Figure 5: Revised model code to actually use player positions

```
package tddExample;
public class RaceGameModel {

    public static final int PLAYER_A = 1;
    public static final int PLAYER_B = 2;

    public static final int PLAYER_A_INIT_POS = 0;
    public static final int PLAYER_B_INIT_POS = 14;

    private int playerApos = PLAYER_A_INIT_POS;
    private int playerBpos = PLAYER_B_INIT_POS;

    public int getPlayerPos(int playerA) {
        if (playerA == PLAYER_A)
            return playerApos;
        else
            return playerBpos;
    }
}
```

Run button

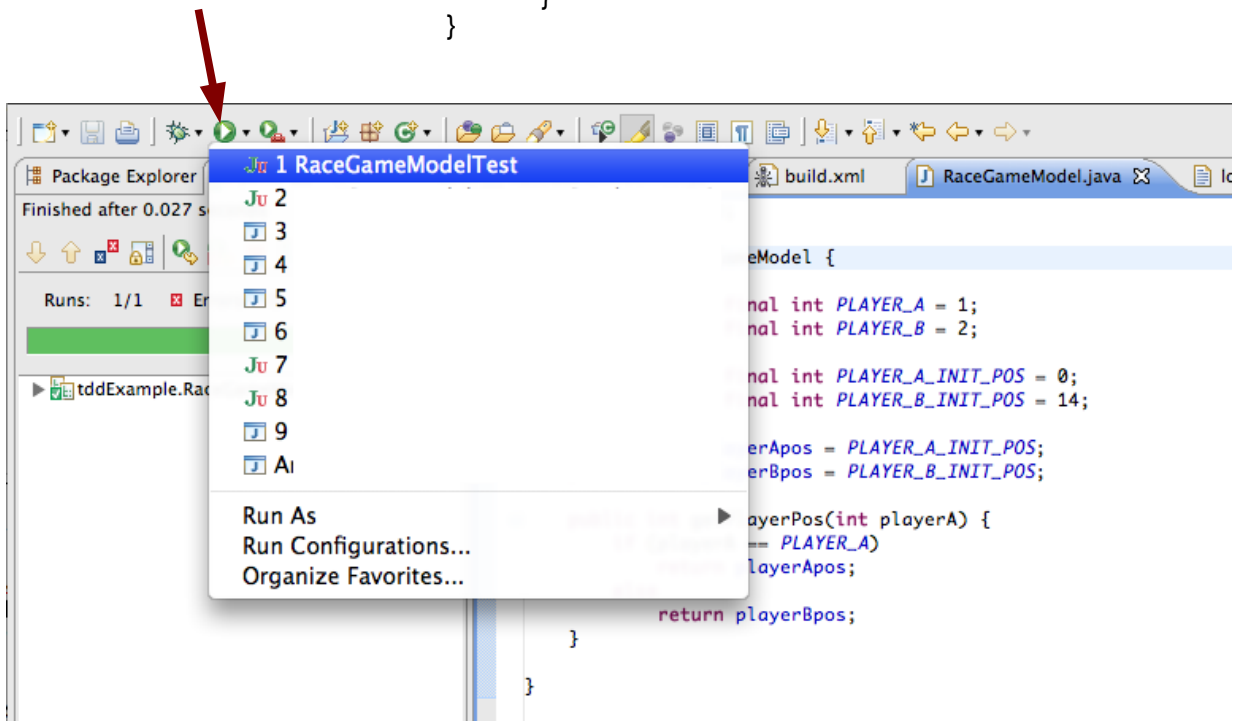
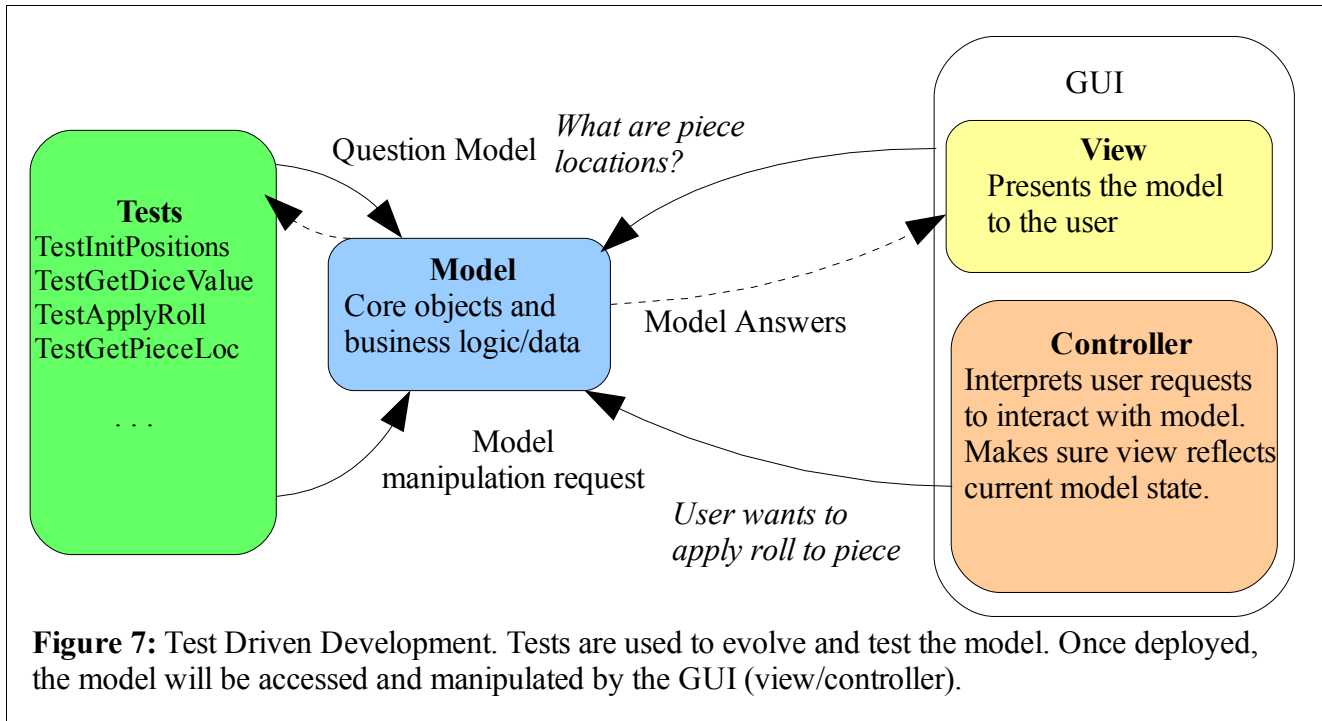


Figure 6: “green bar” appears after running tests on revised model.

In summary, motivated by what your code needs to do next, imagine that code already exists and write a test for it. Create the necessary model code to get your test to compile and run. Generally, the first time a test compiles and runs it should fail. Once the code runs, verify that all the tests pass (“green bar”). If any tests fail, make modifications so all unit tests pass. Before starting on the next feature, consider if your code needs to be refactored. This should be a fairly tight cycle. Methods and tests can be built incrementally. You should make your tests independent of each other and not assume any particular order. Strive for clear meaningful well named tests. Each test should map nicely to a single feature of the code. A test failure should give quick insight into what aspect of the code has problems. Note, that as you build up your tests and model, the GUI is not required. Once deployed, the model routines will be called by the GUI rather than the tests (see figure 7). As development progresses, cycles may also include GUI code especially to get customer feedback on user interface issues. All types of code may grow iteratively and incrementally.



Refactor the solution given in figure 5. Could you come up with a more simple solution that would work for the test case? Make sure to commit your solution (with a comment). Consider other model features and the tests that would be required. Add a few more tests which force the model to grow. (Please wait on implementing anything based upon *random* as we will talk about that later.) Try to get a feel for the effort required to develop code this way – we will be estimating shortly.

Here are some ideas of the possibilities that your code might take to separate model (domain knowledge) from interface code (view and controller). In the original RaceGame, the board was displayed by updating a grid of JLabels – these are Swing components which don’t belong in the model. They are view components and should be updated when the game state changes. Placing the pieces on the board assumes knowledge of the rules (ask the model). Also note that as the game continues the view of the board will need to be updated but it only had to be created once. There is a natural division of view functions here into something like `buildBoard()` and `updateBoard()`. The method `updateBoard()` should ask the model for the player locations. In the original code, this information was found simply by reading the variables `player1` and `player2`. The initial positions could then be based upon how the model sets the initial positions, not upon an assumption made by the viewer. The `actionPerformed()` method contains some horribly mixed up code. Handling events should be primarily controller code but there should be a quick hand-off to the model to pass on user intentions (model knows who's turn it is etc.).