Refactoring to Use MVC

The refactorings that have been applied to *MatchGame.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 (state of each card, number of cards flipped 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.

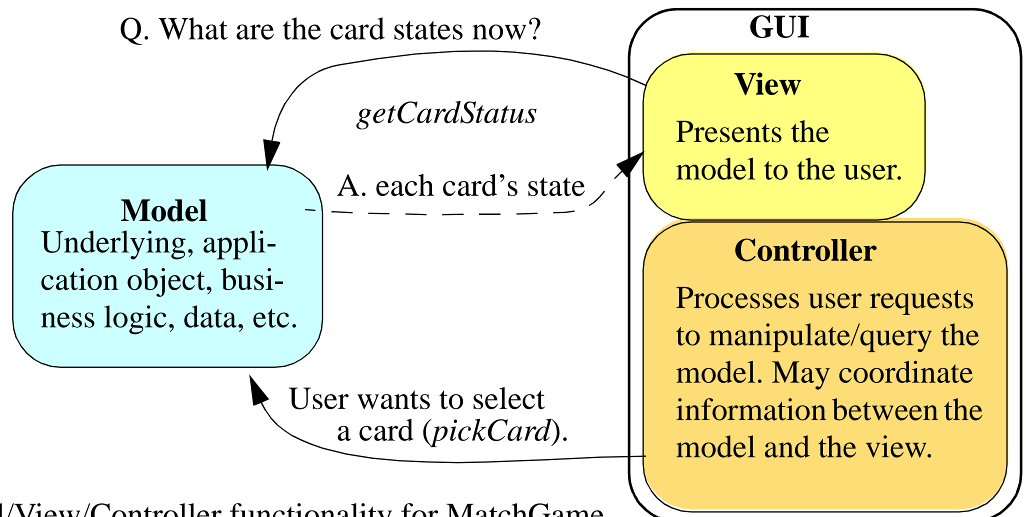


Figure 1: Example Model/View/Controller functionality for MatchGame

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 Match Game might have methods to initialize the game, process card selections, and report the states of the cards. A controller would interpret user inputs (user clicked on a card) and translate them to calls to the model (card selection). The model, knowing the game logic, can process the selection and update the cards and game state as 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 (*getCardStatus, getGameStatus*) 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 cards with the current information. The model might have a method like *getCardStatus*, which is repeatedly called by the view in a routine *updateGUI* which redraws the cards. (Since the graphical interface only has to be created once, a view method like *buildGUI* might 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 (*pickCard*) then tells the view to update itself (*updateGUI*).

If the model is developed separately from the interface, its logic can be developed and tested independently from how it will (might?) be displayed and manipulated by the user. Extreme Programming(XP), uses the practice of Test First or Test Driven Development (TDD) to drive the design of code, 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.

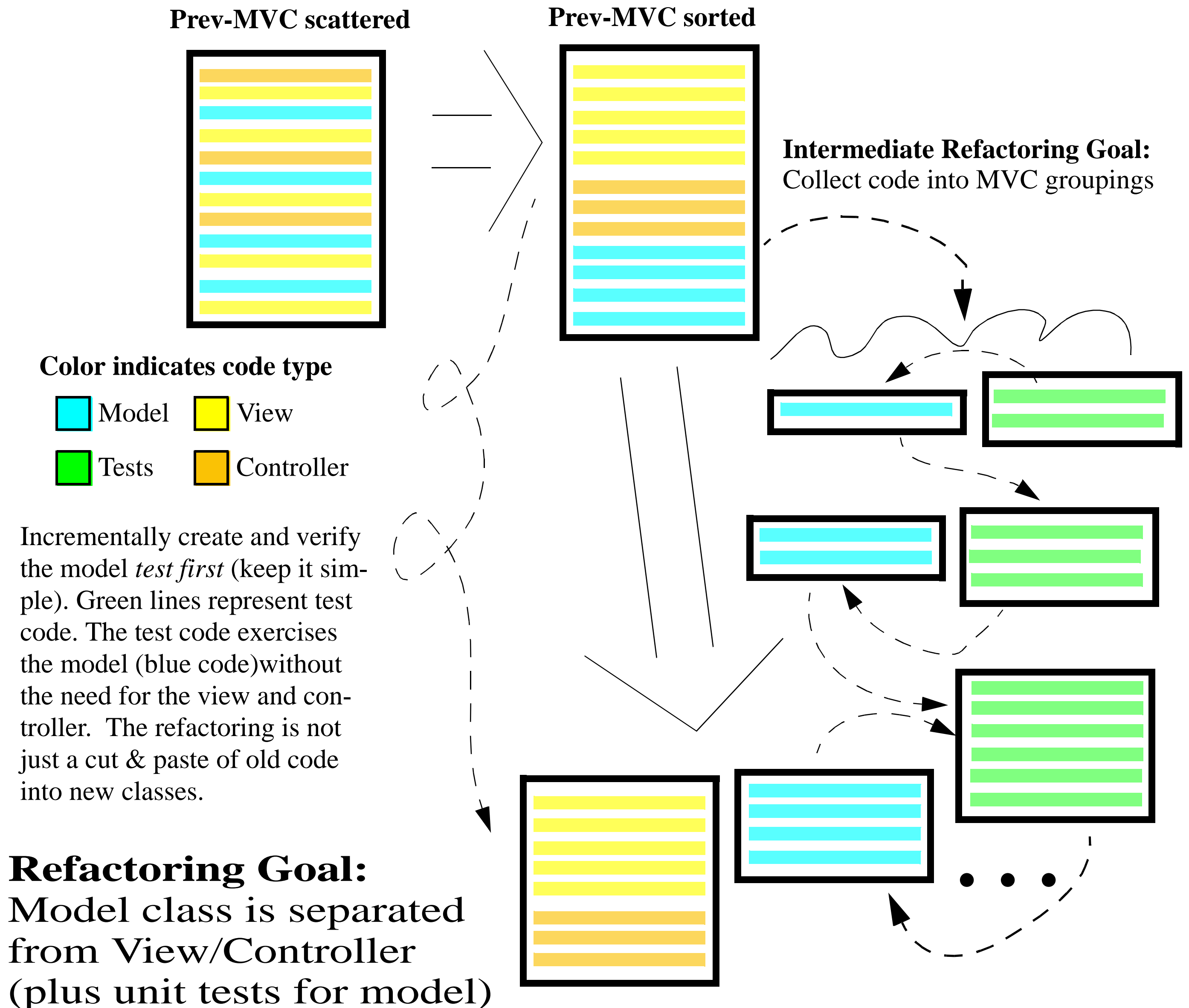
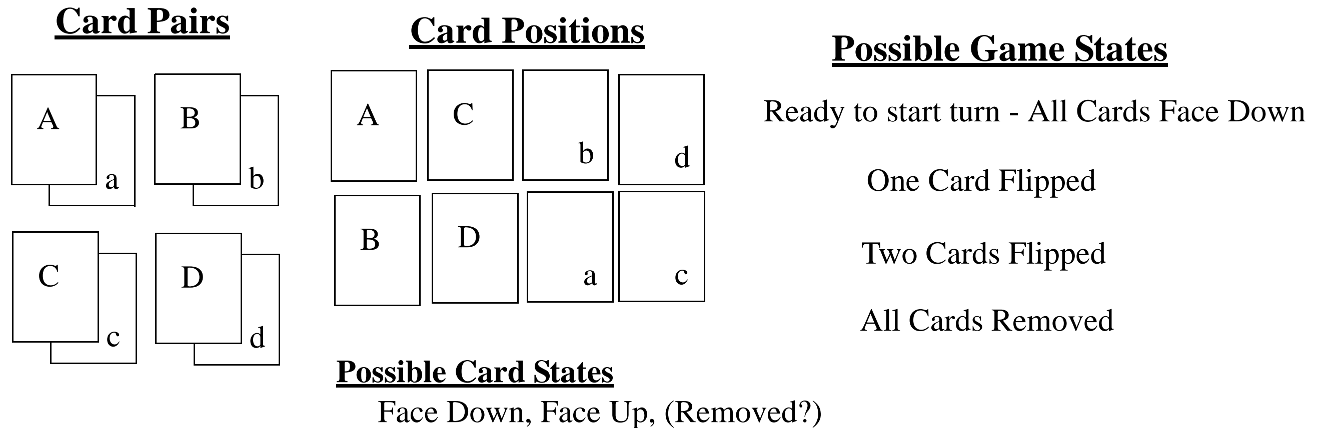


Figure 2: Refactoring to separate the *model* from the user interface. Methods in the *model* contain core logic independent of the UI. These methods should be developed test first. The tests influence the design (keep it simple and modular), serve as example code and documentation, and verify that the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

Building the Model with Test Driven Development

What is 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 match game model, might be thought of as shown in Figure 3. Just as with computer code, there may be multiple approaches and some fine tuning of concepts through iteration and refactoring.



Possible Game Actions/Queries

initialize, dealCards, pickCard, getNumCards, getCardStates, getState

Figure 3: Exploring basic concepts of the Match Game model.

To begin test driven development, consider what happens when the application starts. The model should be initialized to a well defined start state (e.g. the initial game configuration). To display the game state accurately for the user, the view will need to query the model for its state information (states of the cards for instance). **Test Driven Development** is motivated by using tests to force functionality for 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 (we should know the correct answer). The JUnit testing framework lets you build up tests using *asserts* that confirm that expected results match the values retrieved from the model. (See Figure 4)

```
package cs435;
import junit.framework.TestCase;

public class MatchGameModelTester extends TestCase {

   public void testGetInitialState() {
      MatchGameModel model = new MatchGameModel();
      int initialState = model.getState();
      assertEquals("Game initial state", MatchGameModel.READY, initialState);
   }
}
```

This is only an example, your model may use different names, return types etc.

Figure 4: An example JUnit test case confirming the initial game state.

Figure 4 shows an initial JUnit test written for the *MatchGame* application. JUnit is a commonly used framework for defining Java unit tests. Here we have assumed a model for the *MatchGame* 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 (*MatchGameModel*) and methods (model.*getState*) 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 5 shows a simple *MatchGameModel* which makes the compiler happy.

```
package cs435;

public class MatchGameModel {
      public static final int READY = 0;

      public int getState() {
            return 0;
      }
}
```

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

Figure 5: A simple MatchGameModel 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 should be on your system (likely bundled with Eclipse). To remove the red ‘X’s Eclipse uses to indicate source code errors, your project will need to be told where the JUnit library is. The import of the JUnit framework will be marked with an ‘X’ but a “Quick Fix” to *fix your project set-up* will likely appear and suggest to add the JUnit 3 library to your build path. After letting the system do this for you, the JUnit library will be linked in and you can check the properties of this library in the Package Navigator to reveal its location. The location will be needed to update your build script.

Since we will compile and run the tests using Ant, the project *build.xml* file also needs to be modified to include information about where the JUnit libraries are. To allow for each developer to have libraries stored in a different place, the machine specific information is placed in a *local.properties* file which should be created in the project directory. Right now it only needs 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 6 is an example *local.properties* file.

```
#define junit_home as the directory where junit is on your system
junit_home=C:/Programs/eclipse/plugins/org.junit_3.8.2.v20080602-1318
```

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

Modify your *build.xml* file to include the following line just above the other property definitions:

```
<property file="local.properties" />
```

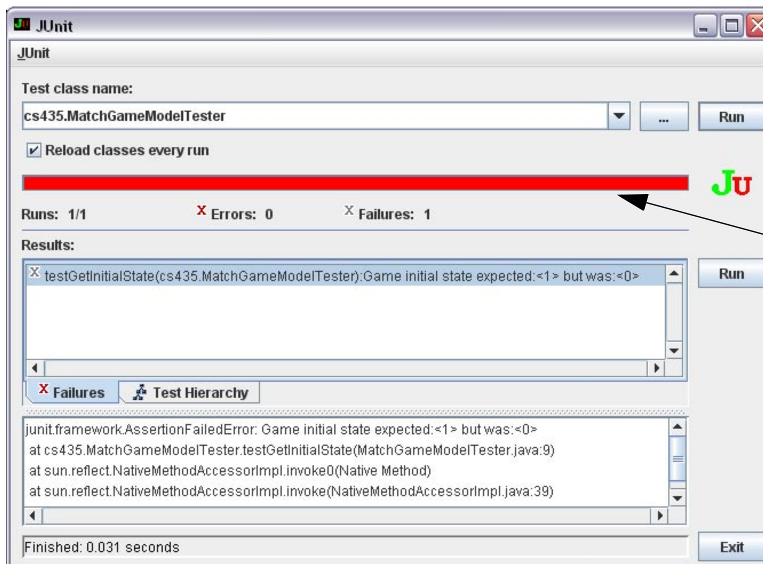
Also refer to Figure 7 and make the modifications to the compile target as well as add the *unit test* target.

```
<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.MatchGameModelTester" />
      <classpath>
         <pathelement location="${junit_home}/junit.jar" />
         <pathelement location="${build.dir}/classes" />
      </classpath>
   </java>
</target>
```

Figure 7: build file target modifications to enable testing with JUnit.

Running the target *unit test* should compile and run the testing framework. Naturally, the current test won't pass since the wrong answer is returned. The result is shown in Figure 8.



The "red bar" will become the "green bar" when all the tests pass.

Figure 8: The JUnit test runner showing the "red bar" indicating at least one failed test.

Of course, one of the goals is to get the tests to pass. Keep all the unit tests passing at 100%. Generally, the first time a test compiles and the code 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 9. After all tests pass, refactor if sections of your code start to smell. Rerun the tests after refactoring to assure everything still works as expected.

If your model needs to do more, write a test to justify the added functionality. This technique helps to reenforce the *YAGNI* principle. *YAGNI* stands for "you ain't gonna need it" which reminds you to focus on the current problems rather than trying to plan too far ahead for future capabilities which may waste time since they often turn out to never be needed. Build the model incrementally by adding the simplest code

needed to pass the tests and writing sensible tests which force the model to grow in support of required functionality. DO NOT simply plan out and write all the model code first, then write the tests.

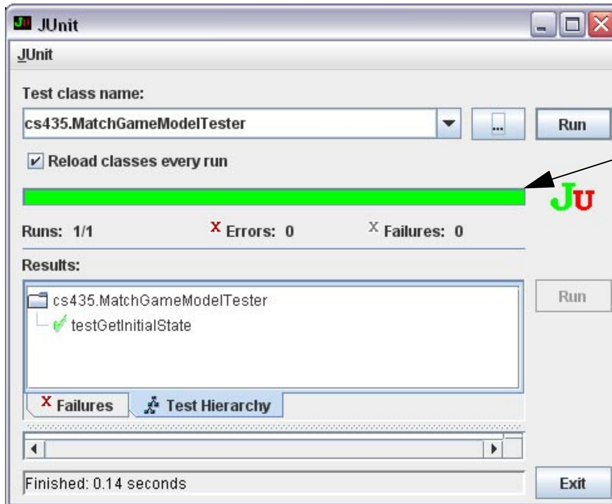


Figure 9: The JUnit test runner showing the “green bar” indicating that all tests have passed.

The “Test Hierarchy” tab has been selected showing which tests 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. If everything is passing but getting messy, refactor and check to make sure that the tests still pass. Test code is code also so keep it clean and refactor as necessary. Test code is also an example of how to use the model code.

Methods and tests can be created incrementally. You should make your tests independent of each other and not assume any particular order. The model should not be aware of the tests - no code branches that check for tests. 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.

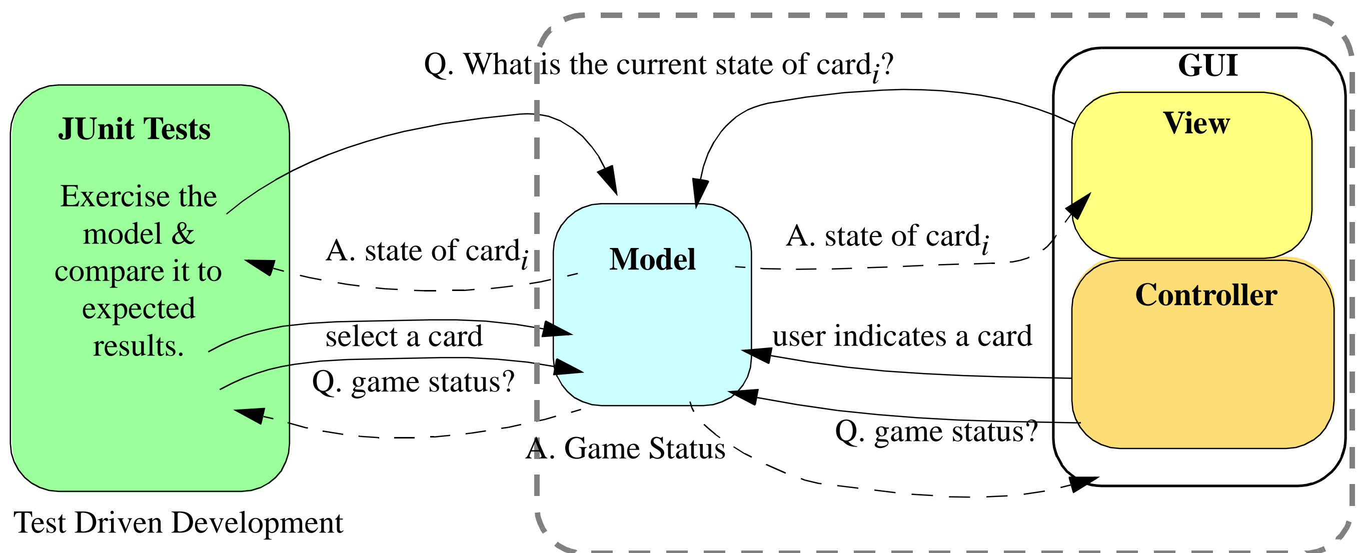


Figure 10: JUnit test cases drive the development of the model. The GUI drives the model in the application. Application code is shown within the dotted lines.

Test Driven Development is also described in Chapter 8 of “Head First Software Development”.