Introducing Application Design and Software Engineering Principles in Introductory CS Courses: Model-View-Controller Java Application Framework

Scot F. Morse
Division of Computer Science
Western Oregon University
345 N. Monmouth Ave.
Monmouth, OR 97361
morses@wou.edu

Charles L. Anderson
Division of Computer Science
Western Oregon University
345 N. Monmouth Ave.
Monmouth, OR 97361

ABSTRACT

This paper presents an approach for introducing application design and software engineering principles early in the CS curriculum. When courses are taught in a language such as Java, programming labs may involve writing a complete application or applet. In these cases it is possible to introduce notions of software engineering principles alongside a topic such as object-oriented design. This paper presents a simple application framework in Java that follows a Model-View-Controller design and that can be used in introductory and core courses to introduce elements of software engineering. Although it is focused on applications having a graphical interface, it may be modified to support command-line programs. The application framework is presented in the context of development tools Apache Ant and JUnit.

1. INTRODUCTION AND MOTIVATION

Computer Science curriculums frequently include courses in Software Engineering and Application Design. Often these courses are offered late in a student's program, long after the basic foundations of programming and algorithmic problem solving have been covered. While many of the formal topics in Software Engineering are rightly covered in an advanced setting, teaching good application design principles and practices need not wait for upper-division courses. With the aid of a simple framework, issues of design, development, and testing can be introduced in a CS2 level course in a way that does not hinder its progress. In this paper the authors present an application framework in the Java programming language that may be used to teach students to organize the logic of their programs in ways that are easy to understand, trace, expand, maintain, and test. The framework chosen follows the highly successful Model-View-Controller (MVC) paradigm.

The impetus for the development of this framework and its trial use in introductory CS courses was the conspicuous observation that students tend to follow the examples given them in their textbooks. Textbook authors strive to present good programming practices and examples, so the examples themselves are not the problem. Rather, the

problem comes when students are asked to write complex programs that go far beyond the scope of the examples found in most textbooks. For example, in a data structures course students may be asked to implement a particular data structure and then use it in a corresponding algorithm in a fully functioning program (e.g. a stack used in a postfix calculator application). Students may be successful implementing the core functionality of the problem (the data structure and integral algorithm) but submitted very poorly-written programs as a whole. When graphical user interfaces (GUI) were a requirement of the assignment, students often wrote the entire program in a single class file containing hundreds of lines of less-than-readable code – graphical objects instantiated right next to the data structures, graphical layout code intermixed with data management, and haphazard uses of static variables, instance variables, and anonymous listeners. Students correctly followed good object oriented (OO) principles in designing and implementing their data structures, but when it came to designing their application, they had no previous experience or examples to draw from. Introductory programming texts focus on principles of programming; data structures texts focus on theory and implementation of data structures and algorithms. Yet many popular texts also present GUI programs as examples, in a very limited form. Page and space restrictions leave little room in these texts for a discussion of good application design and simple software engineering principles. Students are left to either follow these limited examples or develop a good framework on their own. Experience shows few students do the latter.

This paper suggests that these important ideas can be presented in an introductory setting (e.g. in the second quarter of an introductory programming course or the first term of a data structures course) without hindering the development of the core ideas of the course. A solution presented here is for the instructor to provide an *application framework* for the students to use in their own programming projects. This is a "skeleton" application that provides the structure for a functioning program. Students are then required to use this framework to write their programming assignments.

There are many benefits of using a framework of this type. First, it shows the students an example of good application design that complements principles of object oriented design. Because software engineering has become a fundamental part of many curriculums, the earlier students are exposed to the idea of *engineering their programs*, the better. Second, this framework allows students to create rich graphical programs while spending a maximum amount of time on the core ideas of the assignment. Since all students are working from the same core framework, the instructor can choose to provide a custom graphical interface class that students can "plug-in" to their code without changes. Students may use it as is (with only the definition of its public or package level members), enhance it, or extend it as they wish. In a similar way, this framework shows how software development can work in a team environment; they can write their own code without knowing the details of another component. Third, a common framework assists in evaluation and grading tasks. If the primary function of an assignment is the implementation of a data structure, the instructor knows exactly where to look for it. It will not be obscured by code unrelated to its function. In a similar manner, students will know exactly where to look for important code in the instructor's solutions after the assignment is over. In addition, the framework enables unit testing to be carried out automatically, which may be useful in assigning grades. Common tasks such as compilation, execution, file management, and archiving (for submission) can also be automated easily with the

build tool Apache Ant [1]. Lastly, the framework may be presented in various levels of sophistication – a very simple version for an introductory class or a more complex version for advanced classes.

Drawbacks of using this framework center around the additional burden it places on students. Requiring its use forces students to learn how it works at a point early in the term, possibly reducing their focus on the content of the course. It is often difficult for students to be suddenly immersed in another person's code and be expected to work productively. It is for these reasons that the framework is designed to be as simple as possible and organized in a natural and readable fashion.

2. MODEL-VIEW-CONTROLLER PARADIGM

The Model-View-Controller (MVC) is a popular architecture for interactive applications that was originally developed for use in Smalltalk systems [4]. It has enjoyed considerable popularity in Java programs as Sun Microsystems has used it extensively for the Swing GUI toolkit, servlets, and enterprise applications. The MVC architecture is not limited to Java; MVC frameworks exist for many languages and systems including Python, PHP, Microsoft ASP, and Apple's Cocoa application environment.

There are three major components in the MVC architecture:

- A *Model* contains the underlying data and methods of the application. This is often referred to as the "business logic" of the application.
- A *View* contains a representation of the data in the model. This displays data from the model based on the current state and may receive update notifications when the data changes. The view is concerned with output to the user. The view also displays the user interface components that receive input from users.
- A *Controller* connects the model and view and coordinates activities between them. Based on user inputs, it determines which methods on the model should be invoked and which view should display the updated data in the model. The controller is responsible for processing input from the user.

Sometimes the view and controller are combined for simplicity or performance reasons. Regardless of whether the view and controller are separate entities, the model is always separate from the view. This leads to a number of desirable properties including:

- The same set of business logic (a model) may be used with numerous different views to provide different user interfaces for the same underlying application. These could include web page display, traditional GUIs, or a web-services interface.
- By separating the model from the view and controller, separate teams of developers can work on each component, either serially or in parallel. In a teaching environment, the instructor could provide a GUI to the students who would be responsible for implementing the model.
- Automated test harnesses such as JUnit [2,5,7] can be used to perform extensive unit testing on the business logic without tedious testing at a GUI. This reduces or eliminates the need for (often expensive) automated GUI testing tools, and it makes debugging simpler: if the model is known to be correct (based on unit tests), incorrect output must be due to errors in the view (or controller). For students, unit testing results in much higher quality programs because the programs have been exhaustively tested.

To better understand the roles of the different objects in the MVC architecture, consider an application for performing certain tasks in a payroll system like entering time records for employees. At the end of a week, a paycheck needs to be generated for each employee, and taxes need to be computed and withheld. In this example, the model would contain a method to specify number of hours worked by an employee in a day or a week and would most likely store this information in a database. There would be a separate method to generate a paycheck for a given employee, which would probably call methods to look up tax tables in the database, compute the taxes, and store information about the resulting paycheck in the database. Notice that the model is not concerned with where the input data comes from or how the output data is displayed. Also note that only the model knows how to perform the computations, and only the model is concerned with interacting with the database.

The view displays the data to the user. In a GUI application, the view may include text boxes to display data or accept inputs. It may contain some type of tabular display to show data for every employee. It may also include the ability to print the paychecks or summary reports. If the application were web-based, the view would consist of HTML elements like forms for input, tables for output and formatting, and formatted text to display individual values.

The controller processes inputs from the user and determines which method on the model should be invoked and which view should display the result. When the user selects a GUI menu to enter employee timesheet data, the controller selects the proper view to display the input fields. When the user clicks a button to submit an employee time record, the controller calls the appropriate method on the model to cause the data to be recorded in the database.

3. BASIC JAVA APPLICATION FRAMEWORK

A simple Java MVC framework has as its base a class containing the main method and one class for each of the functional domains described above, appropriately named: *Main*, *Model*, *View* and *Controller*.[1] Figure 1 shows this relationship in a UML diagram.

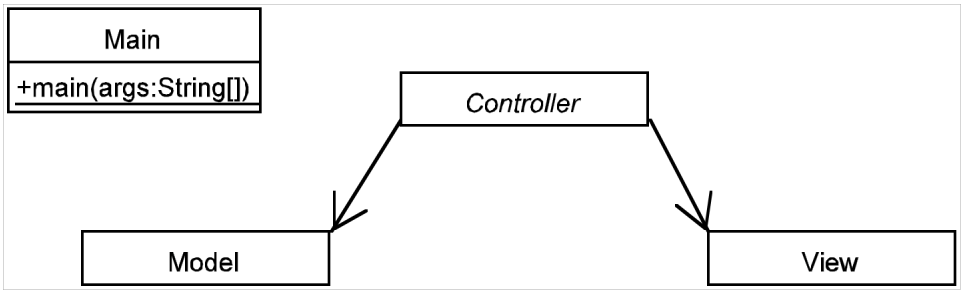

Fig. 1. UML diagram of top-level framework structure. The *Controller* maintains a reference to both the *Model* and the *View*. There is no direct means of communication between the *Model* and the *View* – the *Controller* manages all communications at this level.

[1] Actual class and interface names are in italics and are capitalized.

The *Model* class contains all the fundamental data and logic required by the program while the *View* class contains all the graphical user interface components and functionality. The *Controller* is the primary event handler that responds to events generated by the user through the graphical components in the *View*. (For the moment, there is no way for the *Controller* to respond to events generated in the *Model*.) A *Main* class is used as the entry point for the program. It is responsible for initializing the three objects.

This particular realization of the MVC paradigm is kept as simple as possible at this point. As such, we intentionally separate the model and the view as much as possible. The *Model* does not have a reference to either the *Controller* or the *View*. Similarly, the *View* has no reference to the *Controller* or the *Model*. (This separation should also be enforced by strongly discouraging static variables and methods in all classes.) This is a simpler model than other MVC implementations because the *Model* and *View* are treated as delegate objects of the *Controller*. They each perform their own separate function independent of the other objects. In this way they are easier to implement and can be written independently. As its name suggests, the *Controller* manages the top level functioning of the program by responding to events and "controlling" the *Model* and *View*. It uses them as it needs to in order to implement the functionality of the program. At this point it is instructive to see how the *Controller* typically works. Consider a calculator program. The *Controller* may receive an event corresponding to "Enter" or "Evaluate" an expression. The *View* holds the user input and the *Model* contains all the functionality for evaluating the expression. The *Controller* may perform something like the following, in an *actionPerformed* method handling the appropriate event, e.g.:

```
String answer = model.evaluatePostFixInput(
                    view.getInputExpression() );
view.displayResult( answer );
```

From the perspective of the *Controller* there is no specification of how the *View* is to obtain the input string from the user. There is likewise no knowledge of how the *Model* carries out the computation. These details are left up to the individual classes. However, each method must perform the required function and the *Controller* directs the execution. This primary level of design teaches students about programming by interface.

Next, as described in Figure 2, the details of this general framework are developed. To solidify the separation of components in the framework, note that the three primary classes (*Model*, *View*, and *Controller*) are placed in separate packages. The *Model* and *View* are placed in their own packages (in Figure 3 named with the convention of *course.name.lab#)* and now represent well defined, individual subsystems. The *Controller* is placed one level higher, alongside the *Main* class.

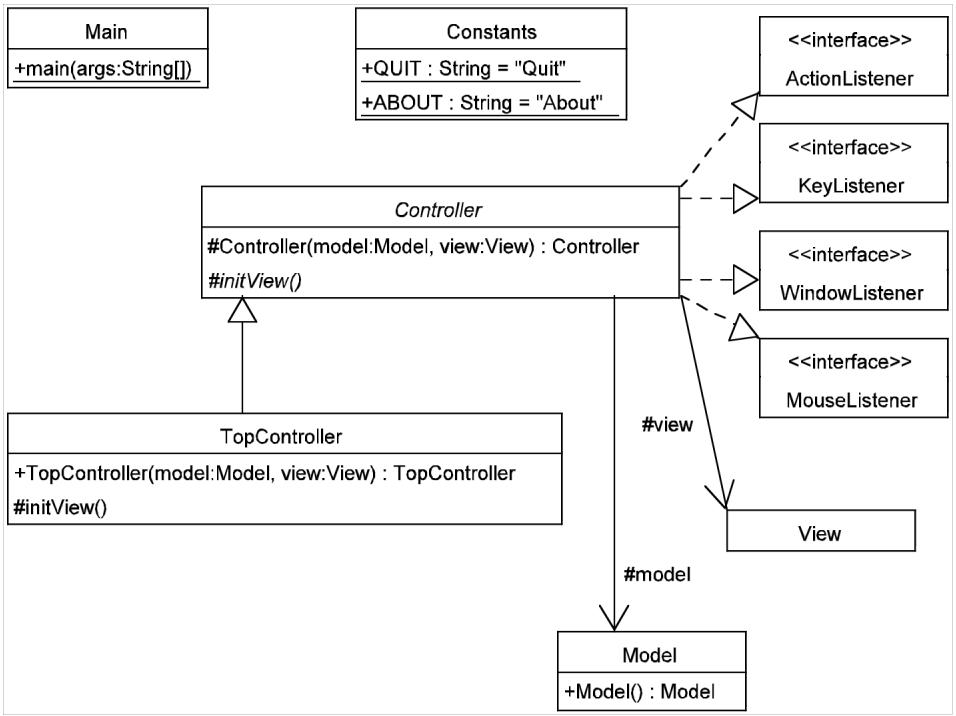

Fig. 2. UML diagram of the basic MVC application framework. The *Controller* is now an abstract class. Applications extend this as needed. In this case the top level application controller is *TopController*, which overrides the base class methods as necessary. Not shown is the fact that the *View* and *Model* are located in separate packages.

For now, the *Model* class is alone in its own package. While often the most important subsystem, the model has no further framework development at this point. One must wait for a specific application to add classes in this area. (Later, under the category of "advanced" features, functionality is added to the model subsystem.)

Figure 2 describes additions to the *Controller* subsystem. The *Controller* serves as the listener object for application-wide events and so must implement a number of listener interfaces. The term "application-wide" is used here to distinguish between events that apply to the application as a whole (Open, Save, or Quit menu items) and to those events local to a particular graphical component (a *KeyEvent* from a text component that needs to be handled locally, in the *View*, and has no effect on other subsystems). Local events often should be handled as close to their sources as possible. Implementing many interfaces introduces numerous, often empty, methods. This presents a good case for inheritance. The *Controller* becomes an abstract class that provides stub implementations of the interface methods. A new concrete class, here called *TopController*, extends *Controller* and overrides listener methods as needed. (For a discussion of this design pattern in C++ see reference [6].) It inherits references to the *Model* and the *View* classes. Using inheritance in this case makes the *Controller* class reusable and also simplifies the actual *TopController* class. Students need not sort through many empty listener methods to find the one they need.

One utility class is added to the top level package: *Constants*. This class is meant to be used for application-wide (often) constant values. Uses can range from default sizes and colors to important initialization values. Its most common use is to hold

constants that identify *ActionEvents*. Since the graphical components are in the view subsystem (buttons, menu items, ...) and the listener is in the *TopController*, there is no direct way to identify the source of an event (by reference), unless the data members of the *View* are given public visibility. In order to maintain good encapsulation in the subsystems, it is suggested to keep members with package or private visibility in all subsystems, but link events to objects using string constants. This is implemented by specifying a constant as the *ActionCommand* of the graphical component:

```
// In the view subsystem
JButton button = new JButton("OK");
button.setActionCommand( Constants.OK_EVENT );
```

Then, when the event is handled in the *TopController*, only the following is necessary to identify the event and its source:

```
// In the TopController; e is an ActionEvent
String command = e.getActionCommand();
if( command.equals( Constants.OK_EVENT ) )
{ ... }
```

The *TopController* can then respond to the event by delegating work to the *Model* then telling the *View* to update itself.

The view subsystem can now be filled out. Since the framework presented here is the GUI version, the *View* must be set up to at least handle a window and some menus. Aside from a constructor and possibly a GUI setup method, the *View* needs the ability to register a listener. The *setListener* method serves this purpose and accepts a reference to a *Controller* object. It is in this method that graphical components may add the *TopController* as a listener. This public method is invoked as soon as it is determined which *TopController* is to serve as the controller for that view. When there is only one, as is the case in this paper, this method can be invoked from either the *main* method or the *TopController* (either through its constructor or the general purpose *initView* method, which is used to direct initialization of the *View* upon startup of the application). It is important to note that the *setListener* method does not violate the separation between the view and model subsystems. The *View* is passed a reference to "its" *Controller*; however, it cannot obtain a reference to the *Model* because of visibility restrictions. A typical view subsystem is shown in Figure 3.

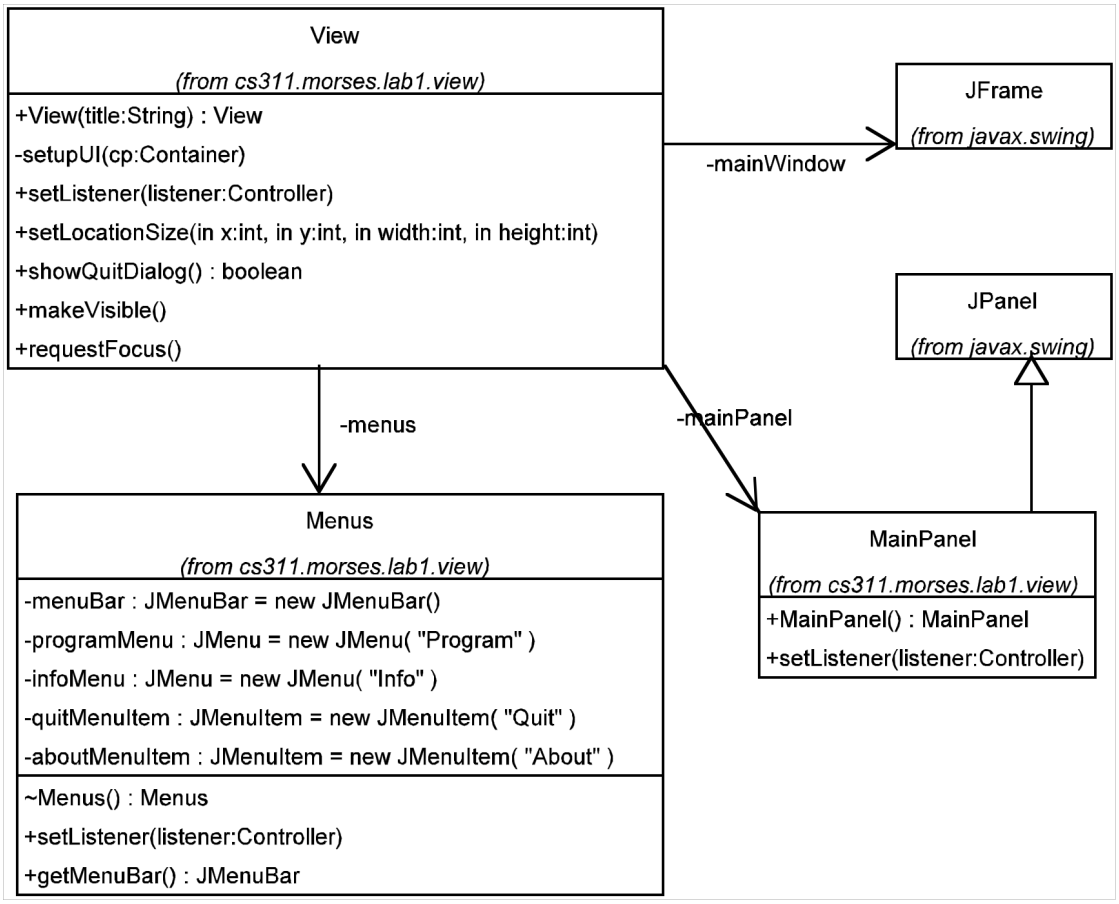

Fig. 3. UML diagram of view subsystem.

The *View* class serves as the entry point to this subsystem. The *View* owns a window (*JFrame*), a *Menus* class (which manages the various menu-related objects), and a *MainPanel* (*JPanel*), which contains the application's main window graphical layout. It is straightforward to keep graphical components and their layouts within the *View* class (or a class extending *JFrame*). The approach of having a custom *JPanel* class is useful when the instructor wishes to give students the GUI for the program. The instructor writes the *MainPanel* class (with its necessary *setListener* method) with the entire GUI layout for the content area of the main window. Students can then simply replace the default *MainPanel* class with the custom one. No further changes are necessary since the *View* is already set up to place a *MainPanel* object in the content pane of the top level window.

It is instructive at this point to trace the application start-up. The *Main* class holds the *main* method, which instantiates the trio classes *Model*, *View* and *Controller*.

```
public static void main( String[] args )
{
 Model model = new Model();
 View view = new View( "Generic Application" );
 Controller controller = new TopController(model,view);
 view.setListener( controller );
 controller.initView();
```

```
    view.setLocationSize( 100, 100, 700, 433 );
    view.makeVisible();
  }
```

The *Model* and *View* must be instantiated first. References to these objects are then passed to the constructor for the *TopController* object. This forms the one-way links between the *Controller* and the two subsystems. At this point nearly all the setup has been completed in each subsystem. All graphical components have been instantiated (but not displayed) in the view and all data related objects set up in the model. In applications that use the file system or a database, it is at this point that the *Model* may acquire access to external resources. Handling exceptions at this point is easy – instantiation of the *Model* and/or *View* could be placed into a try-catch block which exits the application appropriately upon failure. A further enhancement could be that the *Model* must be successfully instantiated before the *View* is created. After the *TopController* is instantiated, it is necessary to ensure that it is registered as the listener object for any components in the *View*, hence the *setListener* method call. At this point the *View* has only been set up in a default state, unaware of the state of the *Model*. To properly initialize it with the *Model*, the *initView* method is invoked on the *TopController* object. This enables the controller to properly initialize the *View* (or indeed the *Model* as well) before the application has been made visible to the user. Finally, once all application objects have been instantiated and initialized, the primary GUI can be placed on the screen accordingly and made visible to the user.

4. MORE ADVANCED EXAMPLES

Section 3 detailed the implementation of a basic version of a MVC Java application. It is enough to use as a framework for a range of programs in introductory courses, yet is sufficiently easy to understand that using it should not hinder the core material presented in the course. There are, however, significant additions that can be made to the framework to improve its design and enhance functionality.

A first addition is to enable communication from the *Model* to the *Controller*. As it is designed above, there is no way for the *Model* class to signal the *Controller* that it needs attention. This type of situation can appear frequently in programs that execute data processing in multiple threads (located in the model subsystem). The *Model* may create a worker thread to process data at the request of the *Controller*. When finished, it may need to notify the *Controller* it is completed and that the *View* may need to be updated. This functionality can easily be implemented by using the Observer pattern [3] realized in the Java *Observable* class and *Observer* interface in the *java.util* package; the *Model* extends *Observable,* and the *Controller* implements *Observer*. The *Model* notifies the *Controller* through the *setChanged* and *notifyObservers* methods. The *Controller* responds through the *update* method, which receives a reference to the *Observable* and an arbitrary message.

Another enhancement to the framework addresses the event handling model itself. *ActionEvents* comprise a large fraction of the events processed by a typical GUI program. Even a simple program may have a half dozen or more menu items and several buttons. This requires there to be a dozen or more event constants handled in the *actionPerformed* method in the *TopController*. Events are identified by strings and so the event handling is often performed in large nested if-else statements. One

can see this exposes a fault in the design as the application scales up in size. Even an application with twenty distinct events would be messy to maintain. A solution to this problem associates an object with each event type. This is commonly known as the Command pattern [3]. Each event action has an associated *Command* object, which contains the code to handle the event. The event handling framework (in the *Controller*) extracts these *Command* objects out of a hash table using its event string constant as a key. The framework then invokes the event handling method without needing to know what event it is. *Command* objects are easily reused for similar operations (Open menu item and Open toolbar button). Figure 4 shows how this pattern can be incorporated into the current framework.

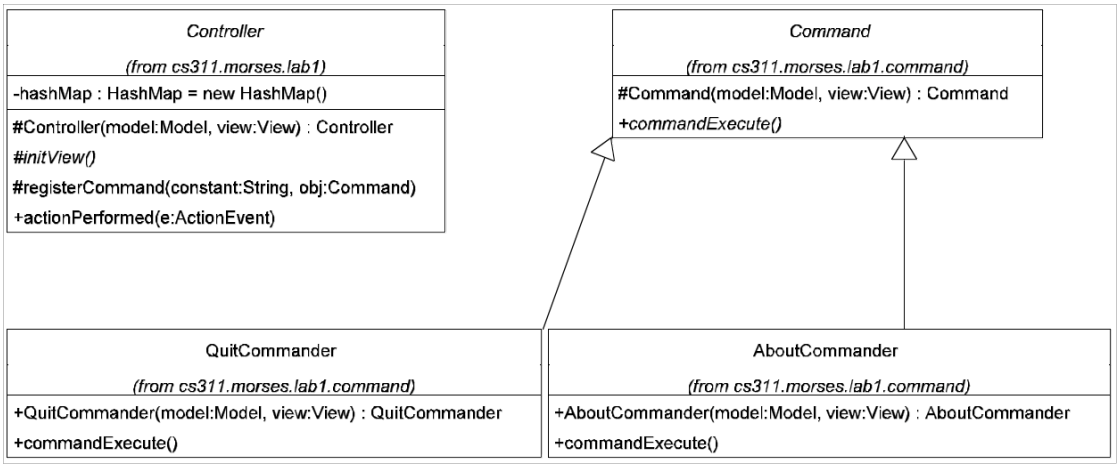

Fig. 4. Integration of the Command pattern into the *ActionEvent* handling framework.

The *Controller* has a *HashMap* object which it uses to store *Command* objects (in the *registerCommand* method, where they are associated with the constant string key). Users extend the *Command* class to implement a "Commander" object, one for each action event to be handled. In Figure 4 two common implementations are shown – one to handle Quit events and one to handle requests for About events. These implementations must provide a constructor and a *commandExecute* method. The *commandExecute* method will be invoked automatically by the *actionPerformed* method in the *Controller* as follows:

```
public void actionPerformed( ActionEvent e )
{
 String strID = e.getActionCommand();
 Command cmder = ( Command ) hashMap.get( strID );
 if ( cmder != null )
   cmder.commandExecute();
 else
   System.out.println( "Command not implemented." );
}
```

In addition to the framework enhancements mentioned above, it is easy to add functionality to the view subsystem. Examples include additional windows (views)

and classes for: an About dialog, a Status bar, a Log class (to show debugging information in customized ways), or a component showing memory usage or garbage collection details. All these additions can be made as separate classes in the view subsystem in ways that do not increase the complexity of the framework. They can be plugged in as desired.

5. DEVELOPMENT TOOLS AND TESTING

When the file and package structure is common between the instructor and all students it becomes possible to use popular development tools to simplify common tasks. One of these is the task of compiling and running applications. Many IDE's work flawlessly with the framework presented here. It is common, however, to experience difficulties when the source code is in packages. The most common cause is problems with path settings for the Java tools and the CLASSPATH used by *java*, *javac*, *jar* and *javadoc*. These problems can be greatly reduced by using a build tool. Apache Ant [1] is a build and development tool that is very powerful, easy to use, cross-platform and free. In the context of this framework, Ant is especially useful in that it works from a single XML build file. The instructor can write and distribute to students a build file containing various "targets" which can be invoked to compile or run programs, build JavaDoc files, create Jar archives, and many other tasks. Once students install Ant, an assignment can be compiled by executing *ant compileLab1* from the command line. Similarly an assignment may be compiled with documentation and jar archives created automatically (and possibly submitted on a shared filesystem) by executing a similar command.

Another benefit to the common framework is the ease of performing unit testing. A tool like JUnit [2,5,7] may be used by students to test their work and by instructors to grade assignments. The instructor may write a test case (a single java class file) and distribute it with the assignment. Students may then evaluate their code using the provided tests or add new tests themselves. Using JUnit within this framework requires students, at most, to change a single import statement in the test class. This form of testing has proven very effective in data structures courses, where students spend most of their time working in the model. The unit testing class can fully exercise their data structure implementations and also evaluate their uses in the application. In terms of grading, unit testing easily catches programming errors missed by instructors grading source code or manually testing code by running the final program.

CONCLUDING COMMENTS

When first presented with this framework, students typically respond negatively. They view it as additional work with unnecessary complications. Introductory students often have not written complicated enough programs to see the benefits afforded them by a well-engineered application. Typically, students are well adjusted after two assignments. Many continue to use it in future courses and find the framework beneficial.

When students do struggle with the framework, a common practice is to bypass it. One way to do this is to make most variables static. Then, when working in the view, they can directly access the data in the model. While accessing data in the model from the view is not always a bad practice (many MVC architectures provide a link directly

from the view to "its" model), in student programs it often results in members that must be declared public. In typical student programs, there is no need for this functionality, and the separation between the view and the model can be preserved.

A tutorial for students describing this framework in greater detail is present on the author's (Morse) website. Source code for the framework and examples of programs using it are also available. The tutorial can be reached by following links from http://www.wou.edu/~morses.

REFERENCES

[1] Apache Ant website, http://ant.apache.org/

[2] Barriocanal, E. G., Urban, M-A. S., Cuevas, I. A. and Perez, P. D. 2002. An Experience in IntegratingAutomated Unit Testing Practices in an Introductory Programming Course. *SIGCSE Bulletin 34*, 125-128.

[3] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

[4] Krasner, Glenn E., Pope, Stephen T. 1988. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3), 26-49.

[5] JUnit website, http://www.junit.org

[6] Milewski, Bartosz 2001. *C++ In Action: Industrial Strength Programming Techniques*, Addison-Wesley.

[7] Olan, Michael 2003. Unit Testing: Test Early, Test Often. *The Journal of Computing in Small Colleges 19*, 319-328.