

Defect, Problem, and Progress Tracking System

John Biddle
Physics/Electrical Engineering
Harvard University
Cambridge, MA 02138

August 9, 2002

Project Supervisor: Jerzy Nogiec

TABLE OF CONTENTS

ABSTRACT.....	3
INTRODUCTION.....	3
ANALYSIS OF DPPTS INTERFACE V1.5.2.....	4
DESIGN OF DPPTS V2.0	8
IMPLEMENTATION OF V2.0	10
<i>ViewPanel</i>	<i>11</i>
<i>MessagePanel</i>	<i>12</i>
<i>QueryPanel</i>	<i>13</i>
<i>DataTableModel</i>	<i>16</i>
<i>ActionPanel.....</i>	<i>17</i>
<i>SearchFrame.....</i>	<i>19</i>
<i>EnterDataFrame.....</i>	<i>21</i>
<i>HistoryFrame.....</i>	<i>24</i>
<i>DescribeFrame.....</i>	<i>26</i>
<i>Emailer.....</i>	<i>27</i>
<i>DbAccessor</i>	<i>29</i>
<i>DataManager.....</i>	<i>30</i>
CONCLUSION	36
REFERENCES	40
ACKNOWLEDGEMENTS	40
APPENDIX A: APPLETFRAME CLASS	41
APPENDIX B: EMAIL.CGI.....	43

Abstract

The Defect, Problem and Progress Tracking System allows the Systems Development and Support Group at the Magnet Test Facility to keep track of reported problems or requests concerning the various systems administered there. The applet that provides an interface for searching, creating and updating defect entries in the defect tracking database is in need of upgrade. However the source code for the applet cannot be changed directly since it was generated by an outdated application builder. Therefore upgrading the Defect, Problem, and Progress Tracking System applet requires implementing the interface from scratch.

Introduction

One of the main goals of the Systems Development and Support Group (SDSG) is to develop and maintain the various control, monitoring and data acquisition systems operated by the Magnet Test Facility (MTF) to test both conventional and superconducting accelerator magnets. Efficient maintenance of these complex systems requires an understanding of the systems' behaviors, knowledge of any problems encountered in the systems, and a mechanism to improve functionality if needed. However keeping track of all observed problems in this large collaborative environment would be a difficult task without a specialized system to support it. To this end the Defect, Problem, and Progress Tracking System (DPPTS) was developed.

DPPTS provides an interface that allows a user to browse through, create, and update problem or request entries for MTF systems. Each problem entry should contain the following fields:

Problem ID	Used to uniquely identify a problem entry.
Title	
Event Date	
Category:	The category that best describes the environment of the problem (i.e. network, peripherals, software, etc).
System:	Where the problem occurred.
Status:	Current status of the problem: open, closed, or deferred.
Type:	Type of request in handling the problem/request.
Severity:	Priority that should be given to the problem/request.
Description:	Description of the problem/request.
Commentor:	Person addressing the problem (usually the user).
Assigned To:	Person assigned to handle the problem.
Deadline:	Date the user expects the problem to be handled by.

When a problem entry is created, the entry is added to a networked database, which may be accessed by remote nodes using DPPTS (see figure 1). And with the information provided in this database, the SDSG can easily keep track of the creation and progress of various problems, requests, and action items for MTF system maintenance. Thus DPPTS is a very powerful tool in monitoring system behavior and upkeep in MTF.

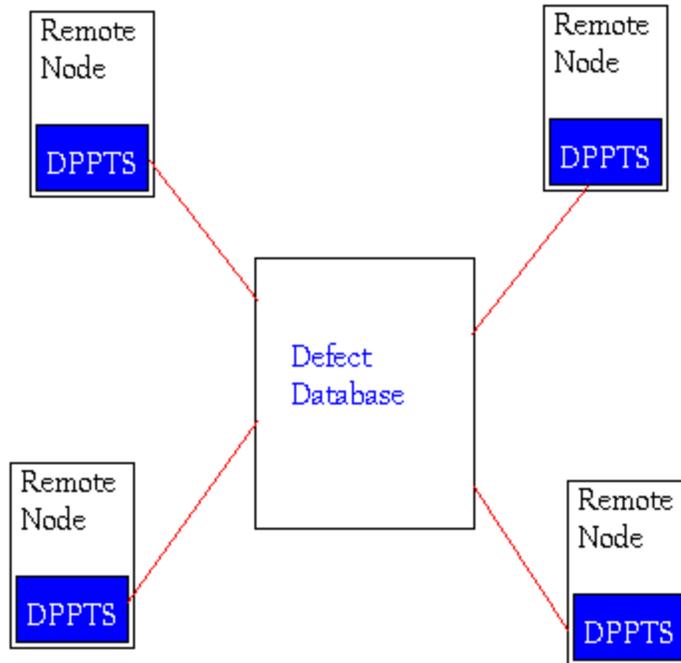


Figure 1: Basic Idea of DPPTS Interfacing with Remote Nodes

Analysis of DPPTS Interface V1.5.2

The interface provided by DPPTS version 1.5.2 is in the form of an applet on the SDSG website. After initialization, the applet should have the appearance shown in figures 1. The applet is divided into 4 panels, each of which has its own individual tasks: the query panel (left), the view panel (right), the action panel (top), and the message panel (bottom).

Defect, Problem, & Progress Tracking V1.5.2

Update
New
History

Search Options						
ID	Date	System	Status	Category	Title	
237	1980-01-04 14:26:00.0	PC Support	closed	Configuration	Defective date due to Y2K problem	
136	1997-07-02 10:55:00.0	DMCS	closed	In-house Software	Hardware test	
133	1998-01-07 10:18:00.0	DMCS	closed	In-house Software	slow scan	
138	1998-02-07 11:13:00.0	DMCS	deferred	In-house Software	Video camera	
168	1998-03-03 13:49:00.0	DMCS	closed	In-house Software	Database model	
167	1998-03-04 13:44:00.0	DMCS	closed	In-house Software	Requirements specification	
171	1998-05-07 10:57:00.0	DMCS	closed	In-house Software	Prototype Positioning System	
135	1998-07-16 10:27:00.0	DMCS	closed	In-house Software	Main menu	
170	1998-08-01 14:01:00.0	DMCS	deferred	In-house Software	Refined data model.	
169	1998-08-10 13:54:00.0	DMCS	closed	In-house Software	Prototype of GUI.	
131	1998-09-23 10:03:00.0	DMCS	closed	In-house Software	Remote File System (RFS)	
134	1998-09-23 10:22:00.0	DMCS	closed	In-house Software	CCS	
132	1998-10-09 10:14:00.0	DMCS	closed	In-house Software	Scribe	
225	1998-10-26 10:41:00.0	PC Support	closed	Network	131.225.47.128 subnet installation	
144	1998-11-20 11:30:00.0	DMCS	closed	In-house Software	Documentation of calculations	
137	1998-12-23 11:02:00.0	DMCS	closed	In-house Software	Calibrations entry application	
148	1999-01-14 08:56:00.0	EMS	closed	In-house Software	Project organization: development	

View results of your query

Figure 2: Defect Tracking Applet

The message panel is the simplest of the four panels. Its job is simply to display the current status of any queries or transactions being sent to the database. If an error has occurred during the process, the message panel will display an appropriate error message. Otherwise, it will display an appropriate “success” message.

The query panel displays all of the search options used to build a defect query. The 4 combo boxes at the top of the panel allow a user to limit a search by category, system, request type, and/or person. Below the combo boxes, there are 4 check boxes that allow a user to search for only open or closed problems, problems that have been made or updated in the last month, and/or problems with emergency status. And below these boxes is a text field that allows a user to search for a problem with a specific problem id number. When the user presses the submit button, all the selected parameters mentioned above are conjoined by AND’s to construct a query.

When a user submits a query, the results are displayed as a table in the view panel. Due to limited space, however, the table displays only the following fields: problem ID, date, system, status, category and title. The table is somewhat interactive in that it allows the user to select a row corresponding with a particular problem entry and may either update the entry or view the history of this entry by pressing the appropriate buttons in the action panel.

Figure 3: New Entry Frame

The action panel offers a few extra options. The “New” and “Update” buttons on the action panel play similar roles. When pressed, these buttons open new windows containing several text and combo box fields to enter and/or change data (see figure 2 and 3). The “History” button opens a new window containing a table of all updates made to the currently selected entry in the view panel (see figure 4). When an entry is selected in the history table, the full description of the entry is displayed in the text area. Note that the history frame also contains an “Update” button to allow the user to add another update. The “Description” button in the history frame brings up another window, which displays all the entries in the history table in text format (see figure 5). The “Email” button in this frame brings up a dialog window asking for the user’s e-mail address. Once the user enters his/her e-mail address, the text-formatted entries are emailed to the given address.

Updating Original Problem

Dismiss Reset Apply

Date: 2002-7-31 10:50 ID: 237 Category: Configuration

System: PC Support Status: Closed Commentor: Unknown Commentor

Type: defect Severity: Low Assigned To: Ping Wang

Description:

Warning: Applet Window

Figure 4: Update Entry Frame

History of all Comments

Descripti... Update Dismiss

ID	Date	Commentor	Status	Severity	Type	AssignedTo	Description
237	1980-01-04 14:26:0...	Dana Walbridge	open	low	defect	Jerzy Nogiec	The clock is also slow. Jerzy will deter...
237	2000-01-04 14:15:0...	Dana Walbridge	open	low	defect	Not Assigned	Jerzy's laptop has a Y2K problem with t...
237	2001-09-27 15:16:0...	Ping Wang	closed	low	defect	Ping Wang	The clock was adjusted. Office 97 was ...

Description

The clock is also slow. Jerzy will determine what course of action to take.

The initial commentor was Dana Walbridge, who did enter his name as the commentor, and was observed by three witnesses of sound mind. It's not yeat clear why the commentor was not included in the first entry for this problem.

Warning: Applet Window

Figure 5: History Frame

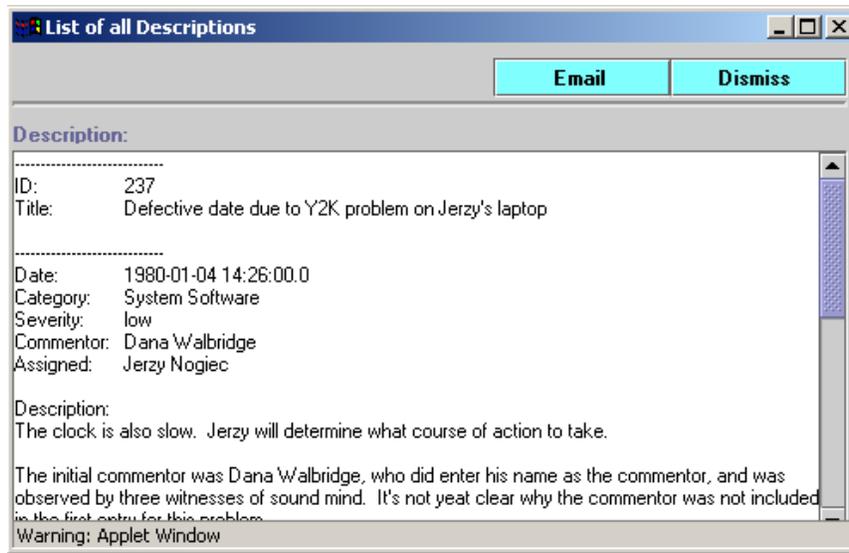


Figure 6: Description Frame

To recap DPPTS serves to meet three main goals: effective database querying, problem entry updates, and new problem entry creation. DPPTS version 1.5.2 meets all of these goals, but there is plenty of room for improvement. Version 1.5.2's querying options are quite limited and should be augmented in order to allow more effective database querying. Some simple improvements to DPPTS would be to allow a keyword search field or to allow a user to search for entries on specific dates.

Making such improvements version 1.5.2 should not be a far too difficult task if version 1.5.2 was coded well. However the code for version 1.5.2 was generated by a Java application builder, PowerJ, giving version 1.5.2 several undesirable characteristics. One such characteristic is that the source code for version 1.5.2 is very obscure and unintuitive, making it very difficult to make changes to the code without using PowerJ. Another such characteristic is that the code is heavily dependent on PowerJ implemented classes. This means that version 1.5.2 cannot run in a standard Java runtime environment without these PowerJ libraries present, and DPPTS cannot advance with the latest Java Runtime Environments unless PowerJ does. But PowerJ, however, is no longer continued. Thus in order to make improvements on DPPTS and take away its dependence on PowerJ, the code for the next version of DPPTS must be written from scratch.

Design of DPPTS V2.0

To begin the design phase of this project, a basic model of components needs to be established. The framework of version 1.5.2 is used as a foundation for the newer version to aid in the design phase. Thus the appearance of the newer version of DPPTS should be quite similar to that of version 1.5.2.

Recall that the DPPTS applet consisted of 4 panels (action, message, query, and view panel) and 4 frames (new entry, update entry, history, and description frame). For the newer version, another frame is added to the list to include advanced search options. Each of these components can be given its own Java class. By using a class for each component, an individual component can be constructed independently from the rest, making it easy to make changes to one component without changing the others. However, the new entry and update entry frames behave largely in the same way, so these two frames can be constructed from the same class. Also several of these classes will need to send and receive data to and from the database. Therefore it would be useful to create classes that will handle the details of interacting with the database. For this cause are the DbAccessor and the DataManager classes constructed. The DbAccessor will be responsible for making the actual connection to the database, and the DataManager will be responsible for sending requests to the DbAccessor and parsing the results. Thus the DataManager is the only class expected to have an instance of the DbAccessor class. In this way, a class only needs an instance of the DataManager to send and receive data to and from the database.

The panel classes are all subclasses of the Java swing component, JPanel, and each is largely independent of each other (see figure 5). The QueryPanel constructs the queries to the database, the ViewPanel displays the result table, the ActionPanel instantiates the new, update, and history frames when appropriate, and the MessagePanel displays information on the status of any requests sent to the database. However since the panels are constructed independently of each other, the panels must be able to communicate with each other to some degree. Since the MessagePanel is basically concerned with the doings of the DataManager alone, the DataManager can be given a MessagePanel object to display messages to the panel. The Action, View, and Query panels, however, need access to the same table object (an instance of the Java swing class JTable), so instead of creating the JTable in the view panel, the JTable is created outside the panels (in the Applet object containing these panels) and the constructors for each of these panels is given the same JTable object. In this way, any changes made to the JTable by one panel can be seen by the other panels. This is the level of communication needed for the applet to work properly.

The frame classes are all subclasses of the Java swing component, JFrame (see figure 6). These classes are, for the most part, straightforward in implementation. The subtleties, however, deal with how information is passed to the frames. The history, description, and update entry frames require information about a particular entry. Also the new and update entry frames are instances of the same class, EnterDataFrame, meaning that this class must be able to distinguish between the two cases. To meet this end, each frame class has a constructor that requires information about the defect entry in question as an argument. In this way, since the new defect entry frame does not require any information, the EnterDataFrame class can distinguish between the two by checking for a null argument to its constructor.

Several classes will be using a DataManager object to access the database, however it will not be very efficient if there are several DataManager objects created to access the

database. To this end the DataManager is made a singleton class. This means that only one instance of the DataManager will exist during run time, and any object needing to use DataManager's methods and members must obtain this instance. This instance will be available to other objects through a static method in DataManager.

Figure xx displays the various classes used in the design and their dependencies.

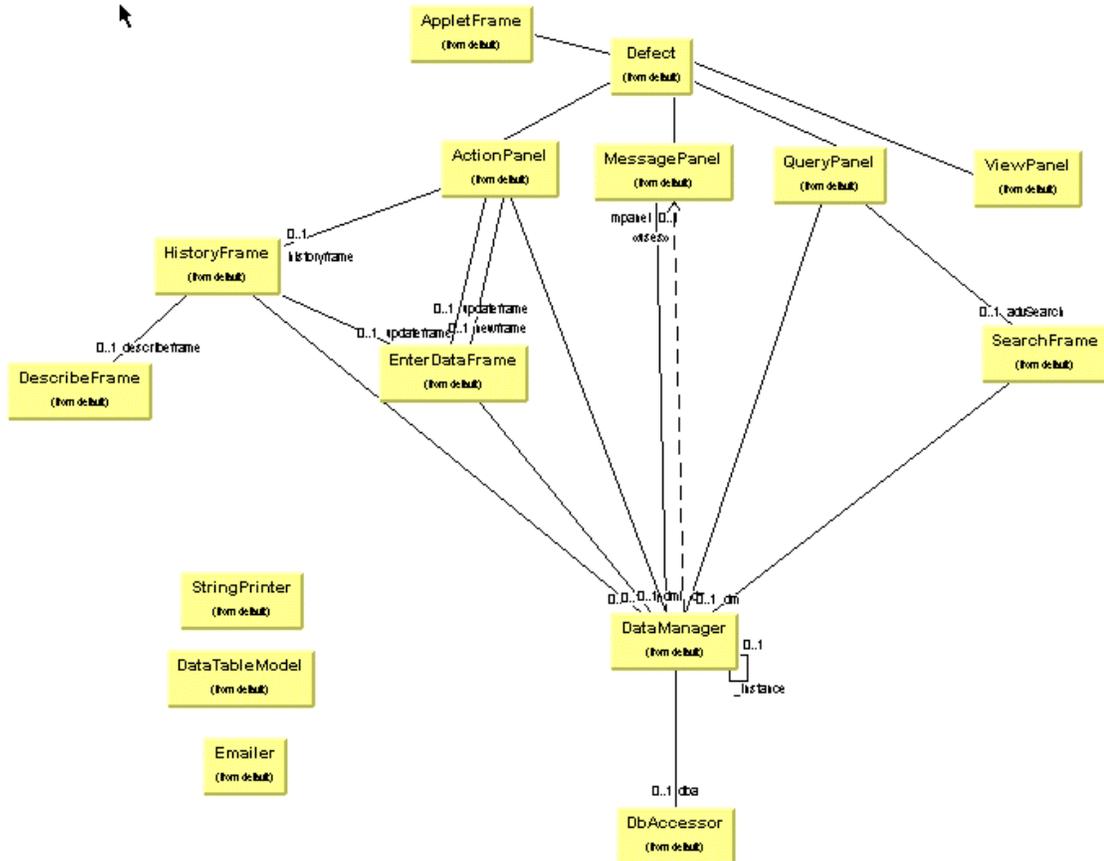


Figure 7: Diagram of Class Dependencies in V2.0

Implementation of V2.0

The descriptions above should give a basic framework on how the components of the DPPTS applet should interact with each other. Now it is appropriate to give more detail on how the applet is built.

Defect

The Defect class extends JApplet and is the parent of all the panel classes. Its init() method, (whose function is similar to that of a constructor) simply needs to instantiate the panel classes and add them to the applet's content pane in some sort of layout. But recall that the panels need to communicate with each other by sharing the same instance of JTable, so the init() method also creates a JTable object and passes this object to the

view, query, and action panels' constructors. Also the singleton DataManager is given the instance of MessagePanel by calling on the DataManager's static method, setMsgPanel().

In most cases, the init() method is the only method that needs to be defined for an applet to function; however, the Defect class also defines 3 static methods. The first, getProperties(), returns a Properties object (similar to a Hashtable) that contains the applet parameters "driver" and "dataSource." The second, getBase(), returns the URL address of the applet. The third is a main() method that allows the applet to be run as an application.

The code for the Defect class is not very long and may be seen here:

```
23 public class Defect extends JApplet {
24     private static Properties props = new Properties();
25     private static URL base;
26
27     public void init() {
28         base = getCodeBase();
29         props.put("driver", getParameter("driver"));
30         props.put("dataSource", getParameter("dataSource"));
31         JTable table = new JTable();
32         MessagePanel mpanel = new MessagePanel();
33         DataManager.setMsgPanel(mpanel);
34         QueryPanel qpanel = new QueryPanel(table);
35         ViewPanel vpanel = new ViewPanel(table);
36         ActionPanel apanel = new ActionPanel(table);
37         getContentPane().setLayout(new BorderLayout());
38         getContentPane().add(mpanel, BorderLayout.SOUTH);
39         getContentPane().add(apanel, BorderLayout.NORTH);
40         getContentPane().add(qpanel, BorderLayout.WEST);
41         getContentPane().add(vpanel, BorderLayout.CENTER);
42     }
43
44     public static Properties getProperties() {
45         return props;
46     }
47
48     public static void main(String [] args) {
49         new AppletFrame(new Defect(), 720, 375);
50     }
51
52     public static URL getBase() {
53         return base;
54     }
55 }
```

All of the functionality of the applet is hidden within the panel classes.

ViewPanel

The ViewPanel simply takes the JTable given to it in its constructor and displays it. Therefore the code for the ViewPanel is not very long either:

```

14 public class ViewPanel extends JPanel{
15     private JTable table;
16     private JScrollPane jsp;
17
18     /* Creates a ViewPanel with the given table.
19      * @param table The table to be displayed.
20      */
21     public ViewPanel(JTable table) {
22         this.table = table;
23         setLayout(new BorderLayout());
24         jsp = new JScrollPane(table);
25         add(jsp, BorderLayout.CENTER);
26     }
27
28 }

```

MessagePanel

The MessagePanel's purpose is to display an appropriate message in the applet concerning the current status of the DataManager. The DataManager, however, is responsible for determining the appropriate message. Thus the MessagePanel only needs to worry about taking a message and displaying it. So the MessagePanel defines three methods. The first, setMessage(), takes a string as the message to be displayed. The second, setColor(), changes the color of the message to the specified Color. And last, the paint() method displays the message on the panel.

Again, the code for the MessagePanel is not very long and may be seen here:

```

13 public class MessagePanel extends JPanel {
14     private String msg;
15     private Color color;
16
17     /**
18      * Creates a MessagePanel.
19      */
20     public MessagePanel() {
21         setPreferredSize(new Dimension(720, 15));
22         msg = "Message Panel";
23         color = Color.black;
24         setBackground(Color.green);
25     }
26
27     /**
28      * Changes the drawn message.
29      * @param str the new message.
30      */
31     public void setMessage(String str) {
32         msg = str;
33         repaint();
34     }
35
36     /**
37      * Draws the Panel.
38      * @param g Graphics object that renders panel.
39      */
40     public void paint (Graphics g) {
41         g.setColor(Color.lightGray);
42         g.fillRect(0,0,getSize().width, getSize().height);
43         g.setColor(color);
44         g.drawString(msg, 0, 12);
45     }
46
47     /**
48      * Sets the color of the message.
49      * @param color new color for the message.
50      */
51     public void setColor(Color color) {
52         this.color = color;
53         repaint();
54     }
55 }

```

QueryPanel

The QueryPanel performs one of two actions. When the “submit query” button is pressed, the QueryPanel constructs a query, sends it to the DataManager, retrieves the results, and changes the table model to reflect the given results. Also, when the “advanced search” button is pressed, the QueryPanel makes a SearchFrame available. So the QueryPanel needs to implement the ActionListener interface. This interface allows the QueryPanel to listen for ActionEvents generated by pressing a button.

There are several tasks that need to be accomplished in QueryPanel’s constructor. The panel components (JButtons, JComboBoxes, JTextFields, etc) need to be instantiated and added to the panel in an appropriate layout (in this case, a GridLayout). The QueryPanel also needs to be added to the JButton’s ActionListener lists using JButton’s method, addActionListener(). And lastly, the fields of the JComboBoxes need to be retrieved from the DataManager using the appropriate getXXX() methods. The first two tasks are rather simple and do not take up much code, but the third task, however, is not so easy and generates a SecurityException. Apparently the DataManager (well actually the

DbAccessor) cannot be accessed in the initialization thread of the panel (which would normally be the initialization thread of the Defect applet as well), so a solution to this problem is to retrieve the fields in another thread. Thus the QueryPanel also implements the Runnable interface. This interface allows the implementing class's run() method to run in a separate thread through a Thread object. So the QueryPanel defines a run() method that retrieves the necessary fields, and the constructor begins a new thread by creating a Thread object:

```

3 public class QueryPanel extends JPanel implements ActionListener,
4   Runnable {
5   // Data Members
6   private JTable table;
7   private DataManager dm = DataManager.getInstance();
8
9   public QueryPanel(JTable table) {
10    this.table = table;
11
12    /* instantiate components ...
13       ..
14       ..
15       ..
16    add ActionListener ... */
17
18    //retrieve fields
19    new Thread(this).start();
20  }
21
22  //For Runnable interface
23  public void run() {
24    // retrieve fields
25    // dm.getXXX();
26  }
27
28  //For ActionListener interface
29  public void actionPerformed(ActionEvent e) {
30    //Handle Action Events
31  }
32 }

```

When either of the buttons is pressed (e.g. “advanced search” or “submit query”), the actionPerformed() method is called with an(ActionEvent) object that tells the source of the event. When the “advanced search” button is pressed, the panel checks to see if a SearchFrame has already been created. If so, that frame is reset and set visible again, and if not, a new SearchFrame is created and set visible:

```

30 private SearchFrame advSearch;
31 private JButton advanced;
32
33 public void actionPerformed(ActionEvent e) {
34   //Advanced Search
35   if (e.getSource() == advanced) {
36     //Display a SearchFrame
37     if (advSearch == null) {
38       advSearch = new SearchFrame(table);
39     } else {
40       advSearch.reset();
41     }
42     advSearch.setVisible(true);
43   }
44 }

```

When the “submit query” button is pressed, two cases need to be distinguished. The first case is when the user has entered a problem ID number to be searched. This type of query is specific, and thus needs a different method from the DataManager (dm) than the basic query:

```
31     private JButton advanced, submit;
32     private JTextField probID;
33
34     public void actionPerformed(ActionEvent e) {
35         //Advanced Search
36         if (e.getSource() == advanced) {
37             //Advanced Search
38         } else if (e.getSource() == submit) {
39             if (probID.getText().length() > 0) {
40                 //Error Check
41                 try {
42                     Integer.parseInt(probID.getText());
43                     synchronized(dm) {
44                         result = dm.idQuery(probID.getText());
45                     }
46                 } catch (NumberFormatException ne) {
47                     JOptionPane.showMessageDialog(null,
48                         "Problem ID must be numeric.", "Error",
49                         JOptionPane.ERROR_MESSAGE);
50                 }
51                 return;
52             }
53         }
54     }
```

Note the call to Integer’s static method, `parseInt()`, and the try/catch block surrounding it and the call to the DataManager. This is simply to insure that the text entered in the JTextField is an integer. If it is not, an error dialog is made. Note also the synchronized block around the DataManager object. This block is used to make sure the QueryPanel is the only object accessing the singleton DataManager at the time. Also note that the code shown here is not completely correct. The `idQuery()` method may throw an Exception and there is not an appropriate catch statement to handle it (see discussion on DataManager).

In the case of a basic query, the QueryPanel needs to construct a query out of all the parameters set in the components of panel. There are 7 query panel components (excluding the text field), and thus there are 7 possible parameters. Instead of having a method in DataManager that takes 7 arguments, the QueryPanel creates a Hashtable and places the query parameters in the Hashtable with appropriate keys and passes this Hashtable to the DataManager:

```

28 public void actionPerformed(ActionEvent e) {
29     Vector result[];
30     if (e.getSource() == advanced) {
31         //Advanced Search
32     } else if (e.getSource() == submit) {
33
34         if (probID.getText().length() > 0) {
35             //Problem ID Query
36         } else {
37             Hashtable submission = new Hashtable();
38             if (system.getSelectedIndex() != 0) {
39                 submission.put("System",
40                     system.getSelectedItem());
41             }
42             if (categories.getSelectedIndex() != 0) {
43                 submission.put("Category",
44                     categories.getSelectedItem());
45             }
46             /* etc */
47             synchronized(dm) {
48                 result = dm.query(submission);
49             }
50         }
51     }
52 }

```

The result returned by DataManager is a 2-element array of Vectors. The first element Vector contains the column names of the result table. The second element Vector is a Vector of Vectors (2-D Vector) containing the row data of the result table. This data is converted to a TableModel by using the DataTableModel class (see discussion on DataTableModel). So the table's TableModel is set to the DataTableModel, changing the data shown by the table to that returned by the DataManager:

```

26 private JComboBox system, categories;
27 private JTextField probID;
28 private JButton submit, advanced;
29
30 public void actionPerformed(ActionEvent e) {
31     Vector result[];
32     if (e.getSource() == advanced) {
33         //Advanced Search
34     } else if (e.getSource() == submit) {
35         if (probID.getText().length() > 0) {
36             //...
37             result = idQuery(probID.getText());
38         } else {
39             //...
40             result = dm.query(submission);
41         }
42     }
43     DataTableModel tm =
44         new DataTableModel(result[1], result[0]);
45     table.setModel(tm);
46     table.repaint();
47 }

```

The change made to the table here in the QueryPanel will be seen in the table displayed in the ViewPanel.

DataTableModel

The DataTableModel class is a simple helper class that creates an object implementing the TableModel interface given table data in the form of Vectors:

```
15 public class DataTableModel extends AbstractTableModel {
16     //Data Members
17     private Vector data;
18     private Vector headers;
19
20     public DataTableModel(Vector data, Vector headers) {
21         this.data = data;
22         this.headers = headers;
23     }
24
25     public int getColumnCount() {
26         return headers.size();
27     }
28
29     public int getRowCount() {
30         return data.size();
31     }
32
33     public String getColumnName(int col) {
34         return (String) headers.elementAt(col);
35     }
36
37     public Class getColumnClass(int col) {
38         return String.class;
39     }
40
41     public boolean isCellEditable(int row, int col) {
42         return false;
43     }
44
45     public Object getValueAt(int row, int col) {
46         return ((Vector) data.elementAt(row)).elementAt(col);
47     }
48
49     public void setValueAt(Object value, int row, int col) {
50         ((Vector) data.elementAt(row)).setElementAt(value, col);
51         fireTableRowsUpdated(row, row);
52     }
53 }
```

The class extends AbstractTableModel which is simply an abstract class implementing the TableModel interface. Several of the methods necessary to implement the interface are defined in AbstractTableModel. The only methods that need to be overridden are those defined in DataTableModel. The constructor takes two Vectors as arguments. The first is expected to hold all the row data as a 2-D Vector. The second is expected to hold the column names. The methods defined are straightforward and part of the interface.

ActionPanel

The ActionPanel has four buttons, each performing different actions. The “new” button brings up a new entry frame. The “update” button brings up an update entry frame. The “history” button brings up a history frame. And the “print” button converts the table data into a string and sends it to a printer. Thus the ActionPanel implements the ActionListener interface. The constructor for the ActionPanel is rather simple:

```

3 public class ActionPanel extends JPanel implements ActionListener {
4     //Data Members
5     private JButton newb;
6     private JButton update;
7     private JButton history;
8     private JButton print;
9     private JTable table;
10    private DataManager dm = DataManager.getInstance();
11
12    public ActionPanel(JTable table) {
13        this.table = table;
14        newb = new JButton("New");
15        update = new JButton("Update");
16        history = new JButton("History");
17        print = new JButton("Print");
18        add(new JLabel("Defect, Problem, & Progress Tracking V2.0"));
19        newb.addActionListener(this);
20        update.addActionListener(this);
21        history.addActionListener(this);
22        print.addActionListener(this);
23        add(newb);
24        add(update);
25        add(history);
26        add(print);
27    }
28
29    public void actionPerformed(ActionEvent e) {
30        //Action Events
31    }
32 }

```

The “new” button is pressed is the simplest of cases. The ActionPanel simply creates a EnterDataFrame or resets one without any data:

```

29     private EnterDataFrame newframe;
30
31     public void actionPerformed(ActionEvent e) {
32         if (e.getSource() == newb) {
33             if (newframe == null) {
34                 newframe = new EnterDataFrame();
35             } else {
36                 newframe.reset();
37             }
38             newframe.setVisible(true);
39         }
40     }

```

The “update” and “history” cases are slightly more complicated since data needs to be taken from the JTable. First off, the index of the selected row in the table is found using JTable’s getSelectedRow() method. Next, the problem ID number of the selected row is retrieved using getValueAt(). With the problem ID number, the desired data can be retrieved from the DataManager to be passed to the history and update frames (see discussion on DataManager):

```

85     private EnterDataFrame updateframe;
86
87     public void actionPerformed(ActionEvent e){
88         if (e.getSource() == newb) {
89             //New Entry Frame
90         } else if (e.getSource() == update) {
91             //Get selected Entry's ID number
92             int selectedRow = table.getSelectedRow();
93             //retrieve ID number as String
94             Hashtable data = dm.getEntryData(id);
95             if (data == null) {
96                 return;
97             }
98             if (updateframe == null) {
99                 updateframe = new EnterDataFrame(data);
100            } else {
101                updateframe.reset(data);
102            }
103            updateframe.setVisible(true);
104        } else if (e.getSource() == history) {
105            //similar to update case
106        }
107    }

```

In the print case, the ActionPanel takes the table model and extracts the data into a string using a couple of nested for loops. Then this string is used to create a StringPrinter object that will send the string to a printer when its print() method is called (the StringPrinter class is messy and not very interesting, and thus not discussed in detail):

```

31     public void actionPerformed(ActionEvent e) {
32         if (e.getSource() == newb) {
33             //newframe
34         } else if (e.getSource() == update) {
35             //update frame
36         } else if (e.getSource() == history) {
37             //history frame
38         } else if (e.getSource() == print) {
39             if (table.getRowCount() <= 0) {
40                 return;
41             }
42             TableModel tm = table.getModel();
43             int colCount = tm.getColumnCount();
44             int rowCount = tm.getRowCount();
45             StringBuffer doc = new StringBuffer();
46             for (int i = 0; i < rowCount; i++) {
47                 for (int j = 0; j < colCount; j++) {
48                     doc.append(tm.getColumnName(j) + ": ");
49                     doc.append(tm.getValueAt(i, j) + "\n");
50                 }
51             }
52             StringPrinter sp = new StringPrinter(doc.toString());
53             sp.print();
54         }
55     }

```

SearchFrame

The SearchFrame class behaves largely like the QueryPanel. When the “submit” button is pressed, the SearchFrame sends a query to the DataManager and displays the results in the table. Thus the SearchFrame must also be instantiated with the same JTable the panels communicate with as an argument.

The constructor for the SearchFrame is considerably longer than the QueryPanel since the SearchFrame components are arranged in a GridBagLayout – a complicated layout that is very flexible, but difficult to use. There’s nothing special about its constructor except for the appearance of 3 JTextFields instead of one, and 7 JComboBoxes instead of 6. In the SearchFrame the “persons” search in the basic query is divided into two searches: “assigned to” field search, and “commentor” field search. The 3 JTextFields are used for more specific searching. 2 JTextFields are used for a date search: one for a start date, and the other for an end date. The third JTextField is used for a case-sensitive keyword search.

The SearchFrame also implements the ActionListener interface and listens for actions generated by three separate JButtons. The “dismiss” button closes the frame (this action does not destroy the SearchFrame object). The “reset” button calls the reset() method that resets all the search fields to their initial values. And the “submit” button sends a query. Looking at the actionPerformed() method, the “reset” and “dismiss” cases are short and simple:

```

56     private JButton dismiss, reset, submit;
57
58     public void actionPerformed(ActionEvent e) {
59         if (e.getSource() == dismiss) {
60             setVisible(false);
61         } else if (e.getSource() == reset) {
62             reset();
63         } else if (e.getSource() == submit) {
64             //submit query
65         }
66     }

```

For the submit case, the SearchFrame needs to make sure the dates entered in the startDate and endDate JTextFields are a valid format. To this end the Java class, SimpleDateFormat is used to create a format standard of the form “yyyy-MM-dd”:

```

57     private SimpleDateFormat dateFormat =
58         new SimpleDateFormat("yyyy-MM-dd");
59     private JButton dismiss, reset, submit;
60     private JTextField startDate, endDate;
61
62     public void actionPerformed(ActionEvent e) {
63         if (e.getSource() == dismiss) {
64             setVisible(false);
65         } else if (e.getSource() == reset) {
66             reset();
67         } else if (e.getSource() == submit) {
68             //submit query
69             Hashtable submission = new Hashtable();
70             if (startDate.getText().length() > 0) {
71                 String date = startDate.getText()
72                 if (dateFormat.parse(date) == null) {
73                     //pop-up window "ERROR"
74                     return;
75                 } else {
76                     submission.put("StartDate", date);
77                 }
78             }
79             //.....
80         }
81     }

```

The error checking for endDate is exactly the same format.

Like in the QueryPanel, the query parameters are placed in a Hashtable that is sent to the DataManager and the results returned in the form of a 2-element Vector array. Thus the code for sending the query and updating the table is similar to that in the QueryPanel:

```
52     public void actionPerformed(ActionEvent e) {
53         if (e.getSource() == dismiss) {
54             setVisible(false);
55         } else if (e.getSource() == reset) {
56             reset();
57         } else if (e.getSource() == submit) {
58             //Check dates
59             Hashtable submission = new Hashtable();
60             //add parameters to Hashtable
61
62             Vector result[];
63             //retrieve results
64             try {
65                 synchronized(dm) {
66                     result = dm.query(submission);
67                 }
68                 if (result == null) {
69                     return;
70                 }
71
72                 DataTableModel tm =
73                     new DataTableModel(result[1], result[0]);
74                 table.setModel(tm);
75                 table.repaint();
76             } catch (Exception ie) {
77                 //...
78             }
79         }
80     }
```

EnterDataFrame

The EnterDataFrame class creates both new and update entry frames, and thus has two constructors. The first constructor takes no arguments and creates a new entry frame. The second constructor takes a Hashtable object representing the data of the entry being updated and creates an update entry frame. If a null object is given as the argument to the second constructor, then a new entry frame is created. With this in mind, the first constructor can be implemented simply by calling the second one with a null argument:

```

4 public class EnterDataFrame extends JFrame implements ActionListener{
5     //Data Members
6
7     public EnterDataFrame(Hashtable data) {
8         //if (data == null) {
9             // New Entry Frame
10        //} else {
11            // Update Entry Frame
12        //}
13    }
14
15    public EnterDataFrame() {
16        this(null);
17    }
18
19    public void actionPerformed(ActionEvent e) {
20        //Action
21    }
22
23 }

```

Like the SearchFrame, the EnterDataFrame class uses a GridBagLayout, and thus has a lengthy constructor. There are a few subtleties, however, that should be highlighted. For one, the EnterDataFrame does not use JTextField or JTextArea for any of its text fields but rather TextField and TextArea, the corresponding Java AWT components. These older text field classes are used instead of swing to allow cut-and-paste operations between the system and the Java applet. This functionality is not allowed in swing text fields existing in an applet. Also there are few parts of the constructor that distinguish between the new and update entry frames. The first is when the title of the frame is set, and the next is when the TextField meant to hold the problem ID number is instantiated. For a new entry frame, the problem ID number is not yet known and thus is not needed. Also, the date field is set to the current time for each frame.

The EnterDataFrame defines two reset() methods. One method takes no arguments and the other takes a Hashtable object. The first reset() method sets all of the data fields of the frame to their initial values. The second method, which should only be called from an update entry frame, resets the data of the frame to that given by the argument (i.e. changes the entry being updated). In each method the date is reset to the current time. With these methods, EnterDataFrames may be reused.

The EnterDataFrame class implements the ActionListener interface to listen to ActionEvents generated by the JButtons: “dismiss,” “reset,” and “apply.” The “dismiss” and “reset” cases are exactly the same as those in the SearchFrame. The “apply” case takes the entered data and sends it to the DataManager to add to the database. However, before the data can be sent, EnterDataFrame must check the data entered for correctness. In order to make a correct submission, the date must be in the correct format, a title and a description must be entered, and a valid category, system, and commentor must be selected. If the data is correct, the field values are placed in a Hashtable and sent to the DataManager. For a new entry frame, DataManager’s newEntry() method is called, and for an update frame, DataManager’s updateEntry() method is called:

```

19     private JButton dismiss, reset, apply;
20
21     public void actionPerformed(ActionEvent e) {
22         if (e.getSource() == dismiss) {
23             setVisible(false);
24         } else if (e.getSource() == reset) {
25             reset();
26         } else if (e.getSource() == apply) {
27             //check for correctness
28             Hashtable submission;
29             //place fields in hashtable
30             try {
31                 synchronized(dm) {
32                     if (data == null) {
33                         dm.newEntry(submission);
34                     } else {
35                         dm.updateEntry(submission);
36                     }
37                 }
38                 JOptionPane.showMessageDialog("Database Updated!");
39                 setVisible(false);
40             } catch (Exception ie) {
41                 JOptionPane.showMessageDialog("Update Failed!");
42             }
43         }
44     }
45 }

```

After the database has been updated, an email is sent to the system coordinator and the person assigned to handle the defect. This is done using EMailer's static method, createEmail() (see Discussion on EMailer). However, to send an email to the system coordinator, the coordinators name and email address need to be obtained. This information is available in the database, and thus may be accessed through the DataManager's getCoordinator() and getEmail() methods. Once this information is obtained, a Hashtable is constructed to hold the email fields such as "to," "from," "subject," etc. Once all the necessary information is added to the Hashtable, the information is given as an argument to EMailer's createEmail() method. Also if someone is assigned to handle the problem, an email is sent to this person as well:

```

50 public void actionPerformed(ActionEvent e) {
51     if (e.getSource() == apply) {
52         try {
53             //send query
54         } catch (Exception ie) {
55             //error, return
56             return;
57         }
58         try {
59             String coordinator = dm.getCoordinator(
60                 (String)systems.getSelectedItem());
61             String coordEmail = dm.getEmail(coordinator);
62             Hashtable email = new Hashtable();
63             email.put("FromName", "Defect Tracking");
64             email.put("FromAddress", "DefectTracking@fnal.gov");
65             email.put("SendToName", coordinator);
66             email.put("SendToAddress", coordEmail);
67             //create body
68             email.put("Body", body);
69             EMailer.createEmail(email);
70             if (assigned.getSelectedIndex() != 0) {
71                 String assignedEmail = dm.getEmail(
72                     (String)assigned.getSelectedItem());
73                 email.put("SendToName",
74                     (String)assigned.getSelectedItem());
75                 email.put("SendToEmail", assignedEmail);
76                 EMailer.createEmail(email);
77             }
78         } catch (Exception ie) {
79             //error, could not send emails
80         }
81     }
82 }

```

HistoryFrame

The HistoryFrame is divided into three regions. The top region contains the action buttons: “dismiss,” “description,” “update,” and “print.” The center region contains a JTable displaying the history data. And the bottom region is a text area containing the description of the currently selected entry in the JTable.

When the selected row of a JTable changes, a ListSelectionEvent is generated and sent to all ListSelectionListeners. Thus, in order to change the description field when the selection is changed, the HistoryFrame class implements the interface ListSelectionListener. With this interface HistoryFrame must define the method valueChanged(), to handle ListSelectionEvents:

```

4 public class HistoryFrame extends JFrame implements
5   ListSelectionListener{
6   //Data Members
7   private JTable history;
8   private JButton dismiss, update, describe, print;
9   private TextArea description;
10  private DataManager dm = DataManager.getInstance();
11
12  public HistoryFrame (Vector[] data) {
13    //create table
14    history = new JTable();
15    //add ListSelectionListener
16    ListSelectionModel lsm = history.getSelectionModel();
17    lsm.addListSelectionListener(this);
18    description = new TextArea();
19    description.setEditable(false);
20    //add table and text area
21    add(history);
22    add(description);
23  }
24
25  public void valueChanged(ListSelectionEvent e) {
26    int selectedRow = history.getSelectedRow();
27    if (selectedRow < 0) {
28      return;
29    }
30    TableModel tm = history.getModel();
31    int col = history.getColumn("Description").getModelIndex();
32    String str = (String) tm.getValueAt(selectedRow, col);
33    description.setText(str);
34    description.setCaretPosition(0);
35  }
36 }

```

In the valueChanged() method, the frame retrieves the description from the TableModel and places it in the TextArea (note that this frame uses AWT TextArea also).

The constructor for HistoryFrame takes a 2-element Vector array. Like the Vector arrays mentioned before, the first element is expected to hold the column names of the history data and the second element is expected to hold the row data in a 2-D Vector. With this array, the constructor creates a JTable and adds it to the frame's content pane.

Unlike the other frames mentioned, the HistoryFrame uses a BorderLayout and therefore, does not have a lengthy constructor.

Like the other frames, the HistoryFrame also defines a reset() method. This method takes a Vector array of the same structure that is expected in the constructor. This array is used to change the TableModel of the currently displayed table:

```

25 public void reset(Vector[] data) {
26   history.setModel(new DataTableModel(data[1], data[0]));
27   description.setText("");
28 }

```

The HistoryFrame also implements that ActionListener interface to listen to ActionEvents generated by the JButtons. The actionPerformed() method handles 4 cases. The “dismiss” case is the same as it is in the other frames. The “update” case is the same as the “update” case in the ActionPanel and brings up an update entry frame. In the

“description” case and the “print” case, the data from the JTable is used to construct a long string. In the “description” case, this string is used to either create or reset a DescribeFrame, and in the “print” case, this string is sent to a StringPrinter to be printed:

```
42     private JButton dismiss, update, describe, print;
43     private DescribeFrame describeframe;
44
45     public void actionPerformed(ActionEvent e) {
46         if (e.getSource() == describe || e.getSource() == print) {
47             TableModel tm = history.getModel();
48             int colCount = tm.getColumnCount();
49             int rowCount = tm.getRowCount();
50             StringBuffer doc = new StringBuffer();
51             for (int i = 0; i < rowCount; i++) {
52                 for (int j = 0; j < colCount; j++) {
53                     doc.append(tm.getColumnName(j) + ":\t");
54                     doc.append(tm.getValueAt(i, j) + "\n");
55                 }
56             }
57             if (e.getSource() == describe) {
58                 int colIndex =
59                     history.getColumnModel().getColumnIndex("ID");
60                 String id = (String) history.getValueAt(0, colIndex);
61                 if (describeframe == null) {
62                     describeframe =
63                         new DescribeFrame(doc.toString(), id);
64                 } else {
65                     describeframe.reset(doc.toString(), id);
66                 }
67                 describeframe.setVisible(true);
68                 return;
69             } else {
70                 StringPrinter sp = new StringPrinter(doc.toString());
71                 sp.print();
72             }
73         }
74     }
```

DescribeFrame

The DescribeFrame simply displays history data in text form. The constructor for DescribeFrame takes two strings: the first being the data to be displayed in the TextArea and the second being the problem id of the history data. The data text is used to create a TextArea that takes up most of the frame’s visible area, and the problem id is used to identify the data. A FlowLayout is used to arrange the frame components.

The DescribeFrame implements the ActionListener interface to listen to the JButtons “dismiss” and “email.” When the “email” button is pressed, the actionPerformed() method uses a JOptionPane to prompt the user to enter his/her email address. Once the email address is obtained, an email is constructed and sent to the given address using the Mailer class:

```

85     public void actionPerformed(ActionEvent e) {
86         if (e.getSource() == dismiss) {
87             setVisible(false);
88         } else if (e.getSource() == email) {
89             try {
90                 //get address from user
91                 String address = JOptionPane.showInputDialog(
92                     "Please enter your e-mail address");
93                 Hashtable submission = new Hashtable();
94                 submission.put("FromAddress",
95                     "DefectTracking@fnal.gov");
96                 submission.put("FromName", "Defect Tracking");
97                 submission.put("SendToName", address);
98                 submission.put("SendToAddress", address);
99                 /* etc */
.00                 Emitter.createEmail(submission);
.01             } catch (Exception ie) {
.02                 //error occurred
.03             }
.04         }
.05     }

```

Emitter

The Emitter class is a helper class used to send emails. The default constructor for Emitter is defined private so that the class may not be instantiated. The class only defines two methods, each of which is static. The first, createEmail(), takes a Hashtable and creates an email based on the data in the Hashtable. The second, send(), is a helper function for createEmail() and is declared private. The send() method simply takes a String and sends it to an OutputStream in an appropriate manner.

To send an email, the java.net package is used and a Socket object connecting to the server, smtp.fnal.gov, is created. With this Socket object, an OutputStream may be obtained to send data to the SMTP server:

```

14     public static void createEmail(Hashtable data)
15         throws Exception {
16
17         Socket hostSocket = new Socket("smtp.fnal.gov", 25);
18         OutputStream out = hostSocket.getOutputStream();
19         //send(out, email);
20         hostSocket.close();
21     }

```

Using a Socket object to connect to a server is just fine when Defect is running as an application, but when Defect is running as an applet, a SecurityException is thrown. This is because all applets run under a SecurityManager that imposes certain restrictions on the capabilities of an applet. One of these restrictions is that an applet cannot communicate with any servers other than the server the applet resides. The Defect applet does not reside on the smtp.fnal.gov server, and therefore cannot communicate with it directly.

There are two possible solutions around this obstacle. One is to install a mailer daemon (like sendmail) on the server with the Defect applet (this solution may be explored later). The other solution is to use Common Gateway Interface (CGI). CGI provides an interface that allows hosts to execute applications on a server. Since these applications

run on the server, they are not as restricted as applets. Thus Emailer can execute a CGI program residing on the same server as the Defect applet to send the requested email. This is the solution that is currently used in the DescribeFrame.

The program used to send the email is a script written in Perl (see Appendix for script code). To communicate with the script, Emailer creates a URLConnection object with the URL address of the script. Then a string is assembled containing the fields “to,” “from,” “subject,” and “body.” Each field is associated with a value in the string using the characters “:” (i.e. “to::johndoe@fnal.gov”) and delimited using “&&” (i.e. “to:<you>&&from:<me>”). Afterwards, an OutputStream is obtained from the connection, and the string is printed on the stream. The script will not run until an InputStream is also obtained, but this must be done after the string is sent to the OutputStream. When an InputStream is requested from the connection, an Exception is very likely to be thrown (this is most likely because of wrong configurations on the server), but this does not mean the script did not run properly. So this Exception is trapped in a try/catch block surrounding the request of an InputStream:

```

14     public static void createEmail(Hashtable data)
15         throws Exception {
16         URL url = new URL(Defect.getBase(), "cgi-bin/email.cgi");
17         URLConnection conn = url.openConnection();
18         PrintWriter out = new PrintWriter(conn.getOutputStream());
19         //send info to script
20         String mailing = "to::" + data.get("SendToAddress") + "&&"
21             + "from::" + data.get("FromAddress") + "&&"
22             + "subject::" + data.get("Subject") + "&&" + "body::"
23             + data.get("Body");
24         out.println(mailing);
25         out.flush();
26         out.close();
27         try{
28             InputStream in = conn.getInputStream();
29         } catch(IOException ie) {
30             /* Exception generated should be httperror code 500.
31              * Otherwise, the script did not run properly */
32             if (ie.toString().indexOf("500") < 0) {
33                 throw ie;
34             }
35         }
36     }

```

The problem with CGI is that it does not allow efficient error handling in the applet. So CGI will be used only when necessary (when Defect runs as an applet), and use the send() method whenever possible (when Defect runs as an application). If Defect runs as an application, the AppletFrame class will be instantiated (see Appendix for AppletFrame class). Thus Emailer chooses whether or not to use CGI depending on whether or not an AppletFrame object exists:

```

14     public static void createEmail(Hashtable data)
15         throws Exception {
16         if (AppletFrame.isInstantiated()) {
17             //use send()
18         } else {
19             //use CGI
20         }
21     }

```

DbAccessor

The DbAccessor establishes the connections to the Defect Tracking database. A DbAccessor is used by the DataManager to communicate with the database. In order to setup communication with the database, the DbAccessor must first determine the data-source of the database and the appropriate driver to access it. These may be determined by Defect's static method, `getProperties()`. The driver is loaded by calling the static method, `Class.forName()`, which loads a class. If Defect is run as an application or the driver and `dataSource` parameters are not defined for the applet, `getProperties()` will return an empty Properties object. In that case DbAccessor uses the Postgres driver as default:

```
4 public class DbAccessor {
5     //Data Members
6     public final static String POSTGRES_DRIVER;
7     public final static String SYBASE_DRIVER;
8     public final static String POSTGRES_DATA_SOURCE;
9     public final static String SYBASE_DATA_SOURCE;
10    private String driver, dataSource;
11
12    public DbAccessor() {
13        Properties pro = Defect.getProperties();
14        driver = pro.getProperty("driver");
15        dataSource = pro.getProperty("dataSource");
16        if (driver == null || driver.equals("")) {
17            driver = POSTGRES_DRIVER;
18            dataSource = POSTGRES_DATA_SOURCE;
19        }
20        try {
21            Class.forName(driver);
22        } catch (Exception e) {
23            e.printStackTrace();
24            System.out.println(
25                "Could not initialize driver!");
26        }
27    }
28 }
```

DbAccessor defines 5 methods: `connect()`, `disconnect()`, `query()`, `update()`, and `getDriverName()`. The `getDriverName()` method simply returns the name of the driver being used. The `connect()` and `disconnect()` methods opens and closes connections to the database. The `connect()` method is expected to be called before `query()` or `update()` is used, and `disconnect()` is expected to be called afterwards. To make a Connection, the DriverManager class is invoked to call on the Driver loaded in the constructor. Once a Connection is made, a Statement is created from the connection to allow the DbAccessor to execute SQL statements. To disconnect, the Statement and Connection objects are closed:

```

29     private final static String uid;
30     private final static String pwd;
31     private Connection con;
32     private Statement stmt;
33
34     public void connect() throws Exception {
35         con = DriverManager.getConnection(dataSource, uid, pwd);
36         stmt = con.createStatement();
37     }
38
39     public void disconnect() throws Exception {
40         if (con != null) {
41             stmt.close();
42             con.close();
43             con = null;
44             stmt = null;
45         }
46     }

```

Note that these methods may throw Exceptions.

The query() method takes a SQL query statement and executes it using the Driver Statement created in the connect() method. The result of the query comes in the form of a ResultSet object (defined in the java.sql package), which is returned by query():

```

48     public ResultSet query(String sql) throws Exception {
49         if (con != null) {
50             ResultSet rst = stmt.executeQuery(sql);
51             return rst;
52         } else {
53             throw new Exception("No Connection to DB!");
54         }
55     }

```

The update() method is similar to the query() method. The update() method takes a SQL update statement and executes the update using the Driver statement. No value is returned:

```

57     public void update(String sql) throws Exception {
58         if (con != null) {
59             stmt.executeUpdate(sql);
60         } else {
61             throw new Exception("No Connection to DB!");
62         }
63     }

```

Like connect() and disconnect(), update() and query() may generate Exceptions that must be caught at some point.

DataManager

The DataManager constructs the SQL statements depending on which methods are being called, and sends them to the DbAccessor. If the DbAccessor returns a ResultSet, the results are processed and returned in an appropriate format.

DataManager is a singleton class and, therefore, has only one instance. The constructor is defined as private and the class provides a static method that returns the instance of the class:

```

4 public class DataManager {
5     private MessagePanel mpanel = new MessagePanel();
6     private static DataManager _instance = new DataManager();
7     private DbAccessor dba = new DbAccessor();
8
9     private DataManager() {
10    }
11
12    public static DataManager getInstance() {
13        return _instance;
14    }
15    public static void setMsgPanel(MessagePanel mpanel) {
16        _instance.mpanel = mpanel;
17    }
18 }

```

The Defect database consists of several tables and fields. Figure 6 shows a schematic of the tables, their fields, and how the tables are related. Apparently the tables are not in the simplest form, and thus long and complicated SQL statements must be constructed to perform queries and update data. Also these SQL statements may be driver dependent, making it necessary for the DataManager to call on DbAccessor's getDriverName() method. The actual syntax and construction of these SQL statements are not very interesting and will not be discussed in length.

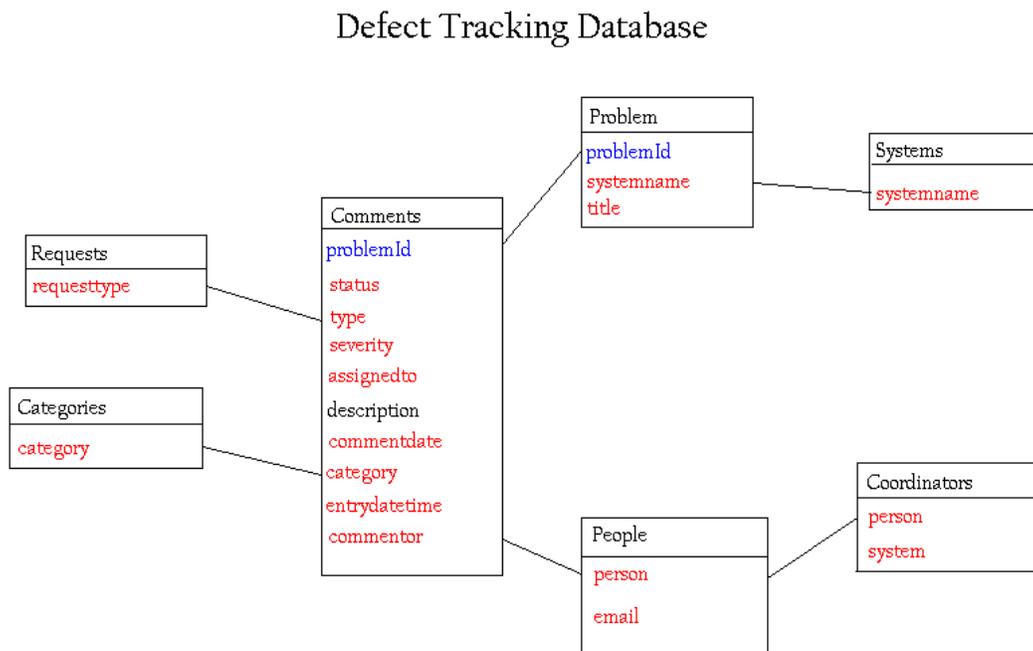


Figure 8: Defect Tracking Database.

DataManager defines 4 methods that returns field names in the database: getCategories(), getSystems(), getPersons(), and getRequests(). These methods return a Vector containing

the desired fields. Once these methods retrieve the fields, the fields are stored so that they may be returned in the future without using the DbAccessor for a second time:

```
21     public Vector getSystems() throws Exception{
22         if (systems != null) {
23             return systems;
24         }
25         String sql = "SELECT * FROM systems order by systemName".
26         mpanel.setColor(Color.black);
27         mpanel.setMessage("Getting Systems from DB");
28         try {
29             dba.connect();
30             rs = dba.query(sql);
31             systems = new Vector();
32             while(rs.next()) {
33                 String str = rs.getString(1);
34                 systems.addElement(str);
35             }
36             mpanel.setMessage("Done");
37             dba.disconnect();
38             return systems;
39         } catch (Exception e) {
40             systems = null;
41             mpanel.setColor(Color.red);
42             mpanel.setMessage("Failed to get Systems from DB.");
43             try{dba.disconnect();} catch(Exception ie) {}
44             throw e;
45         }
46     }
```

Here a connection is made to the DbAccessor and its query() method is called. This method returns a ResultSet, which is processed to produce a Vector in while loop. Once the Vector is made, the connection is closed and the Vector is returned.

Note that the getSystems() method catches Exceptions from the DbAccessor only to throw them back again. Exceptions are caught here for two reasons. The first is to make sure the DbAccessor's disconnect() method is called at some point, and the second is to display appropriate error messages with the MessagePanel. Since the DataManager is not fully equipped to handle the Exceptions completely, the caught Exception is thrown again so that it can be handled by the upper level components. This Exception handling will be seen throughout the DataManager.

The query() method takes a Hashtable holding the query parameters and returns a 2-element Vector array in the format discussed previously. The query() method is defined to handle any query parameter, except a problem id parameter, that may be given by a Defect component and constructs a SQL statement by concatenating the parameters:

```

48     public Vector []query(Hashtable query) throws Exception {
49         String sql = "SELECT p.problemId, c.commentDate, ..."
50         if (query.get("System") != null) {
51             sql = sql + "and systemName = \' "
52                 + query.get("System") + "\' ";
53         }
54         if (query.get("Category") != null) {
55             sql = sql + "and l.category = \' "
56                 + query.get("Category") + "\' ";
57         }
58         if (query.get("Status") != null) {
59             sql = sql + "and l.status= \' "
60                 + query.get("Status") + "\' ";
61         }
62         /* etc */
63         sql = sql + "and c.commentDate = ..."
64         return runQuery(sql);
65     }
66 }

```

Since the query() method attempts to handle any possible combination of parameters that may be given by a Defect component, it is possible to construct a query that will never return any results. Therefore, it is the responsibility of the caller to give a combination of parameters that makes sense.

The idQuery() method takes a string representing a problem ID number and makes a problem ID query. This method is defined outside of the query() method because this search only requires one parameter and the SQL statement is of a different structure:

```

67     public Vector[] idQuery(String id) throws Exception {
68         String sql = "SELECT p.problemId, ..."
69             + "where problemId = " + id
70             + " and c.commentDate = ...";
71         return runQuery(sql);
72     }

```

The getHistory() method is similar to the idQuery() method. It also takes a string representing a problem ID number and returns a Vector array of data. The only difference is the structure of the SQL statement generated.

The query(), getHistory(), and idQuery() methods call the protected method runQuery(). This method takes an SQL statement, sends it to the DbAccessor using its own query() method, and takes the returned ResultSet and processes it to return a 2-element Vector array representing the table data:

```

74     protected Vector [] runQuery(String sql) throws Exception {
75         Vector twovector [] = new Vector[2];
76         mpanel.setColor(Color.black);
77         mpanel.setMessage("Querying DB");
78         try {
79             dba.connect();
80             ResultSet rs = dba.query(sql);
81             ResultSetMetaData rsmd = rs.getMetaData();
82             int count = rsmd.getColumnCount();
83             Vector columnNames = new Vector(count);
84             for (int i = 1; i <= count; i++) {
85                 columnNames.addElement(rsmd.getColumnName(i));
86             }
87             twovector[0] = columnNames;
88             Vector data = new Vector();
89             Vector rowData;
90             while(rs.next()) {
91                 rowData = new Vector();
92                 for (int i = 1; i <= columnNames.size(); i++) {
93                     rowData.addElement(rs.getString(i));
94                 }
95                 data.addElement(rowData);
96             }
97             twovector[1] = data;
98             dba.disconnect();
99             mpanel.setMessage("Query Completed");
100            return twovector;
101        } catch (Exception e) {
102            mpanel.setColor(Color.red);
103            try{dba.disconnect();} catch(Exception ie) {}
104            mpanel.setMessage("Query failed! Try Again Later.");
105            throw e;
106        }
107    }

```

Lines 81-87 retrieve the column names and place them in the first element of the Vector array. Lines 88-97 retrieve the row data and place it in a 2-D Vector. This Vector is placed in the second element of the array, which is returned in line 100.

The last query method defined in DataManager, `getEntryData()` takes a string representing a problem ID number and returns a Hashtable representing entry data. Since its return type is a Hashtable, `getEntryData()` cannot use `runQuery()` and must do most of the work on its own. The major difference between this method and `runQuery()`'s handling of the `ResultSet` returned by `DbAccessor` is that `getEntryData()` expects only one row to exist in the `ResultSet` and this row data is placed in a Hashtable rather than a vector:

```

109 public Hashtable getAllData(String id) throws Exception {
110     String sql = "SELECT c.problemId, p.systemName, ...";
111     Hashtable data = new Hashtable();
112     try {
113         mpanel.setColor(Color.black);
114         mpanel.setMessage("Getting Data for DB");
115         dba.connect();
116         ResultSet rs = dba.query(sql);
117         ResultSetMetaData rsmd = rs.getMetaData();
118         int count = rsmd.getColumnCount();
119         rs.next();
120         for (int i = 1; i <= count; i++) {
121             data.put(rsmd.getColumnName(i), rs.getString(i));
122         }
123         dba.disconnect();
124         return data;
125     } catch (Exception e){
126         mpanel.setColor(Color.red);
127         mpanel.setMessage(
128             "Exception encountered while getting data");
129         try{dba.disconnect();} catch(Exception ie) {}
130         throw e;
131     }
132 }

```

The newEntry() and updateEntry() methods take a Hashtable representing entry data. These methods send updates to the DbAccessor, and thus calls on its update() method. The only difference between these two methods is the construction of the SQL statement that is given to update(). The newEntry() method needs to construct 2 SQL statements, and updateEntry() needs to construct 3 statements. More than one statement is needed since only one table in the database may be updated at a time. Here is some sample code that resembles newEntry():

```

12 public int newEntry(Hashtable data) throws Exception {
13     String idsql = "Select MAX(problemId) from ...";
14     String sql1 = "insert into comments(problemId...";
15     String sql2 = "insert into ....";
16     int id;
17     try {
18         mpanel.setColor(Color.black);
19         mpanel.setMessage("Updating DB");
20         //find max problem id number
21         dba.connect();
22         ResultSet rs = dba.query(idsql);
23         rs.next();
24         id = rs.getInt(1) + 1;
25         /*finish sql construction with problem id */
26         //send new entry
27         dba.update(sql2);
28         dba.update(sql1);
29         mpanel.setMessage("DB updated");
30         dba.disconnect();
31         // return problem id
32         return id;
33     } catch (Exception e) {
34         System.out.println(sql2);
35         mpanel.setColor(Color.red);
36         mpanel.setMessage("Exception encountered");
37         try{dba.disconnect();} catch(Exception ie) {}
38         throw e;
39     }
40 }

```

In the case of newEntry(), the database must be queried in order to find the largest problem ID number that currently exists (Lines 143-145). Then the new entry is added to the database with max problem ID plus 1 as its problem ID number. This number is returned once the database is updated. The updateEntry() method does not have to worry about this, and it simply creates the SQL and calls update().

Conclusion

The appearance of the final product, DPPTS Applet version 2.0, can be seen in figures 7-12. Note that the appearance of the final product is not that different from that of its predecessor, version 1.5.2. This should be no surprise since the two versions have, for the most part, the same specifications.

The screenshot shows the 'Defect' applet window titled 'Defect, Problem, & Progress Tracking V2.0'. It features a navigation bar with 'New', 'Update', 'History', and 'Print' buttons. On the left, there are search filters for 'All Systems', 'All Categories', 'All Request Types', 'All Persons', 'All Stats', and 'All Severities', along with a 'Last Month' checkbox and a 'problem ID:' input field. The main area is a table with the following data:

ID	Date	System	Status	Category	Title
237	1980-01-0...	PC Support	closed	Configurati...	Defective date due to ...
136	1997-07-0...	DMCS	closed	In-house S...	Hardware test
133	1998-01-0...	DMCS	closed	In-house S...	slow scan
138	1998-02-0...	DMCS	deferred	In-house S...	Video camera
168	1998-03-0...	DMCS	closed	In-house S...	Database model
167	1998-03-0...	DMCS	closed	In-house S...	Requirements specifi...
171	1998-05-0...	DMCS	closed	In-house S...	Prototype Positioning ...
135	1998-07-1...	DMCS	closed	In-house S...	Main menu
170	1998-08-0...	DMCS	deferred	In-house S...	Refined data model.
169	1998-08-1...	DMCS	closed	In-house S...	Prototype of GUI.
131	1998-09-2...	DMCS	closed	In-house S...	Remote File System (...)
134	1998-09-2...	DMCS	closed	In-house S...	CCS
132	1998-10-0...	DMCS	closed	In-house S...	Scribe
225	1998-10-2...	PC Support	closed	Network	131.225.47.128 subn...
144	1998-11-2...	DMCS	closed	In-house S...	Documentation of cal...
137	1998-12-2...	DMCS	closed	In-house S...	Calibrations entry ap...
148	1999-01-1...	EMS	closed	In-house S...	Project organization: ...

At the bottom of the window, it says 'Getting Data for DB'.

Figure 9: Defect Applet Version 2.0

Update Defect

Dismiss Reset Apply

Event Date: 2002-08-01 09:50 ID: 237 Categories: Configuration

Title: Defective date due to Y2K problem on Jerzy's laptop

System: PC Support Status: closed Commentor: Unspecified Commentor

Type: defect Severity: low Assigned To: Ping Wang

Descripton:

Figure 10: Update Entry Frame

New Defect

Dismiss Reset Apply

Event Date: 2002-08-01 09:50 Categories: Unknown

Title:

System: Unknown System Status: open Commentor: Unspecified Commentor

Type: defect Severity: low Assigned To: Not Assigned

Descripton:

Figure 11: New Entry Frame

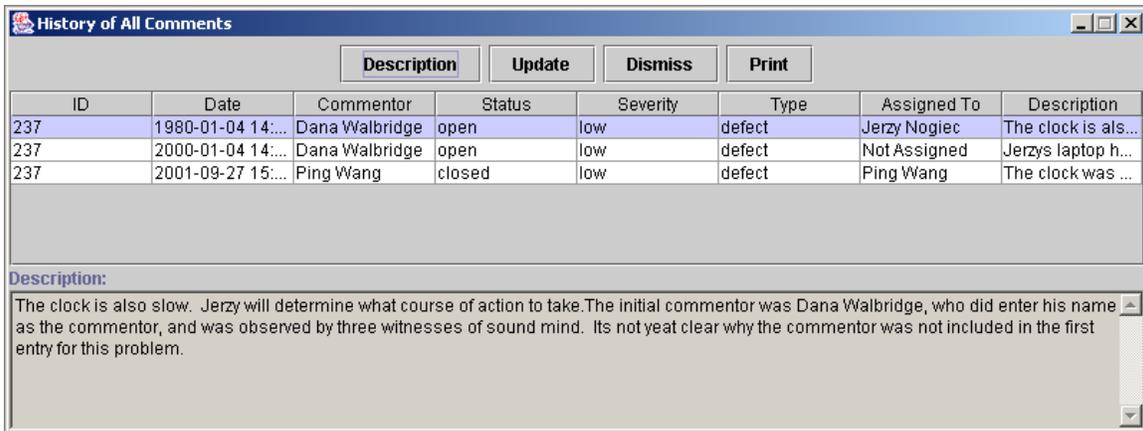


Figure 12: History Frame

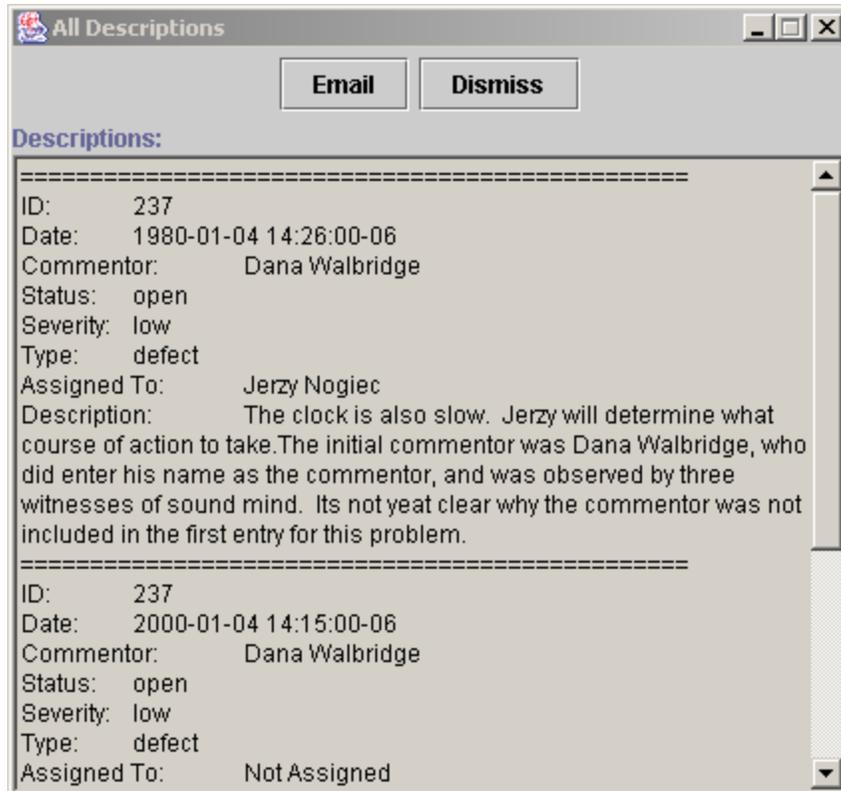


Figure 13: Description Frame

Figure 14: Advanced Search Frame

Version 2.0 may use one of two drivers to access the database: the Sybase driver or the Postgres driver. The classes that provide the functionality of these drivers are not standard classes in Java, however. Thus in order to run version 2.0, these special classes must be included somewhere. Currently the Postgres classes reside in a jar (Java Archive) file called “postgres,” and the Sysbase classes reside in “jconnect2.” In order to run version 2.0 as an application, one of these files must be included in the classpath depending on which driver is being used. As an applet the jar file needs to be added to the applet classes by setting the “ARCHIVE” parameter of the “<APPLET>” tag equal to the desired jar file.

Version 2.0 was implemented using Java Software Development Kit version 1.3.1, and thus requires Java Runtime Environment version 1.3 or better to run properly. So in order to run Defect as an application, a fairly recent JRE needs to be installed on the console Defect is meant to run on, and in order to run Defect as an applet, the user’s browser must possess a fairly recent virtual machine for Defect to run properly. But sadly, any version of Internet Explorer and Netscape (the most common browsers) below version 6.0 uses fairly old virtual machines that are not compatible with many of the updates in JSDK version 1.3. Since these browser versions are the most common, the HTML file containing the “<APPLET>” tag for Defect is converted with the Java program HTMLConverter, which forces the browser loading the converted HTML to use the Java plug-in to run the applet. This guarantees that the applet will be run in a fairly recent runtime environment.

DPPTS Applet version 2.0 currently resides at the address <http://mtfpc22.fnal.gov/Defect>.

References

Cornell, Gary and Cay S. Horstmann. *Core Java 1.2, Volume 1-Fundamentals*.
California: Sun Microsystems Press, 1999.

“Sending E-mail from Perl for NT.” <<https://secure.unisite.net/html/sendmailnt.html>>.

Acknowledgements

I would like to thank the SIST committee members for allowing me this work opportunity at Fermilab. Furthermore, I would also like to thank my supervisor, Jerzy Nogiec, for his guidance and support throughout my internship and give my gratitude to the SDSG team, Kelly Trombly-Freytag, Dana Walbridge, Ping Wang, Gene Desavouret, Pennie Hall, and Frank Burzynski for all their help and support in finishing my project. And finally, I want to give a special thanks to all the SIST interns who have not only made this summer an enjoyable one, but have also enriched my life and left me with an experience I'll never forget.

Appendix A: AppletFrame class

```
/**
 * AppletFrame
 * Source: Core Java Volume 1 - Fundamentals
 *         Example 10-12, p. 580-581
 * @version 1.21 31 July 2002
 * @author Cay Horstmann
 */

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import java.lang.*;
import java.io.*;

public class AppletFrame extends JFrame implements AppletStub, AppletContext{
    private static boolean instantiated = false;

    public AppletFrame(Applet a, int x, int y){
        instantiated = true;
        setTitle(a.getClass().getName());
        setSize(x, y);
        addWindowListener(new WindowAdapter()
        { public void windowClosing(WindowEvent e)
          { System.exit(0);
            }
        }
        );
        Container contentPane = getContentPane();
        contentPane.add(a);
        a.setStub(this);
        a.init();
        show();
        a.start();
    }

    /**
     * Added by John Biddle
     * @returns True if instance of AppletFrame exists.
     */
    public static boolean isInstantiated() {
        return instantiated;
    }
}
```

```
    }  
    // AppletStub methods  
    public boolean isActive() { return true; }  
    public URL getDocumentBase() { return null; }  
    public URL getCodeBase() { return null; }  
    public String getParameter(String name) { return ""; }  
    public AppletContext getAppletContext() { return this; }  
    public void appletResize(int width, int height) {}  
  
    // AppletContext methods  
    public AudioClip getAudioClip(URL url) { return null; }  
    public Image getImage(URL url) { return null; }  
    public Applet getApplet(String name) { return null; }  
    public Enumeration getApplets() { return null; }  
    public void showDocument(URL url) {}  
    public void showDocument(URL url, String target) {}  
    public void showStatus(String status) {}  
    public InputStream getStream(String key) { return null;}  
    public Iterator getStreamKeys() {return null;}  
    public void setStream(String key, InputStream stream) {}  
  
}
```

Appendix B: email.cgi

```
#!/usr/bin/perl
# sends email over smtp
# Requires the fields: "to," from," "subject," and "body."
# Fields must be associated with corresponding values with
# the characters ":", and the field/value associations must
# be delimited by "&&"
# source: "Sending E-mail from Perl for NT",
#   https://secure.unisite.net/html/sendmailnt.html
use Net::SMTP;
$line = <STDIN>;
$temp = $line;
while ($temp) {
    $temp = <STDIN>;
    $line = $line.$temp;
}
@params = split '&&', $line, 4;
for ($i = 0; $i < 4; $i += 1) {
    @list = split '::', @params[$i], 2;
    if (@list[0] eq "to") {
        $to = @list[1];
    } elsif (@list[0] eq "from") {
        $from = @list[1];
    } elsif (@list[0] eq "subject") {
        $subject = @list[1];
    } elsif (@list[0] eq "body") {
        $content = @list[1]
    }
}
$smtp = Net::SMTP->new('smtp.fnal.gov'); # connect to an SMTP server
$smtp->mail($from); # use the sender's address
here
$smtp->to($to); # recipient's address
$smtp->data(); # Start the mail

# Send the header.
$smtp->datasend("From: ".$from."\n");
$smtp->datasend("To: ".$to."\n");
$smtp->datasend("Subject: ".$subject."\n");
$smtp->datasend("\n");

# Send the body.
$smtp->datasend($content);
$smtp->dataend(); # Finish sending the mail
$smtp->quit; # Close the SMTP connection

print 'success';
```