Refactoring to Use MVC

The refactorings that have been applied to *RaceGame.java* so far, have stayed within the same class. The code should now be much easier to understand. However, another smell which remains is due to the class trying to do too much - it needs to keep track of the game state (board, dice, rules, etc.), display the game and process the user input. These different responsibilities can be used as a guide to break the code up into multiple classes which each have a much clearer purpose.

Most importantly, a separate *model* entity can be used to keep track of the game logic, rules, current configuration etc. The user interface and control logic are provided by *view* and *control* components (Figure 1). Often the *view* and *controller* are left in the same class since they are tightly connected. More complex applications may use hierarchies of objects to create the different components. Many frameworks strive to separate business logic (here, the game logic) from the way that it is presented to users. Interweaving model code throughout the interface may make it easy to quickly create something but maintenance, testing and porting suffer greatly.

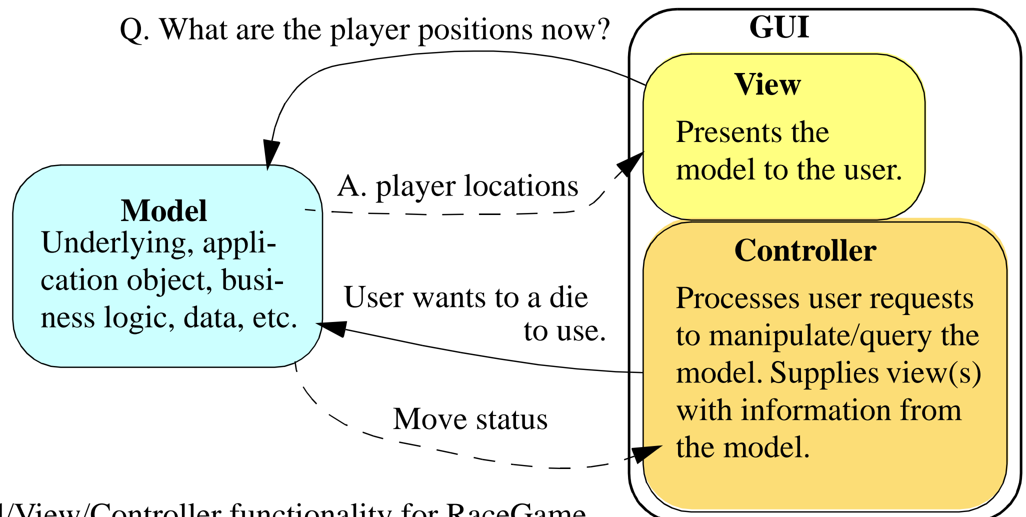

Figure 1: Example Model/View/Controller functionality for RaceGame

Separation of the model and the user interface implies that the model can be developed independently (*a good thing*) if it provides public methods, and perhaps constants, that could allow a coupling with an interface. For instance the model for the Race Game might have methods to initialize the board, process move requests, and report the state of the board locations. A controller would interpret user inputs (user clicked on a button) and translate them to calls to the model (move request). The model, knowing the game logic, can process the move request and update the board if necessary. The model only updates internal data structures so no change appears to the user. The controller tells the view to update itself (*updateGUI*) by asking the model for the latest information (*isLocationEmpty*) and then displaying it (with statements that modify the on screen appearance like *setText* etc.).

To prepare for a refactoring in which the model is separated from the view and controller, markup your code in **Windows Journal**. Use the highlighter tool and indicate which sections belong to which MVC component using the color convention in Figure 1 (model: light blue, view: yellow, controller: orange). Some of your methods may mix MVC components. These are good places to look for further refactorings (esp. mixes involving the model). Refactor to create methods which begin to look almost like those that will appear once the original class is broken up into model and GUI classes. For instance, the view will need to ask the model for information needed to redraw the board with the current information. The model might have a method like *isLocationEmpty*, which is repeatedly called by the view in a routine *updateGUI* which redraws the board. (Since the board only has to be created once, a view method like *buildGUI* may initially define the components). The controller code, mostly in the *actionPerformed* method, interprets the action expressed by the user and sends it to the model (*processDie*) then tells the view to update itself (*updateGUI*).

If the model is developed separate from the interface, it's logic can be developed and tested independent from how it will (might?) be displayed and manipulated by the user. XP, using the practice of Test First or Test Driven Development (TDD) to drive the design of code, can be used to incrementally (re)build the model. The tests promote modular code which does just what has to be done (keeps it simple). Automated tests check for correctness and can be run often to ensure that old code is not broken when changes are made. Tests also serve as a form of documentation showing how the code is to be used.

TDD means tests are written before the actual code 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. Figure 2 shows how this is used in a refactoring to build a separate model class.

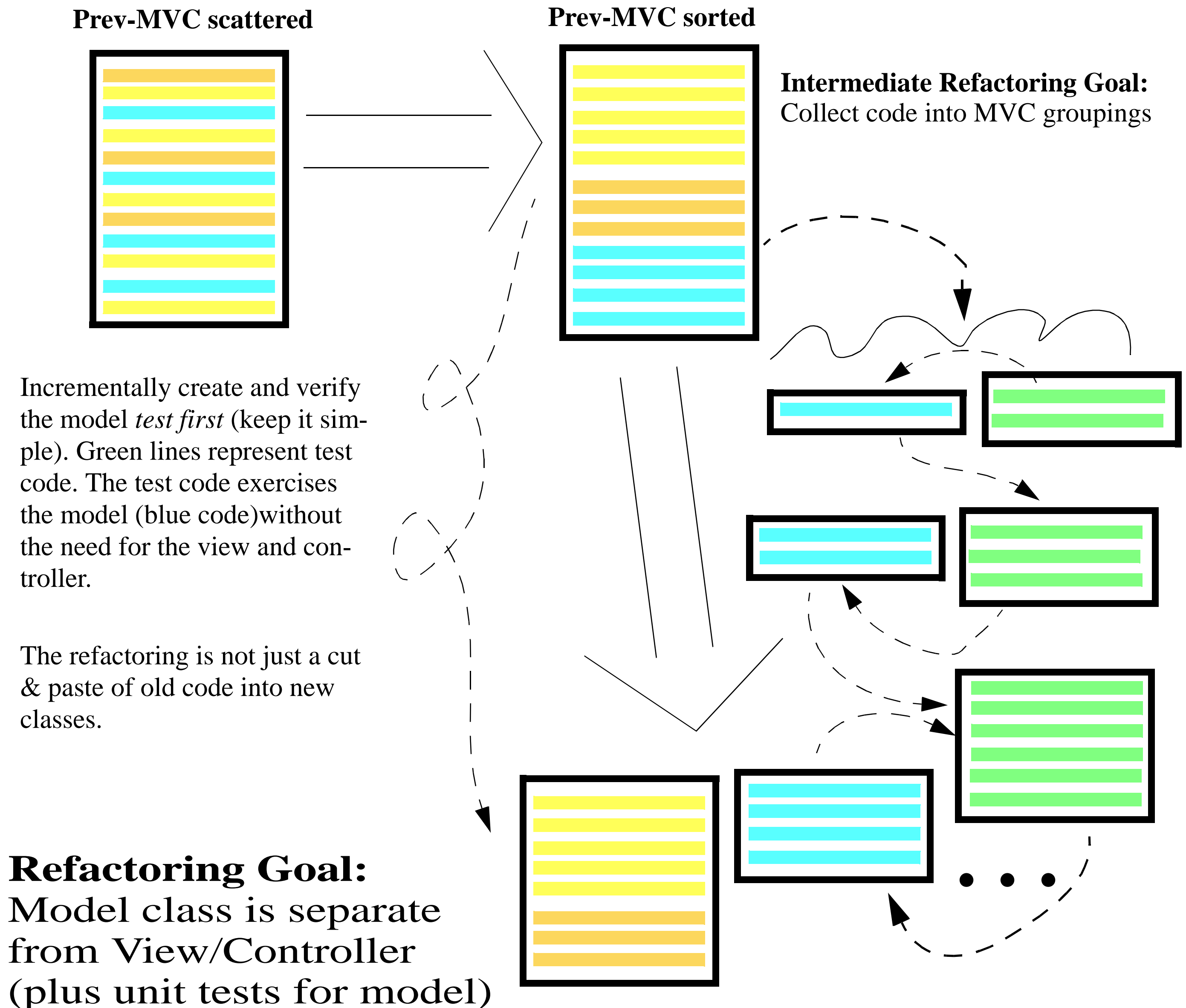

Figure 2: Refactoring the *model* out of the user interface. Methods in the *model* contain core logic independent of UI.These methods should be developed test first. The tests influence the design (keep it simple and modular), serve as example code and documentation, and verify that the model is (and stays) correct. Automated tests lessen the danger of breaking code during refactoring.

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 example;
import junit.framework.TestCase;

public class RaceGameModelTester extends TestCase {
      public static final int PLAY_A_INIT_POS = 0;
      public static final int PLAY_B_INIT_POS = 14;

      public void testGetPlayerPos() {
            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);
      }
}
```

This is only an example, your model may use a different set of methods with different names, return types etc.

Figure 3: An example JUnit test case confirming the initial player positions

Figure 3 shows an initial JUnit test written for the RaceGame application. JUnit is a commonly used framework for defining Java unit tests. Here we have assumed a model for the RaceGame application exists and have instantiated an instance of it. Methods starting with the word "test" verify the model works as expected by asserting expected behavior. The JUnit framework can automatically run all such methods and produce a report. One way the test drives the development is through compile errors since initially the test may refer to objects (*RaceGameModel*) and methods (model.*getPlayerPos*) etc. which do not yet exist. Eclipse will often make suggestions in these cases which may help to quickly generate code needed to get a successful compile. Figure 4 shows a simple RaceGameModel which makes the compiler happy.

```
package example;

public class RaceGameModel {
      public static final int PLAYER_A = 0;
      public static final int PLAYER_B = 0;

      public int getPlayerPos(int player) {
            return 0;
      }
}
```

This is only an example, your model may use a different set of methods with different names, return types etc.

Figure 4: A simple RaceGameModel which can be compiled and tested.

The compiler will also need to know where the JUnit libraries are. These examples use JUnit 3.8.2 which you should download onto your system. The project build.xml file will then need to be modified to run tests. Consult the software downloads page from Week 2.

A *local.properties* file should be created in the project directory to contain a path indicating where the JUnit directory is on your particular machine. Do not place the property file under version control since each machine may have their *junit.jar* in a different place. Figure 5 is an example *local.properties* file.

```
#define junit_home as the directory where junit is on your system
junit_home=C:/extras/org.junit_3.8.2
```

Figure 5: A sample local.properties file. Note the '/' used to specify directory path

Modify the compile target to be as shown in Figure 6. Also add the "unit test" target to the project build.xml file. Run the tests by executing the "unit test" target.

```
<target name="compile" depends="init">
   <javac debug="true" deprecation="true" destdir="${build.dir}/classes" srcdir="${src.dir}"
verbose="true">
      <classpath>
         <pathelement location="${junit_home}/junit.jar" />
      </classpath>
   </javac>
</target>

<target name="unit test" depends="compile" description="Execute Unit Tests">
   <java classname="junit.swingui.TestRunner" failonerror="true" fork="true">
      <arg value="example.RaceGameModelTester" />
      <classpath>
         <pathelement location="${junit_home}/junit.jar" />
         <pathelement location="${build.dir}/classes" />
      </classpath>
   </java>
</target>
```

Figure 6: build file target modifications to enable testing with JUnit.

Running the target "unit test" should compile and run the testing framework. Of course, the current test won't pass since the wrong answers are returned. The result is shown in Figure 7.

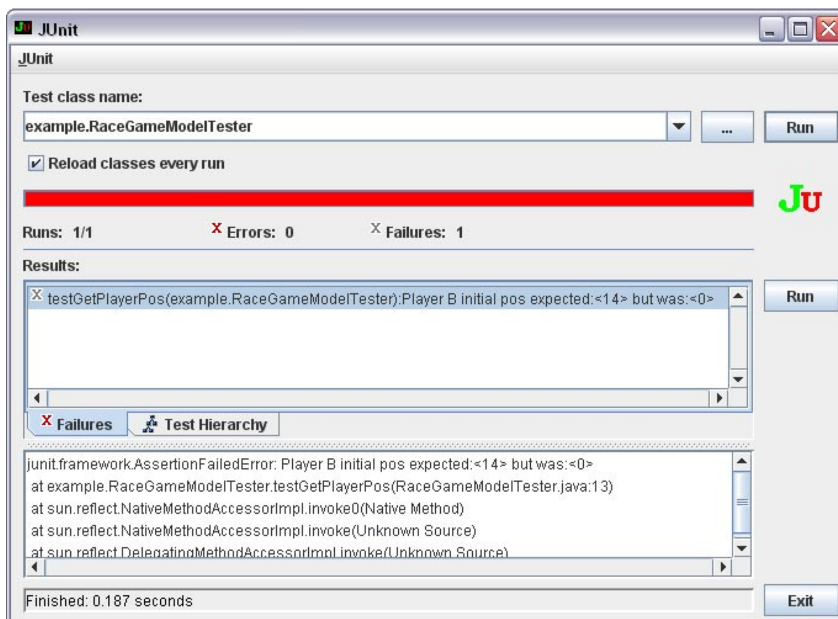

Figure 7: The JUnit test runner showing the "red bar" indicating at least one failed test.

Of course, the goal is to get the tests to pass. Take an honest approach to adding the simplest code needed to pass the tests. If your model needs to do more, write a test to justify the added functionality. Keep all the unit tests passing at 100%. Generally, the first time a test compiles and runs it should fail. Modify the code to make it pass. If your modifications break other tests, get those tests to pass again as well. When all the tests run successfully, the test runner will show you the "green bar" as in Figure 8.

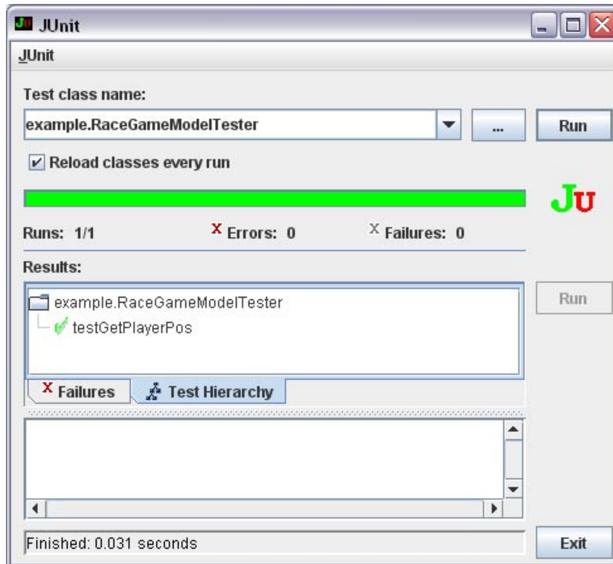

Figure 8: The JUnit test runner showing the "green bar" indicating that all tests have passed.

The "Test Hierarchy" tab has been selected showing which test have been run.

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. Once the code runs, verify that all the tests pass ("green bar"). If any tests fail, make modifications to so all unit tests pass. 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 tests. Each test should nicely map 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. 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.

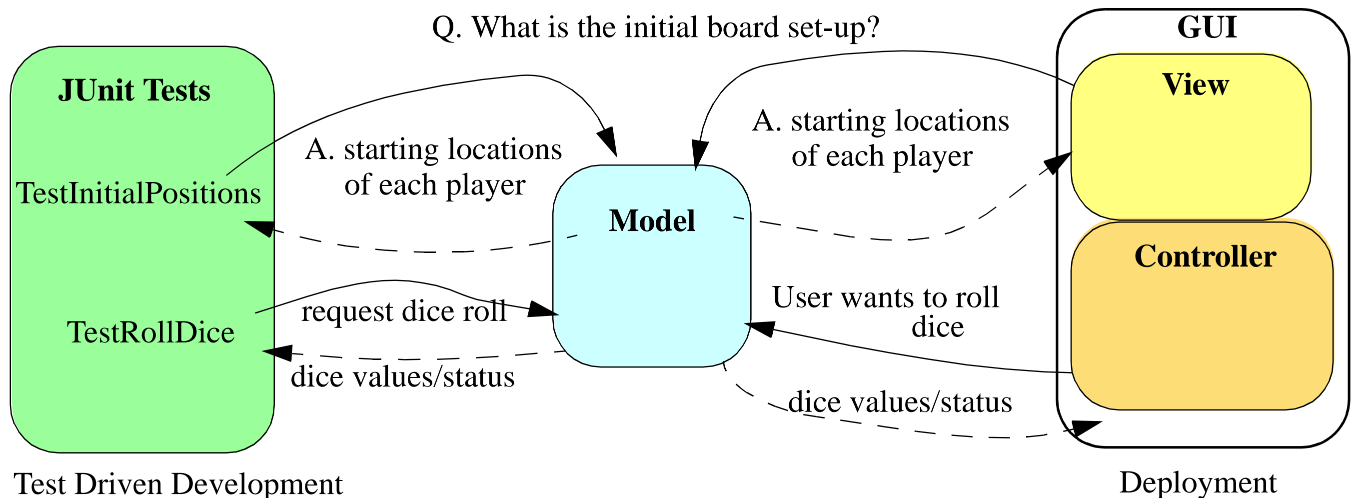

Figure 9: JUnit test cases drive the development of the model while the GUI is used in deployment