**The Model/View Pattern**

***Introduction***

Successful development of large applications depends on the principle of divide and conquer. A large complex task is best accomplished by dividing it into simpler, cooperating subtasks. However, we do not take full advantage of this precept if we mix code for controlling an application's interface with code for managing an application’s underlying data. Consequently, computer professionals frequently break large programs into a model and a view. The view handles the interface and communicates with a model that manages the application's data.

This approach simplifies the task of writing complex applications and increases their maintainability. It is common for users to request changes to an application’s interface, so it is advantageous to make these changes without getting entangled in the intricacies of the model. Similarly, changes can be made to the model without worrying about the interface. The separation of model and view is also beneficial when an application requires several windows. A separate class supports each window, or view, and all views communicate with a common model.

In addition, on large projects teams of programmers can work independently on the view and the model, provided they agree ahead of time on the public methods included in the model.

***Responsibilities of the model and view***

In general, when we separate an application into a model and view, the responsibilities of the view are as follows:

|  | Instantiate and arrange the window objects. |
| --- | --- |
|  | Instantiate and initialize the model. |
|  | Handle user-generated events such as button clicks and menu selections by sending messages to the model. |
|  | Accurately represent the model to the user. |

The responsibilities of the model are as follows:

|  | Define and manage the application’s data (this usually requires coordinating the activities of several programmer-defined classes). |
| --- | --- |
|  | Respond to messages from the view. |

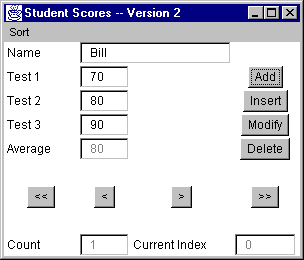
The division of labor between the model and the view has proven itself useful in many different situations and is called a ***pattern***. There are many other patterns in the realm of object-oriented programming. Each describes how a common programming situation can be handled by a collection of classes with predefined roles communicating in a predefined manner.

***Example***

As an example of the mode/view pattern, we present an application that maintains an array of student data. There are three classes:

|  | **StudentTestScoreView**, which manages the interface |
| --- | --- |
|  | **StudentTestScoreModel**, which supports manipulations of an array of student objects. |
|  | **Student**, which represents a single student |

Here is a snapshot of the interface:



The next table describes the function of each button.

| **Button Label** | **Effect** |
| --- | --- |
| Add | Creates a new student object with the data displayed and inserts it at the end of the list. The new student becomes the current student. |
| Insert | Creates a new student object with the data displayed and inserts it before the current student. The new student becomes the current student. |
| Modify | Replaces the current student's data with the data displayed. |
| Delete | Removes the current student from the list. If the list becomes empty or the deleted student was the last student, the data fields are cleared; otherwise, the next student becomes the current student. |
| << | Moves to the first student in the list and displays its data. |
| < | Moves to the previous student in the list and displays its data. |
| > | Moves to the next student in the list and displays its data. |
| >> | Moves to the last student in the list and displays its data. |

Here are listings for the three classes **StudentTestScoresView**, **StudentTestScoresModel**, and **Student**:

import java.awt.\*;

import BreezyGUI.\*;

**public class StudentTestScoresView extends GBFrame{**

// Window objects --------------------------------------

Button addButton = addButton ("Add" ,2,4,1,1);

Button insertButton = addButton ("Insert",3,4,1,1);

Button modifyButton = addButton ("Modify",4,4,1,1);

Button deleteButton = addButton ("Delete",5,4,1,1);

Label blankLine1 = addLabel ("" , 6,1,1,1);

Button firstButton = addButton ("<<", 7,1,1,1);

Button previousButton = addButton ("<", 7,2,1,1);

Button nextButton = addButton (">", 7,3,1,1);

Button lastButton = addButton (">>", 7,4,1,1);

Label nameLabel = addLabel ("Name" ,1,1,1,1);

Label test1Label = addLabel ("Test 1" ,2,1,1,1);

Label test2Label = addLabel ("Test 2" ,3,1,1,1);

Label test3Label = addLabel ("Test 3" ,4,1,1,1);

Label averageLabel = addLabel ("Average" ,5,1,1,1);

TextField nameField = addTextField ("",1,2,2,1);

IntegerField test1Field = addIntegerField (0 ,2,2,1,1);

IntegerField test2Field = addIntegerField (0 ,3,2,1,1);

IntegerField test3Field = addIntegerField (0 ,4,2,1,1);

IntegerField averageField = addIntegerField (0 ,5,2,1,1);

Label blankLine2

= addLabel ("" ,8,1,1,1);

Label countLabel

= addLabel ("Count" ,9,1,1,1);

IntegerField countField

= addIntegerField (0 ,9,2,1,1);

Label indexLabel

= addLabel ("Current Index" ,9,3,1,1);

IntegerField indexField

= addIntegerField (-1 ,9,4,1,1);

MenuItem sortByNameMI = addMenuItem ("Sort","By Name");

MenuItem sortByAverageMI = addMenuItem ("Sort","By Average");

// Other instance variables ---------------------------------

StudentTestScoresModel model;

// Constructor-----------------------------------------------

public StudentTestScoresView(){

setTitle ("Student Scores -- Version 3");

model = new StudentTestScoresModel();

averageField.setEditable (false);

countField.setEditable (false);

indexField.setEditable (false);

displayCurrentStudent();

}

// buttonClicked method--------------------------------------

public void buttonClicked (Button buttonObj){

if (buttonObj == addButton){

Student stud = getDataOnScreen();

String str = model.addStudent (stud);

if (str != null)

messageBox (str);

else

displayCurrentStudent();

}

// insert, modify, and delete are left as an exercise

else if (buttonObj == firstButton){

model.moveToFirstStudent();

displayCurrentStudent();

}

else if (buttonObj == previousButton); // left as an exercise

else if (buttonObj == nextButton){

model.moveToNextStudent();

displayCurrentStudent();

}

else if (buttonObj == lastButton); // left as an exercise

}

// menuSelected method---------------------------------------

public void menuItemSelected (MenuItem menuItemObj){

if (menuItemObj == sortByNameMI){

model.sortStudentsByName();

model.moveToFirstStudent();

displayCurrentStudent();

}

// Sorting by average left as an exercise

}

// Private methods-------------------------------------------

Student getDataOnScreen(){

String nm = nameField.getText().trim();

int[] tests = new int[Student.NUM\_TESTS];

tests[0] = test1Field.getNumber();

tests[1] = test2Field.getNumber();

tests[2] = test3Field.getNumber();

Student stud = new Student (nm, tests);

return stud;

}

void displayCurrentStudent(){

Student stud = model.getCurrentStudent();

if (stud == null){

nameField.setText ("");

test1Field.setNumber (0);

test2Field.setNumber (0);

test3Field.setNumber (0);

averageField.setNumber (0);

}else{

nameField.setText (stud.getName());

test1Field.setNumber (stud.getScore(1));

test2Field.setNumber (stud.getScore(2));

test3Field.setNumber (stud.getScore(3));

averageField.setNumber (stud.getAverage());

}

countField.setNumber (model.getStudentCount());

indexField.setNumber (model.getIndexSelectedStudent());

}

public static void main (String[] args){

Frame frm = new StudentTestScoresView();

frm.setSize (400, 250);

frm.setVisible(true);

}

}

**public class StudentTestScoresModel extends Object{**

// Instance variables ---------------------------------

Student[] students = new Student[10];

int indexSelectedStudent;

int studentCount;

// Constructor-----------------------------------------------

public StudentTestScoresModel(){

indexSelectedStudent = -1;

studentCount = 0;

}

public String addStudent (Student stud){

if (studentCount == students.length)

return "SORRY: student array is full";

String str = stud.validateData();

if (str != null)

return str;

students[studentCount] = stud;

indexSelectedStudent = studentCount;

studentCount++;

return null;

}

public Student getCurrentStudent(){

if (indexSelectedStudent == -1)

return null;

else

return students[indexSelectedStudent];

}

public int getIndexSelectedStudent(){

return indexSelectedStudent;

}

public int getStudentCount(){

return studentCount;

}

void moveToFirstStudent(){

if (studentCount == 0)

indexSelectedStudent = -1;

else

indexSelectedStudent = 0;

}

void moveToPreviousStudent(){

// Exercise

}

void moveToNextStudent(){

if (studentCount == 0)

indexSelectedStudent = -1;

else

indexSelectedStudent

= Math.min (studentCount - 1, indexSelectedStudent + 1);

}

void moveToLastStudent(){

// Exericse

}

void sortStudentsByName(){

for (int i = 0; i < studentCount - 1; i++){

String namei = students[i].getName();

for (int j = i + 1; j < studentCount; j++){

String namej = students[j].getName();

if (namei.compareTo (namej) > 0){

Student temp = students[i];

students[i] = students[j];

students[j] = temp;

}

}

}

}

}

**public class Student extends Object{**

public final static int NUM\_TESTS = 3;

private final static int MIN\_SCORE = 0;

private final static int MAX\_SCORE = 100;

private String name;

private int[] tests = new int[NUM\_TESTS];

public Student(){

name = "";

for (int i = 0; i < NUM\_TESTS; i++)

tests[i] = 0;

}

public Student(String nm, int[] t){

name = nm;

for (int i = 0; i < NUM\_TESTS; i++)

tests[i] = t[i];

}

public Student(Student s){

name = s.name;

for (int i = 0; i < NUM\_TESTS; i++)

tests[i] = s.tests[i];

}

public void setName (String nm){

name = nm;

}

public String getName (){

return name;

}

public void setScore (int i, int score){

// where 1 <= i <= NUM\_TESTS

tests[i - 1] = score;

}

public int getScore (int i){

// where 1 <= i <= NUM\_TESTS

return tests[i - 1];

}

public int getAverage(){

int sum = 0;

for (int i = 0; i < NUM\_TESTS; i++)

sum += tests[i];

return sum / NUM\_TESTS;

}

public int getHighScore(){

int highScore;

highScore = tests[0];

for (int i = 1; i < NUM\_TESTS; i++){

highScore = Math.max (highScore, tests[i]);

}

return highScore;

}

public String toString(){

String str;

str = "Name: " + name + "\n";

for (int i = 0; i < NUM\_TESTS; i++){

str += "tests " + i + ": " + tests[i] + "\n";

}

str += "Average: " + getAverage();

return str;

}

public String validateData(){

if (name.equals ("")) return "SORRY: name required";

for (int i = 0; i < NUM\_TESTS; i++){

if (tests[i] < MIN\_SCORE || tests[i] > MAX\_SCORE){

String str = "SORRY: must have "+ MIN\_SCORE

+ " <= test score <= " + MAX\_SCORE;

return str;

}

}

return null;

}

}