Refactoring to Use MVC

The refactorings that have been applied to *PegPuzzle.java* so far, have stayed within the same class. The code should now be reduced in size and easier to understand. Another smell still remains however since the class is trying to do too much - it needs to keep track of the game (board, 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 is provided by *view* and *control* components. See Figure 1. For simpler applications the *view* and *controller* are often 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.

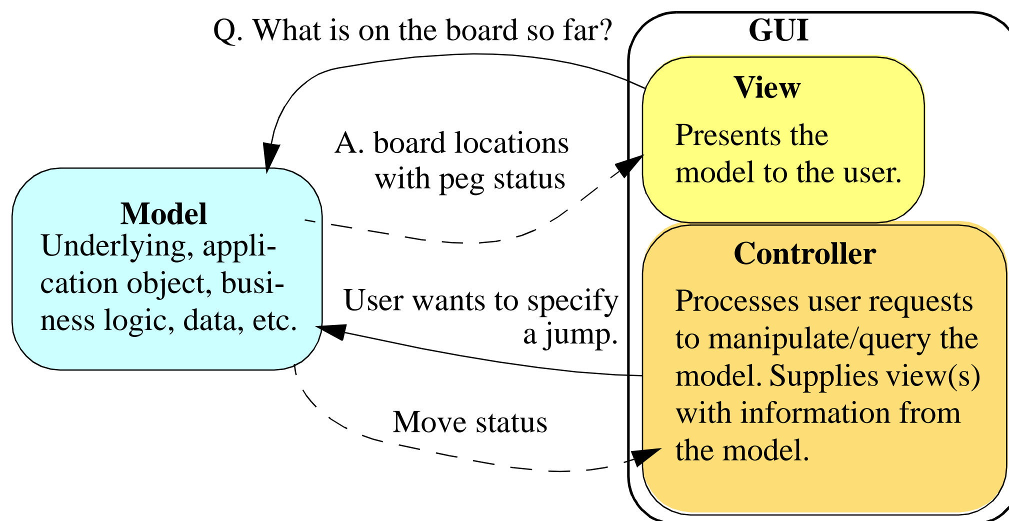

Figure 1: Example Model/View/Controller functionality for Peg Puzzle

The model can be developed independently from the user interface but it must provide methods, and perhaps constants, that can be accessed by the interface. For instance the model for the peg puzzle might have methods to initialize the board, process move requests, and report the state of the board locations. The controller interprets user inputs (user clicked on a button) and translates 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 (*getBoard*) and then displaying it (*setText* etc.).

To prepare for a refactoring in which the model is separated from the view and controller class, try to reduce duplication and clarify your code. Then, create meaningful methods which begin to look almost like the methods that will appear once the class is broken up. 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 *isPegAtLocation*, which is repeatedly called by the view in a routine *updateGUI* which redraws the board. (Since the board only has to be created once, the view might also have a routine *buildGUI* which initially defines the components). The controller code, mostly in the *actionPerformed* method, interprets the action expressed by the user and sends it to the model (*processLocation*). Then tell the view to update itself (*updateGUI*).

To help review your code, after it is nicely refactored for clarity and to remove duplication, **mark it up in Windows Journal**. Use the highlighter tool and <u>indicate which sections belong to which MVC component using the color convention in Figure 1 (model: light blue, view: yellow, controller: orange)</u>. Some of your methods may mix MVC components and should be further refactored. Some model methods may have small bits of view in them as the original code stores the board within the view. Each group should turn in their mark-ups before beginning on a separate model class. Groups should also make sure that their project is under version control at this point. (Start of class Wed.)

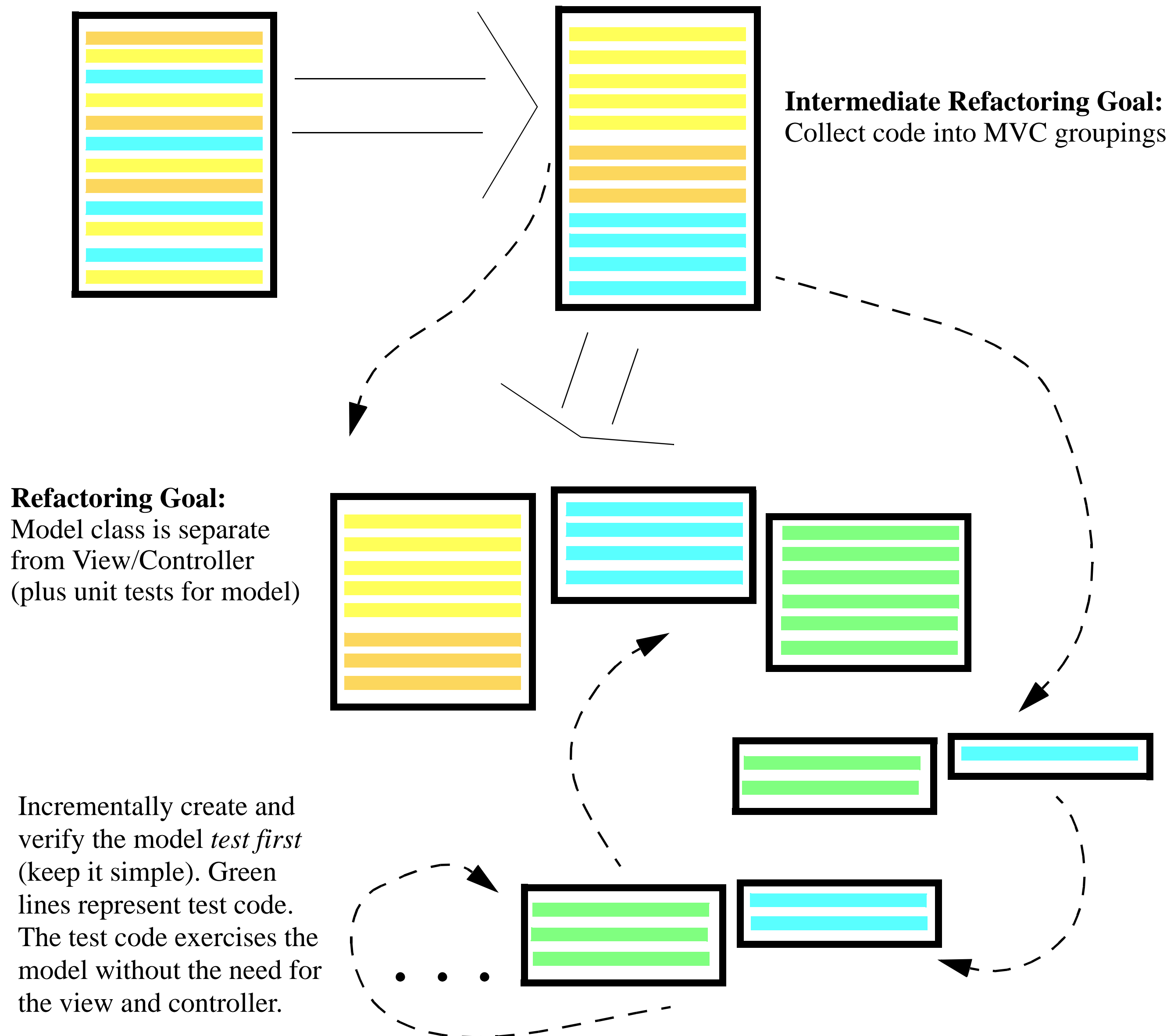

Figure 2: Refactoring the *model* out of the user interface. Methods in the *model* contain core logic independent of UI.These methods should be developed test first. The tests influence the design (keep it simple and modular), serve as example code and documentation, and verify that the model is (and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

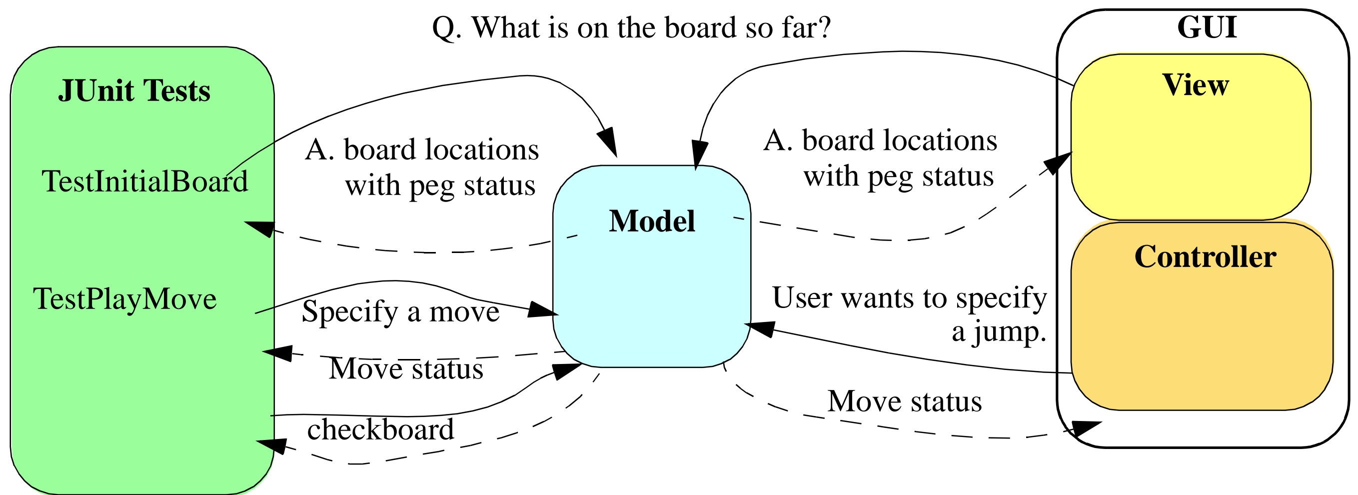

Figure 3: JUnit test cases drive the development of the model (GUI not necessary)

In Java, JUnit is a commonly used framework for defining unit tests.The tests call the same routines that the view and controller would so the model can be developed and tested without an interface. Just as you run a piece of code and check its output for correctness because you know the expected values, you can create test code to do that. Since we are getting used to XP practices, and using Test Driven Development (TDD), how do we express a test for code that doesn't yet exist? Think about how we would want to use the system if it did exist and how could we tell if it was working correctly. Start simply. For instance, when the application starts, the model should be initialized to some well defined start state (e.g. the initial board configuration). To display the model, the view will need to query it for its state information (which positions have pegs) so it can be visually represented accurately on the display.

Tests force us to consider the details of how to call methods, their parameters, what they should return etc. Test code acts as executable examples. They use *assert* calls so that they can automatically verify correctness. The JUnit framework provides various assert methods to compare results to what is expected. The framework automatically calls methods which start with the word "test". See the example in Figure 4.

```
package cs435ex1;
import junit.framework.TestCase;

public class PegPuzzleModelTester extends TestCase {
    public void testInitialBoard() {
        PegPuzzleModel model = new PegPuzzleModel();
        int numLocations = PegPuzzleModel.NUM_OF_LOCATIONS;
        int centerPosition = numLocations / 2;
        for (int i = 0; i < numLocations; i++) {
            if (i == centerPosition)
                assertFalse("center should be empty", model.isPegAtLocation(i));
            else
                assertTrue("start with a peg", model.isPegAtLocation(i));
        }
    }
}
```

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

Figure 4: An example JUnit test case confirming the initial state of the model

Setting up your project for testing

1. Create a new java class named *PegPuzzleModelTester* in the *cs435ex1* package (select the package and use the right button pop-up menu: New --> Class). Replace the contents of the file by Copy/Pasting from Figure 4. Eclipse will scan the code and indicate lines which may contain errors (red x boxes). Lightbulbs show that Eclipse has some possible suggestions for fixes. If you hold the cursor down over the top lightbulb (a suggestion for the problem with the import statement) suggestions will pop-up. Select the one which adds the JUnit 3.8.1 library to this project. The package view on the right will show another library (*junit.jar*) added to your project and the source code will have fewer problem lines left. Of course problems still remain because the *PegPuzzleModel* class does not exist.

2. To get the test to compile, create the *PegPuzzleModel* class. The goal is to keep it simple and to get it compile with a minimal amount of work.Create the *PegPuzzleModel* class shown in Figure 5.

```
package cs435ex1;

public class PegPuzzleModel {
	public static final int NUM_OF_LOCATIONS = 5;
	public boolean isPegAtLocation(int i) {
		return true;
	}
}
```

Figure 5: A simple *PegPuzzleModel* class so that we can compile the test code.

3. To actually compile and run the test, we need to modify the Ant build file. The new version is on the web. It changes the compile target to use *junit.jar* and adds a new target, *utest*, to run the unit tests. In addition, it defines where *junit_home* is by reading in a *local.properties* file.

4. The *local.properties* file should be created in your project directory and contain a path (use foward slashes '/') indicating where the junit directory is on that particular machine. Do not place this file under version control; each machine may have their *junit.jar* in a different place. Figure 6 is an example *local.properties* file.

```
#define junit_home as the directory where junit is on your system
junit_home=C:/Programs/eclipse3.2/plugins/org.junit_3.8.1
```

Figure 6: A sample local.properties file. Note the '/' used to specify directory path

5. Run, the tests by executing the Ant target *utest*. Assuming everything compiles, you will be presented with a JUnit test runner showing the results of your test runs. Green bar shows all tests pass, red indicates at least one problem. The model code given is too simple to pass the tests.

6. Modify the code just enough to get the test to pass. If you think the model needs to be more complex, create a test first to show how that feature is called and what the expected results should be. Keep in mind the functionality actually needed to support the GUI. Notice that the model is being run without any GUI code, the JUnit tests are driving the model code.

7. So the basic process is ...

Write Test case, Write enough Code to pass test ... repeat until needed functionality has been added.