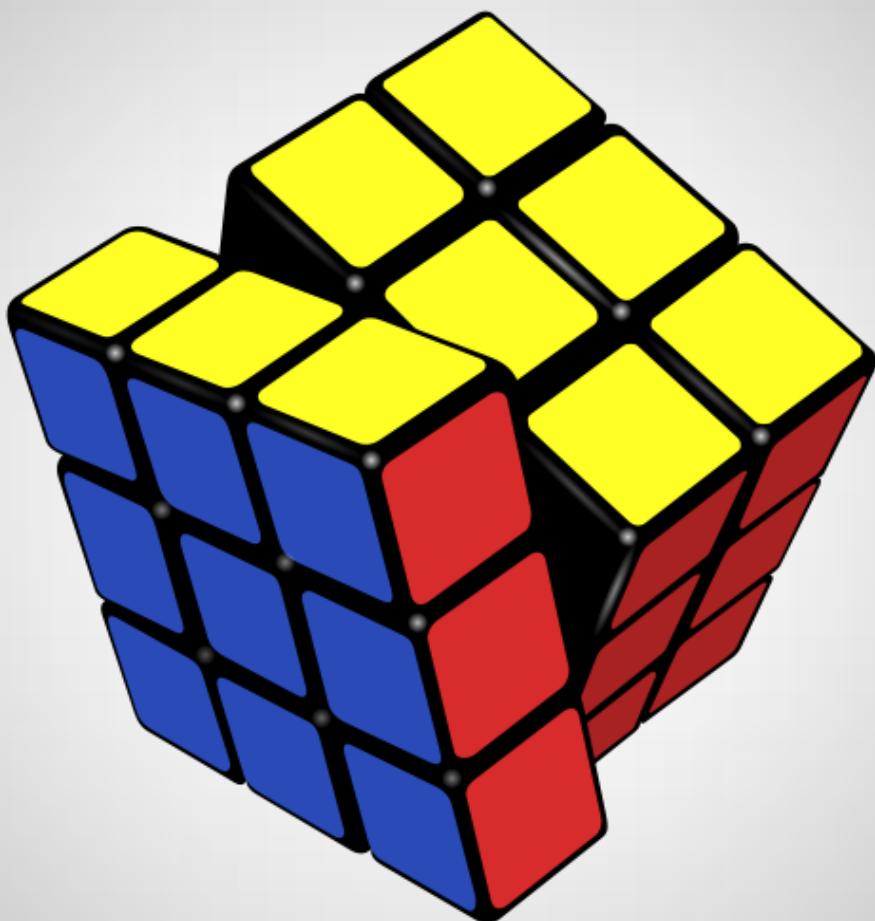


SECTION 6

# APPRAISAL



## Contents

Introduction .....	2
Comparison of System with Objectives .....	3
Useability of the System .....	10
Target Acquisition Time.....	10
Latency Time.....	11
Readability .....	12
Navigability.....	13
Possible Improvements/Extensions .....	14
Analysis of User Feedback .....	16

## Introduction

Kuubik was developed to help people learn how to solve the Rubik's cube, and to help the administrator of a school's mathematics club to record information about competitions and members in the club. There are features for both learning and practising, so the system can be used by people with all levels of experience. The system was intended to provide a comprehensive environment that would enable users to further their skill rapidly. The system can be used by anyone who wishes to learn how to solve the Rubik's cube.

The program allows the user to:

- Learn how to solve the Rubik's cube using built-in and custom-made tutorials.
- Automatically generate solutions for a given cube-state.
- Paint custom states onto a virtual Rubik's cube.
- Practise solving the Rubik's cube using randomly generated scrambles.
- Save/load cube states, solve information, statistics and scrambles.
- View a graph of your times and save this graph as an image.
- Record details about members, such as name, date of birth, gender, etc.
- Create new competitions and record the dates for these competitions.
- Record the times that the members achieve in each competition.
- Automatically sort members by best average for each competition.

## Comparison of System with Objectives

The following table shows the objective and the corresponding aspect of the system that has been achieved. The proposed objectives can be found on page 15 of the analysis section. See the appraisal questionnaires on page 7 of the appendix section to see user feedback on proposed objectives.

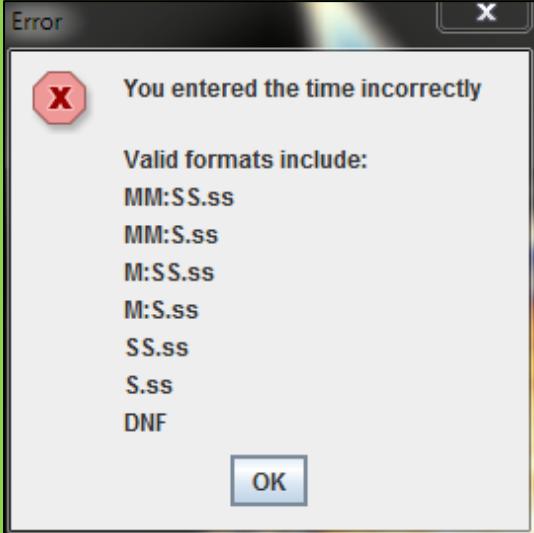
**Key**

Not Achieved	Achieved	Surpassed
Objective	System Functionality	
1. The system should be able to run on Windows 7.	<p>The system, including database features, operates on the Windows 7 operating system. I have tested the system on four different machines running Windows 7 with no problems.</p> <p>The program also runs on Windows Vista and Windows XP, which is important so that many people with different operating systems can use the system.</p>	
2. The system should allow the user to record solves with ease and should store the information for the solve automatically.	<p>There are (only) three buttons in the main window presented to the user when the program opens, one of which is the 'Start New Solve' button, meaning that recording a solve is very simple. After completing a solve, the time taken is stored in the list at the right-hand side of the screen.</p> <p>See the appraisal questionnaires on page 7 of the appendix section.</p>	
3. The system should provide an interactive learning process.	<p>The user can enter tutorial mode by selecting a tutorial from the tutorial menu.</p>	
4. The system should allow the user to load information about other solves into the program.	<p><i>File → Load Solve Information</i> allows the user to load the information stored in a text file that could have been sent over the Internet or shared with them through a message.</p>	
5. a) The system should calculate statistics automatically and display them to the user.	<p>After the user completes/deletes/edits a solve, the new statistics are calculated and displayed in the text area at the bottom of the main window. The <i>Current Average of x</i> statistics are immediately visible, and the <i>Best Average of x</i> statistics can be viewed by scrolling down in the text area.</p>	

Objective	System Functionality
5. b) The statistics should be able to be saved to a pdf file.	By using the <i>File → Save Statistics – pdf</i> menu option, the user can choose the location to which the pdf file will be saved. The information is stored in the style of a table.
6. The system should load within 10 seconds.	On average, the system loads and all features are usable in 8 seconds.
7. The system must allow the user to store pertinent data, such as solve information, algorithms etc.	Algorithms can be stored permanently in the Algorithm Table window ( <i>View → Show Algorithm Table → Add Algorithm</i> ). The details for a particular solve can be saved to a text file using the Solve Editor window ( <i>Edit → Edit Selected Solve → Save to File</i> ).
8. a) The system should present a graph of the previous 12 times in a separate window.	<i>View → Show Time Graph</i> opens a separate window showing a graph of the past 12 times. If there are fewer than twelve times, then as many as possible are shown in the window.
8. b) There should be an option to save the time graph as an image on the user's computer.	<i>Time Graph → File → Save as Image</i> allows the user to choose the location to which the .png image of the graph should be saved. This image can then be opened in a suitable image viewer.
9. The system should allow the execution of previous solves to be performed automatically.	After completing/loading a solve, the execution of the solve can be viewed by opening the solve in the Solve Editor window and then clicking the 'View Execution' button. The speed at which moves are performed is specified in the Preferences window.
10. The system should be able to generate solutions to any (valid) given state and show this solution to the user.	<p>After entering a valid cube state, either by performing moves or painting the stickers, a solution can be generated by using <i>Tools → Solve Cube</i>, and this solution is shown in the text area at the bottom of the main window along with a text explanation of what is going on in each step. This solution is then performed on the cube automatically at the speed specified in the Preferences window.</p> <p>See test index 3.05 - 3.10 on page 25 of the testing section.</p>

Objective	System Functionality
11. The system should allow the user to import/export cube states, i.e. the permutations of the pieces should be able to be saved and loaded routinely.	<i>File</i> → Save Cube-State – txt allows the user to save the current state of the cube to a text file. <i>File</i> → Load Cube-State – txt allows the user to choose a text file containing the cube state that they wish to load into the program. If the file chosen is valid, then the state stored in the file will be extracted and applied to the virtual cube in the main window.
12. The system should be able to generate a solution for a selected piece.	<i>Tools</i> → Solve Piece enters the user in a mode in which clicking a piece on the cube generates a solution for that piece. A warning window is shown before generating the solution in order to confirm their decision.
13. The system must allow the user to change the colour of stickers manually; this allows the user to paint a custom state on the cube.	<i>Tools</i> → Paint Custom State opens the Color Selection window which allows the user to select the colour that they want to paint the stickers they click. There is also an option to clear all of the stickers before painting in order to make the task easier, <i>Tools</i> → Clear Stickers which turns all stickers to grey.
14. a) The system must allow the user to save the solves in the current session to the database.	To store the solves listed at the right-hand side of the main window, the user can use <i>File</i> → Save Session to Database to save all of the solves in the list to the database.
14. b) The solves saved in the database must be displayed in a separate window must provide an option to sort the solves by fastest time to slowest time.	The user can view the solves stored in the database by using <i>View</i> → Show Solve Table. By default, the solves are sorted with fastest time first and slowest time last, but the user can specify the field by which the records should be sorted by using <i>Solve Table</i> → Sorting menu and then choosing the way in which they wish the records to be sorted.

Objective	System Functionality
15. a) The system must provide easy means of navigation.	<p>Most of the navigation through the system takes place through the menu bar in the main window, so the point of navigation is very central. Once another window is opened, such as the Member Table window, there are only input windows, i.e. all major windows can be accessed from the main window and any windows that need to be accessed from within other windows are specific to that window.</p> <p>See the appraisal questionnaires on page 7 of the appendix section.</p>
15. b) There must be clearly labelled menu items which take the user to the relevant part of the system.	<p>Menu items have a concise description and a consistent font, meaning that the user will not be confused when speculating what each menu item does.</p>
15. c) There must be understandable buttons throughout to help the user access all features of the system.	<p>Buttons have a consistent font and common clear descriptions of the functions so that the user can learn how to use the system quickly. When hovering over some buttons, a brief message will appear to give a better explanation of the function of the button.</p> <p>See the appraisal questionnaires on page 7 of the appendix section.</p>
15. d) Most of the database forms should have similar Add, Edit, and Delete buttons so that the user can manipulate data easily.	<p><b>Solve Table:</b> Add Solve, Edit Solve, Delete Solve  <b>Competition Table:</b> Add Competition, Edit Competition Date, Delete Competition  <b>Member-Competition Window:</b> Add, Edit, Delete  <b>Member Table:</b> Add Member, Edit Member, Delete Member  <b>Algorithm Table:</b> Add Algorithm, Delete Algorithm. The user can double-click to edit a cell, so there is no edit button.</p>
16. a) The system should present error messages when the user performs an incorrect action, such as submitting invalid data.	<p>Error messages are displayed when the user makes an error. Error messages are commonly displayed in input forms where the user tries to submit invalid data, e.g. an invalid email. When these errors are made, an error message is shown with a red octagon with an 'x' in the middle, indicating that the user has made an error.</p>

Objective	System Functionality
16. b) The error messages should explain what the error is, and should give some indication of how to fix the problem.	<p>Error messages inform the user of the mistake and suggest possible ways of fixing the problem:</p>  <p>See the appraisal questionnaires on page 7 of the appendix section.</p>
17. The system should allow the user to adjust the preferences for the system, such as changing the rate at which automatic execution takes place.	<p>The user can manipulate the real-time solving speed, the inspection time, the size of the text for the scramble, and can choose whether or not a warning message is shown when click-to-solve mode is entered using the Preferences window (<i>Options → Preferences</i>).</p>
18. a) The system must have the ability to add, delete, and update member details.	<p>The user can manipulate all data relating to members using the Member Table window (<i>View → Show Member Table</i>).</p>
18. b) The following details about members should be stored: forenames, surname, gender, date of birth, email, and form class.	<p>All of these attributes can be stored about each member and are displayed in the table.</p>
18. c) There should be labelled text field boxes to enter this information with text indicating what the field is looking for.	<p>The following labels are positioned beside each of the text fields: 'Forenames', 'Surname', 'Gender', 'Date of Birth', 'Email', and 'Form Class'. These indicate exactly what is expected. Guidelines are given on how to enter data into these text fields on page 14 of the administrator user manual.</p>
19. The system must be able to create a table to display all details stored on the members in the club.	<p>The user can view all details stored on all members in the Member Table window (<i>View → Show Member Table</i>).</p>

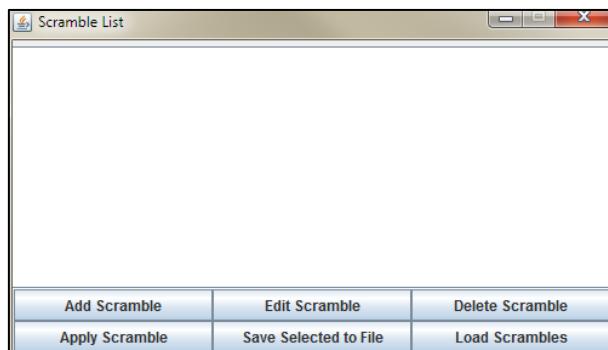
Objective	System Functionality
20. The system must have the ability to add, delete, and update competition details, e.g. the date of the competition.	The user can manipulate all data relating to competitions in the Competition Table window ( <i>View → Show Competition Table</i> ). To edit the competition date, the user can select a competition and then click the 'Edit Competition Date' button.
21. The system must be able to create a table showing the ID and date for each competition.	The information about each competition is shown in the Competition Table window ( <i>View → Show Competition Table</i> ) in a clear and resizeable table.
22. a) The system must have the ability to add, delete, and update member-competition details, i.e. the times achieved by users in a competition. These details should include member ID, competition ID, and the times achieved (time 1, time 2, ..., time 5).	The Member-Competition window ( <i>View → Show Competition Table → View Rankings</i> ) allows the user to manipulate all details about competitors in the selected competition. Using the Member-Competition Form, the member ID can be chosen, and the times can be entered along with validation checks being performed.
22. b) There should be labelled text field boxes to enter this information with text indicating what the field is looking for.	The following labels are positioned beside the text fields: 'Member ID', 'Time 1', 'Time 2', 'Time 3', 'Time 4', 'Time 5'. These indicate exactly what is expected. Guidelines are given on how to enter data into these text fields on page 14 of the administrator user manual.
23. a) The system must be able to create a table to display member-competition information showing all records saved for a selected competition.	The times achieved by each competitor in a selected competition can be shown in the Member-Competition window ( <i>View → Show Competition Table → View Rankings</i> ).
23. b) The 'average' of each record should be calculated as displayed automatically.	The average of the 5 times, i.e. the mean of the middle three times, is calculated and displayed in the same row as the member ID and times for that member.
23. c) The records should be sorted according to average.	The records are sorted with the best average at the top and the worst average at the bottom (all corresponding data is reordered). The rank is shown beside each average so that the user can quickly see the ranking of each average. If two averages are the same, then the individual times are compared to determine which average is the best.

Objective	System Functionality
24. a) The system should be able to filter solves in the solve table by the time attribute so that only the data that match the criteria will be shown.	The times in the Solve Table window ( <i>View → Show Solve Table</i> ) can be filtered so that only times between a specified upper and lower boundary will be shown in the table ( <i>Filter → Filter by Time</i> ).
24. b) The results should be returned within 0.5 seconds.	The results from the filter are returned instantly.
25. The system should be able to store up to 100 members.	The system can store many more members than this since the amount of data stored for each member is relatively small.
26. The system must be completed by April 2014.	The system was complete by 19 March 2014.
27. The system must be relatively bug-free.	<p>The only bugs that were found were fixed during the testing stage (see page 44 of the testing section). No other bugs have been reported by any users. This does not confirm that there are no bugs, but it indicates that the system is stable enough to work without common bug problems.</p> <p>See the appraisal questionnaires on page 7 of the appendix section.</p>
28. a) In order to realistically create the system I shall use high-level programming languages.	The two programming languages used (Java and SQLite) are both high-level.
28. b) I shall use Java to design and program the main system.	The system with which the user interacts was designed in Java, and all background functionality, such as generating solutions for the cube, were written in Java. The buttons, tables, menus etc. used for accessing data in the database were written in Java.
28. c) I shall use SQLite to manage, retrieve and store information from/to the database.	All database functionality, including creating, querying, and updating tables, was achieved using SQLite library through Java.

## Usability of the System

### Target Acquisition Time

Fitts' law states that the time required for a pointer to move rapidly to a particular area is linearly related to the logarithm of  $1 + \frac{D}{W}$  where  $D$  is the distance from the starting point to the centre of the target, and  $W$  is the width of the target measured along the axis of motion. The time taken is given by the formula  $T = a + b \log_2 \left( 1 + \frac{D}{W} \right)$  where  $a$  and  $b$  device-dependent constants. Analysis of the graphs of  $T$  against  $D$  with constant  $W$ , and  $T$  against  $W$  with constant  $D$  show that increase in length has a greater effect on the movement time than increase in width, meaning that the distance between the visual elements is generally more important than the width of the individual elements. When designing the windows and forms, I tried to arrange the buttons and text fields so that the most frequently used items were together and were in the most accessible region in the screen. For example, in the Scramble List window, the add scramble, edit scramble, and apply scramble features are the most commonly used, so they are placed at the bottom left of the window, and the other three buttons are placed at the bottom right of the window. The buttons are grouped, meaning that the available functions are easy to access and the distance between the important features is minimal, but their size is also reasonable so that the user does not experience frustration when trying to access a feature.



The Scramble List window's buttons are the smallest type of buttons in the entire system; all other buttons are approximately twice the height and 1.5 times the width of these buttons and are all placed close together in a grouped location. According to Fitts' law, this design will result in good HCI. Layout consistency is achieved since the buttons in every window are positioned at the bottom-centre of the window, meaning that the user will become familiar with the layout of the windows very quickly.

## Latency Time

Most buttons clicks and operations are performed with no noticeable delay. Even complex operations, such as sorting 500 solve records or generating a solution for a given cube state, are completed in negligible time. There are three operations that require significant time to complete their tasks:

- 1. Loading the ‘Terminology’ pdf document**

This takes time because the default pdf reader software on the user’s computer must be opened in order to view the document. While the program is loading, an hourglass is shown beside the mouse cursor , showing that the task is working in the background and may take some time.

- 2. Saving solves to the database**

The user can save solves to the database by using *File → Save Session to Database*, but if the number of solves in the session exceeds 100, then the time taken can be significant (> 1.5 seconds). While the solves are being saved, the cursor becomes a complete hourglass , indicating that the system is busy trying to complete a task. The cursor changes back to the default cursor once the task is finished.

- 3. Deleting solves from the database**

If the number of solves being deleted from the Solve Table window is large (> 100 records), then the time taken for this task to complete is noticeable. While the solves are being removed from the table and from the database, the mouse cursor changes to a complete hourglass , indicating that the system is busy trying to complete a task. The cursor changes back to the default cursor once the task is finished.

## Readability

There are only three different fonts used in the system: arial, courier new, and lucida sans. The majority of the text uses arial for the font so that consistency is preserved.

All text is clearly visible in each of the forms, error messages, and windows. The text in the cells of tables is sometimes hidden partially so that the rest of the data in the table can be seen, but the column can usually be resized to view the rest of the text, or the record can be opened in the corresponding editor so that the full text can be seen. For example, the solutions in the Solve Table window are typically too long to fit inside the cells of the table, so they are hidden partially as shown below (the ellipsis represents the hidden text):

ID	Time	Penalty	Comment	Scramble	Solution	Date Added
9	12.19	0		L' D' U2 R U' D ...	x2 U' L2 R' U R' ...	2014-03-08 20:...
39	12.20					2014-03-23 19:...
8	12.29	0	Easy → Difficult	F B' R2 U2 B F ...	y' x2 R' F2 D2 y...	2014-03-08 20:...
15	13.22	0	Fast	L2 U' R' D2 U2 ...	x2 L F' D' R D' ...	2014-02-13 14:...
37	13.22	0	Fast	L2 U' R' D2 U2 ...	x2 L F' D' R D' ...	2014-02-13 14:...
14	19.50	0		B R' B L' R' D2 ...	y' x2 y D L R' U...	2014-03-18 20:...
7	20.00					2014-03-18 20:...
44	21.20					2014-03-23 19:...
31	21.57	0		D2 L B2 F2 D2 ...	y2 x2 y' R2 F B' ...	2014-03-08 22:...
30	21.73	0		L D B D U F2 D ...	x2 y' D' U2 L F' ...	2014-03-08 22:...
6	31.11	0		B' F L' D2 L2 U' ...	x2 y' D' R F y' D...	2014-03-08 20:...
46	1:05.40					2014-04-06 12:...
47	1:30.60					2014-04-06 12:...

But the full text can be seen by double-clicking on the row of the table. For example, by double-clicking on the top row of the table, this window is shown with the full text for all attributes:

The screenshot shows a Windows-style dialog box titled "Solve Form". It contains the following fields:

- Time:** 12.19
- Penalty:** 0
- Comment:** (empty)
- Scramble:** L' D' U2 R U' D L2 R U' L' U' L D2 B2 D' B' R2 B2 F D2 R2 D' R2 D2 F
- Solution:** x2 U' L2 R' U R' F y' R U' R' y' U2 R' U' R U2 L' U' L U' R U2 R' y' U' R' U' R y' U2 y' R' U2 R U' R' U' R U M' U M U2 M' U M U2 R U' R U R U R U' R' U' R2 U'
- Date Added:** 2014-03-08 20:33:16

At the bottom right of the dialog is a blue "Submit" button.

Almost all text in the system is black on a light-grey background, meaning that the text is very easy to read. The text in the Color Selection window is black on white, yellow, red, orange, and green, and then white on blue, all of which are easily readable because the text size is quite large in this window.

## Navigability

Most windows can be accessed from the menu bar in the main window. When these other windows are opened, the main window stays open so that the user can always access the other parts of the system no matter what part of the system is being used. Getting out of any part of the system, e.g. the Solve Table window, is as easy as closing the window.

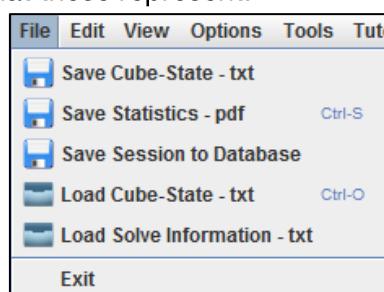
The main window can switch between timing mode and tutorial mode. To enter tutorial mode, the user can select a tutorial from the tutorial menu (*Tutorial → Desired tutorial*). To exit tutorial mode and re-enter timing mode, the user can use *Tutorial → Exit Tutorial*.

There are several shortcuts to speed up experienced users. In addition, editing functions can usually be accessed by double-clicking rather than clicking than selected the desired element then clicking the 'Edit' button.

Action	Shortcut
Start new solve	Alt + 1
Save statistics	Ctrl + S
Load cube state	Ctrl + O
Add solve	Ctrl + D
Edit selected solve	Ctrl + E
Cancel solve	Ctrl + Q
Solve cube	Ctrl + R
Time graph – Save as image	Ctrl + S
Time graph – Close window	Ctrl + W
Time graph – Reset zoom	Ctrl + R
Time graph – 2D	Ctrl + 2
Time graph – 3D	Ctrl + 3

The windows have a consistent layout with the buttons at the bottom of the window, the menus at the top, and any fields or tables centred in the window, allowing the user to quickly navigate through the system.

The menu items for saving and loading files have floppy disk and folder icons beside them to help the user understand what these represent.



## Possible Improvements/Extensions

I propose the following improvements/extensions:

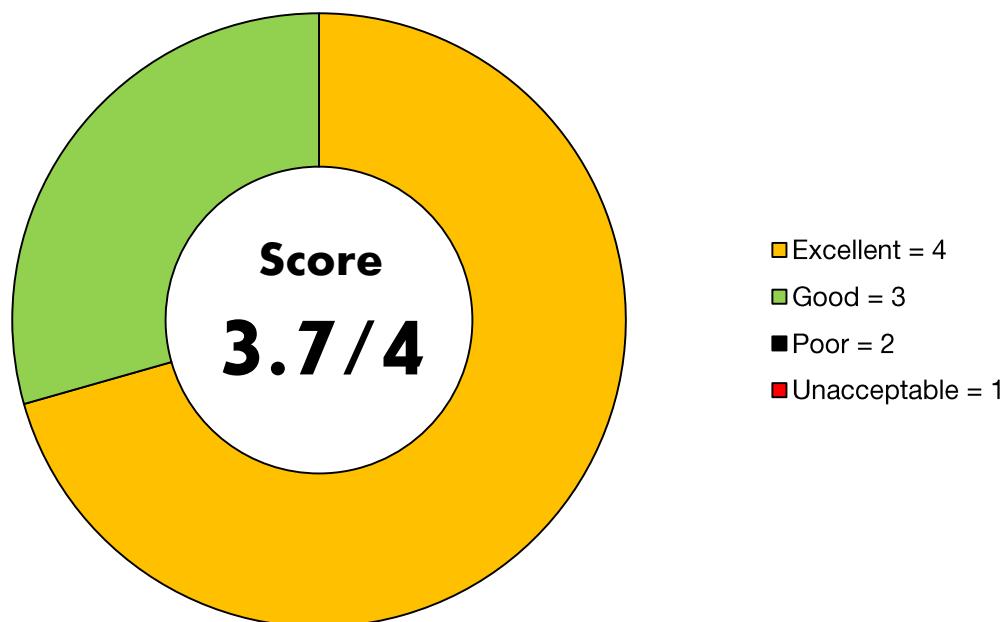
- The virtual cube, which is currently displayed as a 2D impression of a cube, could be changed so that it looks more realistic using OpenGL and could have actual motion when moves are being made rather than just the stickers updating.
- Currently, the size of the cube is fixed regardless of window size; this could be changed so that as the window size changes, the size of the cube changes accordingly. When developing the prototype for the virtual cube, I originally let Java resize the cube automatically, but unfortunately the image was too skewed and there were gaps between the rectangles/stickers. With a bit more knowledge of Java graphics, this would be an easy fix.
- Only the 3x3x3 Rubik's cube is supported, but there are many other sizes (2x2x2 – 11x11x11) which could be added later. Each could be implemented in hard code with the corresponding features, or some method of assembling an x-dimension cube could be devised since all regular cubes consist of a combination of edges, corners, and centres. For cube-dimensions higher than the 3x3x3 cube, a new 'Centre' class would have to be written.
- Features could be added so that custom statistics, e.g. average of 7, are available. This requires no change to the code used to calculate the statistics since the signature of the method allows for any size above 5 to be calculated. Only the user interface would need to be changed in some way to allow for these averages to be calculated.
- Rows in the Solve Table can only be filtered by time, but they could also be filtered by penalty or date. Java's built-in 'Date' class allows for comparisons to be made between two dates to determine which is the earlier date, so filtering the records in the table would be very easy – only the input parameters would have to be validated. Penalties could be filtered in the same way, and no real validation (other than integer type checks) would need to be performed.
- Backing up data could be done automatically on a regular basis to a user-specified location. This could be done by saving the date of the last update in the database or in a text file, and then every time the program opens, this date would be checked and it would be determined if the database would be copied to the location specified by the user in the preferences.
- Adding network capabilities to the system would simplify the recording of times in a competition. Currently, there is the 'Use Scramble in List' feature, which allows the user to use the 5 scrambles for the competition, but when these times are recorded, they must be given to a timekeeper so that the times can be recorded into the administrator's (Dr McIvor) system. With a networked system, these times could be sent to the member-competition table for that competition automatically.

- Different methods, e.g. Roux, ZZ, Petrus, CFOP, could be added so that the ‘Solve Cube’ feature (*Tools* → *Solve Cube*) could generate a solution for these methods. The structure of the SolveMaster classes could be changed slightly so that the solving method could be given as a string parameter and then the moves could be generated depending on which parameter is given.

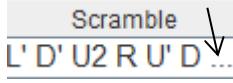
For improvements/extensions suggested by the end user, see the appraisal questionnaires on page 7 of the appendix section and *Analysis of User Feedback* on page 16.

## Analysis of User Feedback

To obtain user feedback, I provided Dr McIvor and two students with the installation materials, the appropriate user manuals, and the appraisal questionnaires. The ratings for the objectives, which were obtained from the appraisal questionnaires (see page 7 of the appendix section), are summarised in the diagram below:



Target acquisition time	
Comment	Response
Student 1: "The only part of the program that took me a while to understand was the algorithm window because I wasn't sure how to edit the cells in the table at first."	The user manual explains how to edit cells in the algorithm table, but help could be given in the window, such as tooltip text, which would tell the user to double-click to edit the cell.
Student 2: "The only window that takes a bit of time to get used to is the member-competition form because there are two other windows that you have to open before getting to this one (competition table and member-competition window)."	This stream of windows is almost unavoidable, unless there was a feature available in the main window, such as 'View Rankings for Competition' and then the user could enter the ID of the competition for which they want to view the rankings, but this only reduces the number of windows by one (no competition table).

Latency	
Comment	Response
Dr McIvor: "The only thing that is slightly slow is opening the terminology document ..."	This issue could be fixed by opening the document within Kuubik rather than relying on opening an external pdf reader.
Student 1: "... I noticed a bit of delay when I once saved 105 solves to the solve table ..."	SQLite needs more attention than other DBMSs for speed-optimal performance, so with further experience with SQLite I could probably speed up this process. In addition, the current implementation of the database aspects of the system were intended to be easier for development rather than for efficiency, so with modifications, these updates could be faster.
Readability	
Comment	Response
Dr McIvor: "Some of the text in certain windows could be made a bit bigger, or could maybe resize with the size of the window to make it more readable on larger screens."	Changing the size of text in windows is very easy and is really a matter of preference. Resizing text, however, requires more understanding of Java graphics, which is beyond the scope of my current knowledge.
Dr McIvor: "The only text that I feel is a bit confusing is the scramble and solution text in the Solve Table because it gets cut off."	These comments are referring to this:  <p>JTables do not provide automatic line wrapping, but a custom implementation of the TableCellRenderer interface can be written in order to deliver line wrapping. More research in this area would help me devise an implementation that would improve the display of the table.</p>
Student 2: "I can read all of the text in the program very clearly other than in the solve table window where the text looks a bit cluttered ..."	JTables do not provide automatic line wrapping, but a custom implementation of the TableCellRenderer interface can be written in order to deliver line wrapping. More research in this area would help me devise an implementation that would improve the display of the table.
Student 1: "... I think the text shown for the tutorials should have had more text in the vertical direction so that I wouldn't have to scroll so much during tutorials."	The layout of the main window was planned so that when the window is resized, the text does not get hidden, but changes could be made so that the text area is shown in the middle of the window rather than at the bottom so that less scrolling needs to be done. A feature could be added so that the tutorial text and buttons could be displayed in a separate window.
Student 2: "... it would be nice if I could change the font [for the scramble]."	This is a valid point; adding this feature would be very simple and useful. A section for changing the scramble font could be added with a drop-down list from which the user could choose the desired font.

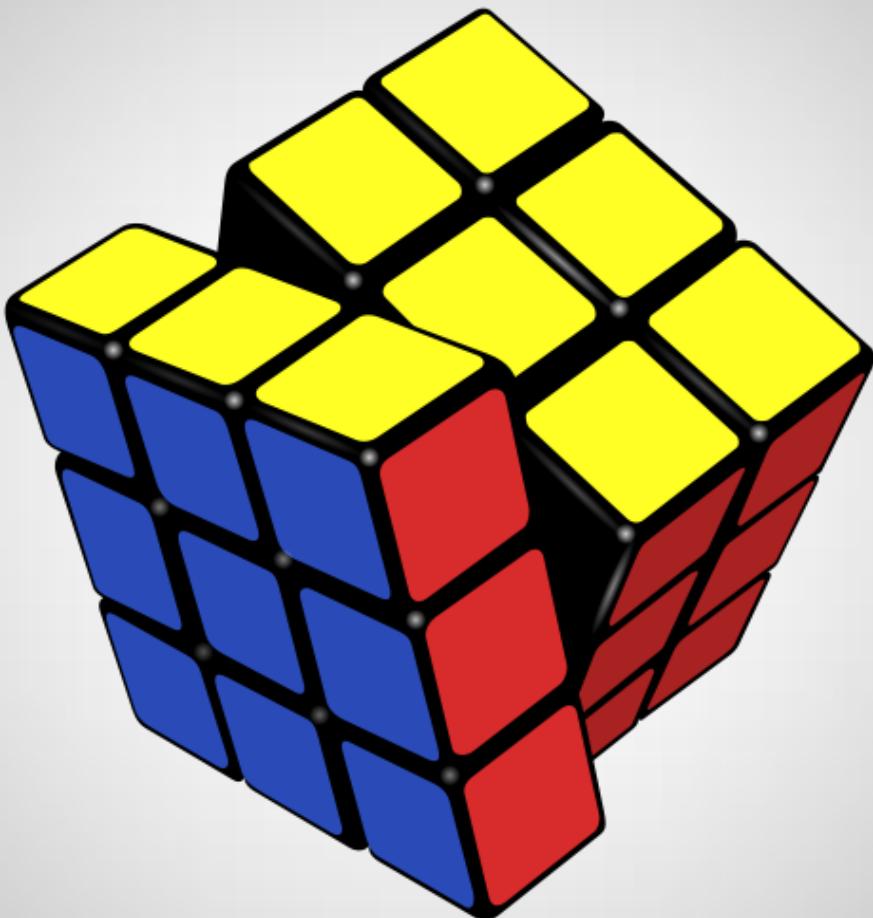
Navigability	
Comment	Response
Dr McIvor: "Individually, the windows are clearly displayed, but when multiple windows are open, they can sometimes hide behind each other when the main window is open."	Window arrangement can be improved by using JDialogs rather than JFrames so that the hierarchy of windows is more apparent and enforced. For example, this would mean that all Kuubik windows would be minimised when the main window is minimised.
Student 2: "...it can be confusing sometimes when windows like the competition table are in front of the member-competition table ..."	In general, pop-up menus are better than pull-down menus, so this is an important comment. Icons could be added to the main window that, when clicked, would bring the user to the relevant part of the system. Employing this idea would be feasible.
Other comments and suggested improvements	
Comment	Response
Dr McIvor: "Could the icon for each of the windows be changed to an image of a Rubik's cube rather than the cup of coffee?"	After receiving the final user feedback, I changed the image so that it now appears like this: 
Dr McIvor: "When I resize the window below a certain size, some of the cube gets hidden."	Future versions of Kuubik using a better graphics implementation and/or OpenGL will solve this problem. (See <i>Possible Improvements/Extensions</i> on page 14 for more information).
Dr McIvor: "Only WCA notation is accepted, but SiGN (Simple General Notation) notation is very common, so maybe that could be included somehow."	SiGN is a very popular notation, but it is usually used for higher-dimension cubes, such as 4x4x4 and 5x5x5, so I overlooked this feature, but including this feature would help attract more users, and would be vital if later versions support higher-dimension cubes.
Dr McIvor: "A more interactive process for learning the controls would be advantageous."	I agree that the tutorial mode is not quite sufficient for helping the user to learn the controls. A slightly different process could be developed in order to help the user learn the controls quickly.
Dr McIvor: "I cannot undo any of my actions in the program, such as deleting a piece of information."	This is an important subject because the user should always feel safe when using the software, and if they are worried about losing data forever, then this is a big disadvantage. Java provides some advanced features for undoing/redoing certain operations, but most of the functionality needs to be custom-developed. This is a must for the next version of Kuubik.
Student 1: "Improvement: undo feature."	

Dr McIvor: "When I view the execution of a solve, I cannot pause or rewind, I just have to start from the beginning of the execution if I want to view a certain section of the solve again."	This would certainly be a nice feature to include, but it would probably be easier for the user to just perform the moves on the virtual cube or a physical cube. This could be done by recording the current position of the move being performed and then applying the forward or reversed moves as required.
Student 1: "Improvement: smooth animation for the cube."	For more experienced users, the instant turning is an advantage, but for beginner or intermediate users, this effect could be considered confusing. Future versions of Kuubik could use OpenGL to render the cube rather than the 2D Java graphics so that this more complex animation can take place.
Student 2: "It took a while to get used to the animation because the faces of the cube don't turn like a real cube ..."	This would be an important feature to add so that users who are uncomfortable with the default key-mapping could create a custom, setup that they can use more effectively. This would be another section to add to the preferences and then the program would use this key-mapping to determine which moves to perform.
Student 1: "Improvement: allow custom keys to be used for turning the cube."	This would be an enhancement that would be very easy to implement because there is a constant in the 'Main' class that stores the number of times to graph. I believe the best method of employing this solution would be to use a slider in the time graph window so that the number of times to graph can be chosen.
Student 1: "Improvement: graph more times in the time graph window."	The backing up process is entirely manual, which should be changed to an automatic procedure so that the user can select a location to which the database should be backed up and specify how often it should be backed up, e.g. once every week.
Student 2: "I wasn't sure if the program was actually opening after I clicked the double-clicked the icon. Maybe there could be a welcome image similar to the microsoft word's start-up image."	Splash screens are perfect for programs that have long loading times. I am researching how to configure manifest files so that an animated splash screen can be shown while the program is opening.
Student 2: "Some form of algorithm trainer would be handy so that the cube is scrambled randomly from a set of scrambles. The 'use scrambles in list' option allows me to scramble the cube using custom scrambles, but they are not in a random order."	This would be very useful for beginners because learning algorithms is a difficult task. This would require only the addition of a random number generator, which Java provides, to choose the next scramble for the cube.

Student 2: "I was surprised when I re-opened the program and found that the scrambles in the scramble list were not there. I read the manual and I now know that the data is not saved in this window, but it would have been nice to have a warning about this."	There could be a warning when the user tries to close the program that informs the user about the loss of data in the scramble list and in the main window.
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------

SECTION 7

# APPENDIX



## Contents

Analysis Questionnaire 1.....	3
Analysis Questionnaire 2.....	5
Appraisal Questionnaire – Administrator .....	7
Appraisal Questionnaire – Student 1 .....	12
Appraisal Questionnaire – Student 2 .....	18
Interviews .....	22
Interview 2 .....	22
Interview 3 .....	23
Interview 4 .....	24
Interview 5 .....	26
Interview 6 .....	28
Source Code.....	29
Class Algorithm .....	29
Class AlgorithmDatabaseConnection.....	31
Class AlgorithmDatabasePopUp .....	36
Class ColorSelection .....	44
Class Competition.....	50
Class CompetitionDatabaseConnection.....	54
Class CompetitionDatabasePopUp .....	58
Class Corner.....	67
Class CornerSolver.....	70
Class CrossSolver .....	78
Class Cube .....	91
Class Cubie .....	108
Class CustomPdfWriter.....	115
Class Edge .....	118
Class EdgeSolver .....	122
Class LinearSearch .....	129
Class Main.....	132
Class Member .....	217
Class MemberCompetition .....	223
Class MemberCompetitionDatabaseConnection .....	228
Class MemberCompetitionDatabasePopUp.....	232
Class MemberDatabaseConnection .....	253
Class MemberDatabasePopUp.....	258
Class MouseSelectionSolver.....	275
Class OrientationSolver.....	288

Class PermutationSolver.....	293
Class Preferences .....	299
Class Scramble .....	311
Class ScramblePopUp.....	313
Class Slice.....	323
Class Solve.....	331
Class SolveDatabaseConnection.....	342
Class SolveDatabasePopUp.....	347
Class SolveDBType.....	376
Class SolveMaster.....	378
Class Sorter.....	402
Class Statistics.....	410
Class TextFile .....	423
Class TimeGraph.....	430
Class TimeListPopUp.....	439
Class Tutorial.....	451
Procedure List.....	466

## Analysis Questionnaire 1

### Kuubik Analysis Questionnaire (Students)

What is the primary operating system you use? (Please tick one box)

- Windows 7
- Windows Vista
- Windows XP
- Mac OS
- Linux
- Other (please specify): \_\_\_\_\_

What information would you like to be able to store in the system?

- Storing algorithms would be helpful so that I can keep track of all the algorithms I should know.
- I've tried recording my solutions for solves, but it is very difficult because I have to reconstruct the moves after videoing the solve, so it would be good to have the scramble and solution stored automatically.

Do you find the current method of learning (using algorithm sheets etc.) effective and why?

- It's not very effective. I have to ask questions a lot because the information on the sheet isn't always clear. In addition, it can be a slow process because the algorithm sheets say 'perform these moves to solve this piece', but I have to perform the ~~the~~ algorithm in reverse so that I can actually practice the case.

What utilities/features would you like to see in the system?

- Sometimes I'm not sure how to solve a particular piece because it is positioned awkwardly, so it would be helpful if I could generate a solution for a particular piece.
- finding examples of the method we're taught is very difficult, so if the system could generate examples/solutions for cubes then that would be a great resource.

How could the current system be improved?

- I would love to see new approach / Learning which would be interactive and have more examples and explanations for the difficult sub-steps. Automatic scrunching for this would be especially helpful.
- The timing websites don't save my times forever, and there's no easy way to save these times to my computer

Are there any automated procedures that need to be carried out, e.g. calculations?

- I am very interested in the statistics of my times. The most important two are 'average' of 5' and 'average of 12'
- My friends sometimes send my interesting scrambles and solves over the internet, so I would need to be able to apply scrambles and view solves executed automatically.

Thank you for your time.

Signature: David Zy Date: 23/10/2013

## Analysis Questionnaire 2

### Kuubik Analysis Questionnaire (Students)

What is the primary operating system you use? (Please tick one box)

- Windows 7
- Windows Vista
- Windows XP
- Mac OS
- Linux
- Other (please specify): \_\_\_\_\_

What information would you like to be able to store in the system?

Details about solves, especially the time and solution for the solves, and maybe a comment for the penalties, e.g. "PLL ship".  
I find it difficult to keep track of the algorithms we're supposed to know, so having some method of saving the algorithms with a description of what they do would be useful.  
Being able to look it up quickly that I find on the internet would be very helpful

Do you find the current method of learning (using algorithm sheets etc.) effective and why?

It's quite good because I can understand the instructions most of the time, but other times I get lost because the images and the text don't seem to make sense together. Scrambles aren't provided for the cases shown on the handouts, so it's hard to visualise what the instructions are explaining.

What utilities/features would you like to see in the system?

Automatic Scrubbling would mean I could do many more solves per day than at the moment.  
I share solves and scrambles with people over the Internet quite often so if there could be an option to export the details for a particular solve then this would be much easier. I have to copy + paste the statistics from websites and save them in a text file if I want to save the stats for a set of times, but this is frustrating.  
When I learn a new algorithm, it's a slow process practising it because I have to perform the algorithm backwards & then perform the algorithm forwards to see if I'm performing it properly. So if algorithms could be automatically reversed then I could learn algorithms much more quickly.  
Being able to generate the moves to solve a cube would be invaluable.

How could the current system be improved?

Some new routine for learning how to solve a cube  
Saving times over a long period of time could be made less awkward  
Entering penalties done automatically  
If I want to share a cube state with someone without the Scramble, I have to solve the cube and write out the moves in reverse so that they can scramble the cube themselves. So I need to consider methods or tips on how to do this.

Are there any automated procedures that need to be carried out, e.g. calculations?

Stats: best average of 5, 12, 50, 100 and the current value of these averages  
Scrambling the cube automatically  
Generating solutions for pieces and for the entire cube  
I would like to be able to sort by solved solves so that I can see the details of my fastest solves.

Thank you for your time.

Signature: R Day Date: 23/10/13

## Appraisal Questionnaire – Administrator

### Kuubik Appraisal Questionnaire (Administrator)

Please rate the Kuubik system on the following points by ticking the appropriate box:

Objective Index		Excellent	Good	Poor	Unacceptable
1	Performance on Windows 7.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2	Recording solves.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3	The interactive learning process (tutorials).	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4	Loading information about other solves into the system.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5a	Calculating and displaying statistics.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5b	Saving statistics to pdf.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
6	Time to open the program.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
7	Storing solve information, algorithms etc.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8a	The time graph.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8b	Saving images of the time graph.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
9	Automatic execution of solves listed in the main window.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
10	Generating solutions to given cube states.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
11	Importing/exporting cube states.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
12	Generating solutions for a selected piece.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
13	Painting custom states using the Color Selection window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Objective Index		Excellent	Good	Poor	Unacceptable
14a	Saving the current session to the database.	✓			
14b	The display and sorting features in the Solve Table window.		✓		
15a	Methods of navigation.		✓		
15b	Clarity of menu items.	✓			
15c	Clarity of buttons.	✓			
15d	Consistency of database forms.	✓			
16a	Error messages shown in appropriate situations.	✓			
16b	Error messages give an indication of how to solve the problem.	✓			
17	Changing preferences.	✓			
18a	Adding, deleting, and updating member details.	✓			
18b	Storage of forenames, surname, gender, date of birth, email, and form class of members.	✓			
18c	Clarity of labels when entering member details.	✓			
19	Quality of Member Table window.		✓		
20	Adding, deleting, and updating competition details, e.g. the date of the competition.	✓			
21	Quality of Competition Table window.	✓			
22a	Adding, updating, and deleting member-competition details.		✓		
22b	Clarity of labels when entering member-competition details.	✓			
23a	Quality of Member-Competition window.	✓			

Objective Index		Excellent	Good	Poor	Unacceptable
23b	Calculation and display of the 'average of 5' for each record in the Member-Competition window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
23c	Sorting of the records in the Member-Competition window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
24a	Filtering of the records in the Solve Table window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
24b	Time to return filtering results in the Solve Table window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
25	Storage of many members' details.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
26	Punctuality of system completion.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
27	Bug-free environment.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please comment on the following aspects of the system:

Ease of use  
when trying to  
access a given  
feature

ALL FEATURES ARE EASY TO ACCESS AS THEY ARE LOCATED IN INTUITIVE PLACES, E.G. 'SOLVE CUBE' IS IN THE 'TOOLS' MENU, AND BECAUSE THE BUTTONS AND MENUS ARE LARGE ENOUGH TO CLICK EASILY. THE PLACING OF BUTTONS AT THE BOTTOM OF EVERY WINDOW MEANS THAT IT'S EASY TO MOVE FROM ONE WINDOW TO ANOTHER.

Latency

THE ONLY THING THAT IS SLIGHTLY SLOW IS OPENING THE TERMINOLOGY DOCUMENT BUT OTHER THAN THAT, WHEN I CLICK A BUTTON OR SELECT A MENU OPTION, THE EFFECT TAKES PLACE INSTANTLY.

Readability

THE FONTS USED ARE VERY READABLE IN TERMS OF SIZE AND STYLE. SOME OF THE TEXT IN CERTAIN WINDOWS COULD BE MADE A BIT BIGGER OR COULD MAYBE RESIZE WITH THE SIZE OF THE WINDOW TO MAKE IT MORE READABLE ON LARGER SCREENS. THE ONLY TEXT THAT I FEEL IS A BIT CONFUSING IS THE SCRAMBLE AND SOLUTION TEXT IN THE 'SOLVE TABLE' BECAUSE IT GETS CUT OFF. THIS TEXT COULD MAYBE BE BROUGHT OVER SEVERAL LINES INSTEAD OF CUT OFF ON A SINGLE LINE.

Navigability

I DIDN'T FEEL LOST AT ANY POINT WHEN USING THE SYSTEM BECAUSE ALL OF THE OPTIONS WERE MADE VERY CLEAR IN EACH OF THE WINDOWS. THE SHORTCUTS ARE A NICE ADDITION BUT THERE MAYBE COULD HAVE BEEN MORE OF THEM. INDIVIDUALLY, THE WINDOWS ARE DISPLAYED CLEARLY BUT WHEN MULTIPLE WINDOWS ARE OPEN, THEY CAN SOMETIMES HIDE BEHIND EACH OTHER, ESPECIALLY BEHIND THE MAIN WINDOW.

**Other comments and suggested improvements**

- COULD THE ICON FOR EACH OF THE WINDOWS BE CHANGED TO AN IMAGE OF A RUBIK'S CUBE RATHER THAN THE CUP OF COFFEE?
- WHEN I RESIZE THE WINDOW BELOW A CERTAIN SIZE SOME OF THE CUBE GETS HIDDEN.
- ONLY WCA NOTATION IS ACCEPTED BUT SIGN (SIMPLE GENERAL NOTATION) NOTATION IS VERY COMMON, SO MAYBE THAT COULD BE INCLUDED SOMEHOW.
- A MORE INTERACTIVE PROCESS FOR LEARNING THE CONTROLS WOULD BE ADVANTAGEOUS.
- I CANNOT UNDO ANY OF MY ACTIONS IN THE PROGRAM, SUCH AS DELETING A PIECE OF INFORMATION.
- WHEN I VIEW THE EXECUTION OF A SOLVE I CANNOT PAUSE OR REWIND; I JUST HAVE TO START FROM THE BEGINNING OF THE EXECUTION IF I WANT TO VIEW A CERTAIN SECTION OF THE SOLVE AGAIN.

Thank you for your time.

Signature: Jm A [initials] Date: 25/03/14.

## Appraisal Questionnaire – Student 1

### Kuubik Appraisal Questionnaire (Students)

Please rate the Kuubik system on the following points:

Objective Index		Excellent	Good	Poor	Unacceptable
1	Performance on Windows 7.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2	Recording solves.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3	The interactive learning process (tutorials).	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4	Loading information about other solves into the system.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5a	Calculating and displaying statistics.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5b	Saving statistics to pdf.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
6	Time to open the program.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
7	Storing solve information, algorithms etc.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8a	The time graph.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8b	Saving images of the time graph.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
9	Automatic execution of solves listed in the main window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
10	Generating solutions to given cube states.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
11	Importing/exporting cube states.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
12	Generating solutions for a selected piece.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
13	Painting custom states using the Color Selection window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
14a	Saving the current session to the database.	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
14b	The display and sorting features in the Solve Table window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Objective Index		Assessment Scale			
		Excellent	Good	Poor	Unacceptable
15a	Methods of navigation.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
15b	Clarity of menu items..	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
15c	Clarity of buttons.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
15d	Consistency of database forms.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
16a	Error messages shown in appropriate situations.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
16b	Error messages give an indication of how to solve the problem.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
17	Changing preferences.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
24a	Filtering of the records in the Solve Table window.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
24b	Time to return filtering results in the Solve Table window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
26	Punctuality of system completion.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
27	Bug-free environment.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

## Kuubik Appraisal Questionnaire (Students)

Please rate the Kuubik system on the following points:

Objective Index		Excellent	Good	Poor	Unacceptable
1	Performance on Windows 7.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2	Recording solves.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3	The interactive learning process (tutorials).	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4	Loading information about other solves into the system.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5a	Calculating and displaying statistics.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5b	Saving statistics to pdf.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
6	Time to open the program.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
7	Storing solve information, algorithms etc.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8a	The time graph.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8b	Saving images of the time graph.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
9	Automatic execution of solves listed in the main window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
10	Generating solutions to given cube states.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
11	Importing/exporting cube states.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
12	Generating solutions for a selected piece.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
13	Painting custom states using the Color Selection window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
14a	Saving the current session to the database.	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
14b	The display and sorting features in the Solve Table window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Objective Index		Assessment Scale			
		Excellent	Good	Poor	Unacceptable
15a	Methods of navigation.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
15b	Clarity of menu items..	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
15c	Clarity of buttons.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
15d	Consistency of database forms.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
16a	Error messages shown in appropriate situations.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
16b	Error messages give an indication of how to solve the problem.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
17	Changing preferences.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
24a	Filtering of the records in the Solve Table window.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
24b	Time to return filtering results in the Solve Table window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
26	Punctuality of system completion.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
27	Bug-free environment.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please comment on the following aspects of the system:

Ease of use  
when trying to  
access a given  
feature

It's very easy to access all the features because most of them are available in the menu bar of the Kubit window or can be found by opening one other window. The only part that took me a while to understand was the algorithm window because I wasn't sure how to edit the cells in the table at first.

Latency

I can't notice any delay in most parts of the program. Even when I sort hundred of rows in the value table, I can't see any latency between clicking the sort button and when they are actually sorted. Opening the help page takes a bit of time though, and I noticed a delay when I was saving 100 solves to the solve table.

Readability

I didn't have any trouble reading any of the text in the program, and all of the numbers were displayed very clearly and not too small, but I think the text shown for the tutorial should have rotated in the vertical direction so that I wouldn't have to scroll so much during tutorials.

Navigability

I like that all parts of the program can be accessed from the same place but I would prefer to have a menu in the middle of the screen or something similar rather than using the dropdown menus.

**Other comments and suggested improvements**

- Improvement: undo feature.
- Improvement: smooth animations for the cube.
- Improvement: allow custom keys to be used for turning the cube.
- Improvement: graph more lines in the time graph window.
- I find backing up the database is a bit tedious with the current method.

Thank you for your time.

Signature: Thunwut Date: 25/03/2014

## Appraisal Questionnaire – Student 2

### Kuubik Appraisal Questionnaire (Students)

Please rate the Kuubik system on the following points:

Objective Index		Excellent	Good	Poor	Unacceptable
1	Performance on Windows 7.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2	Recording solves.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3	The interactive learning process (tutorials).	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4	Loading information about other solves into the system.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5a	Calculating and displaying statistics.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5b	Saving statistics to pdf.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
6	Time to open the program.	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
7	Storing solve information, algorithms etc.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8a	The time graph.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8b	Saving images of the time graph.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
9	Automatic execution of solves listed in the main window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
10	Generating solutions to given cube states.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
11	Importing/exporting cube states.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
12	Generating solutions for a selected piece.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
13	Painting custom states using the Color Selection window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
14a	Saving the current session to the database.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
14b	The display and sorting features in the Solve Table window.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Objective Index		Excellent	Good	Poor	Unacceptable
15a	Methods of navigation.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
15b	Clarity of menu items.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
15c	Clarity of buttons.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
15d	Consistency of database forms.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
16a	Error messages shown in appropriate situations.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
16b	Error messages give an indication of how to solve the problem.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
17	Changing preferences.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
24a	Filtering of the records in the Solve Table window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
24b	Time to return filtering results in the Solve Table window.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
26	Punctuality of system completion.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
27	Bug-free environment.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please comment on the following aspects of the system:

Ease of use  
when trying to  
access a given  
feature

It is quite ~~easily~~ easy to access each feature,  
especially the ones that are available in the  
main window. The only window that takes a  
bit of time to get used to is the member-  
competition form because there are two other  
windows that you have to open before  
getting to this one (competition table and  
member-competition window)

Latency

I can't spot any latency in the program  
program other than the time taken to start up  
the software.

Readability

I can read all of the text in the program very  
clearly other than in the solve table window  
where the text looks a bit cluttered, but I  
suppose this is unavoidable with so much  
information. It's good that the size of the  
text for the scramble can be changed,  
but it would be nice if I could change the  
font.

Navigability

It's straightforward getting from one window  
to another, and entering and exiting  
windows is simple as well. However, it can  
be a bit confusing sometimes when windows  
like the competition table are in front of  
the member-competition table because  
I feel like the member-competition window  
should always be in front.

## Other comments and suggested improvements

- I wasn't sure if the program was actually opening after I double-clicked the icon.  
maybe there ~~is~~ could be a welcome image similar to the microsoft word's start-up image.
- Some form of algorithm trainer would be handy so that the cube is scrambled randomly from a set of scrambles. The 'use scrambles in list' option allows me to scramble the cube using custom scrambles, but they are not in a random order.
- I was surprised when I re-opened the program & found that the scrambles in the scramble list were not there. I read the manual & I now know that the data is not saved in this window, but it would have been nice to have a warning about this.
- It took a while to get used to the animation because the faces of the cube don't look like a real cube, but I'm sure that with more time and/or experience this could be designed like a real Rubik's cube.

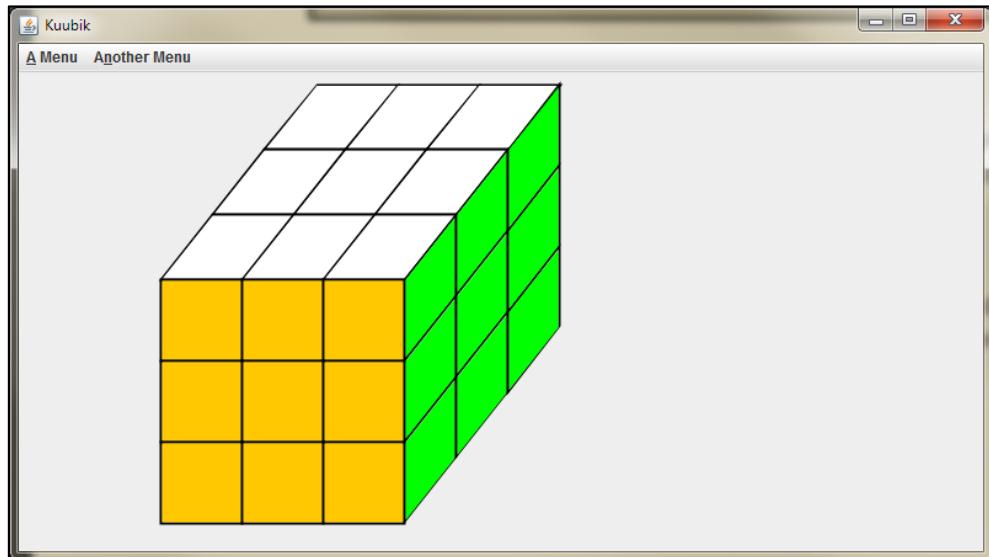
Thank you for your time.

Signature: (Signature) Date: 25/3/14

## Interviews

### Interview 2

This interview was carried out on 20 November 2013 after I had developed a prototype for the Kuubik system. The main objective of this interview was to establish the user's thoughts on the design and functionality of the virtual cube.



**What are your thoughts on the display of the virtual cube?**

**Dr McIvor:** "It displays exactly like I wanted it to. The colours are bright, the stickers are large enough, and the angle of the cube should allow for speedy solving."

**Does the functionality of the cube meet your expectations?**

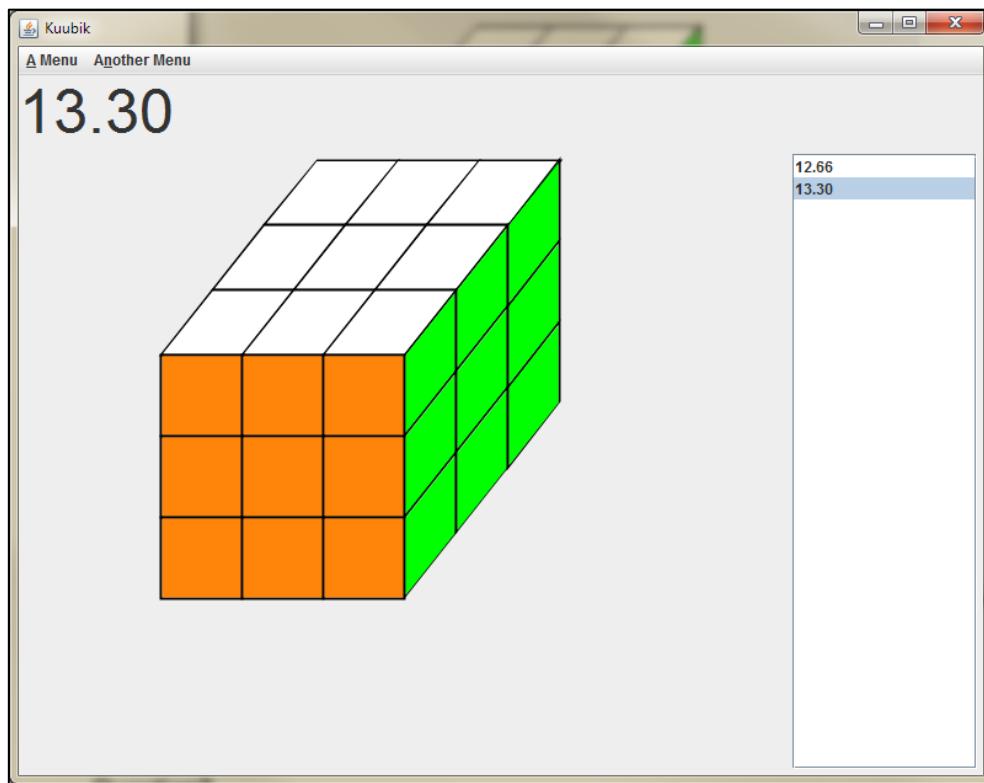
**Dr McIvor:** "The cube can perform all the basic moves required to solve the cube, but I would like it if it could also do wide and middle slice moves like Lw, Lw', Rw, Rw', M and M'."

**Any other comments?**

**Dr McIvor:** "The orange stickers are a bit too similar to the yellow stickers, which could cause recognition difficulties when turning the cube quickly. You could fix this by making the orange stickers darker so that there is a higher contrast between the two colours."

## Interview 3

This interview was carried out on 29 November 2013 after I had made the changes requested by the user and added a timer and a list to store the times.



What do you think of the display of the program?

**Dr McIvor:** “I like the change to the orange stickers, it contrasts well with the yellow stickers. The layout is perfect, but the size of the text in the list is a bit small, could it be made larger?”

Are the additional wide and middle moves working the way you expected?

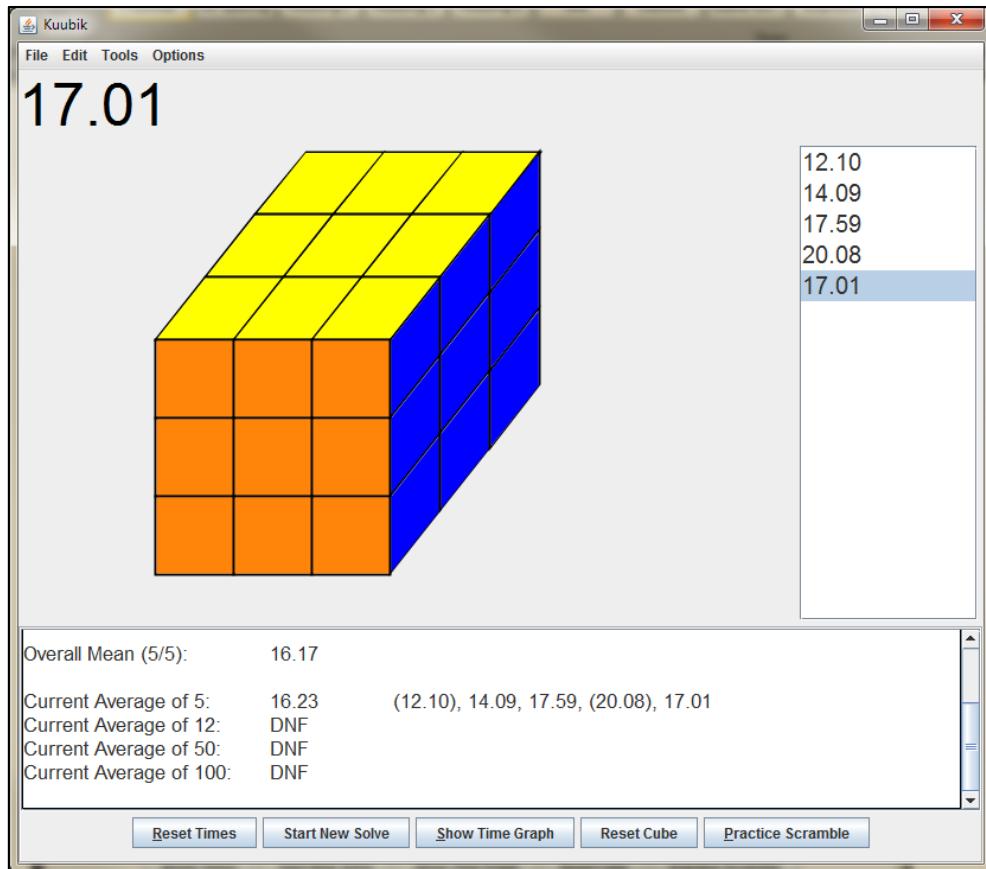
**Dr McIvor:** “They work perfectly and were quite intuitive for me to learn.”

Any other comments?

**Dr McIvor:** “Just remember to keep adding the features for the timer and the list so that times can be edited and deleted and so on, and so that there will be an inspection timer before a solve starts.”

## Interview 4

This interview was carried out on 30 December 2013 after I had made the changes requested by the user and added an inspection timer, editing and deleting options for the listed times, statistics calculation, solution generation, the time graph, and the scramble generator. Some of the buttons were placed with the intention of making developing easier rather than an indication of the final system's layout.



**What do you think of the display of the program?**

**Dr McIvor:** "The size of the text in the list is perfect now. The way the statistics are displayed is very clear and readable. The inspection timer works as I expected, and the timer starts when it is supposed to. The editor form for solves is arranged well. Some of the buttons and menu items aren't laid out the way I had expected, but I hope this will become closer to the planned layout when the project comes closer to completion."

**Are the generated solutions as you expected?**

**Dr McIvor:** "I love how it is formatted, and it is amazing that an English explanation is given for each sub-step. I think this will help the students learn very quickly."

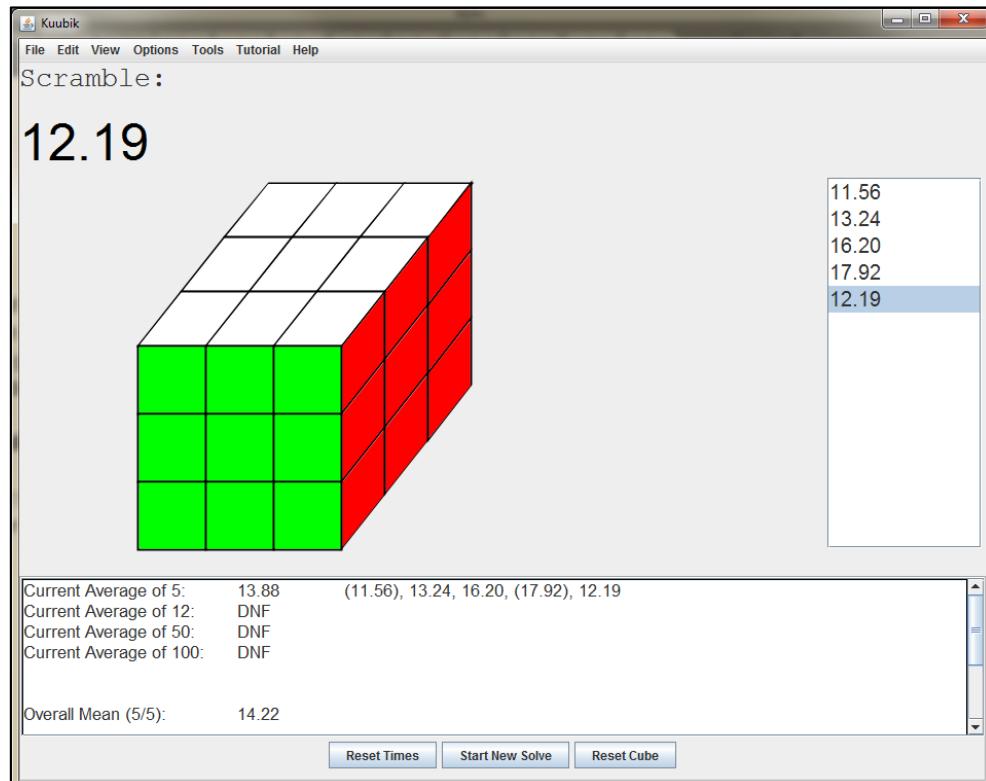
**Any other comments?**

**Dr McIvor:** "There need to be a button that allows the solves to be saved to the computer and another button to allow the solve to be executed automatically. I would also prefer it if the scramble for each solve was shown by default at the top of the screen. Also, currently

the timer can only be started when the inspection timer is running, but I would like it if the timer could be started manually so that, for example, the time taken to perform an algorithm could be measured and recorded.”

## Interview 5

This interview was carried out on 7 January 2013 after I had made the changes requested by the user, added windows for the algorithm table, the solve table, the scramble list and preferences, and I added the tutorial mode.



Solve List						
ID	Time	Penalty	Comment	Scramble	Solution	Date Added (yyyy-mm-dd ...)
1	0.15	0	*			2014-01-06 22:58:48
2	0.20	0	*			2014-01-06 22:17:36
3	0.26	0				2014-01-06 22:58:48
4	0.58	0				2014-01-06 22:58:48
5	0.90	0				2014-01-06 22:58:48
6	6.50	0	*			2014-01-06 22:17:36
7	54.20	0	*			2014-01-06 22:17:36
8	1:04.50	0	*			2014-01-06 22:17:36
9	2:05.10	0	*			2014-01-06 22:17:36

At the bottom of the window are three buttons: 'Add Solve', 'Delete Solve', and 'Load into Program'.

Are you happy with the modified layout?

**Dr McIvor:** "It is good that the scramble is shown at the top of the window now, and the layout of the buttons and other windows has been changed to my preferred options, including the editor form for solves, which is excellent."

**Are you happy with the algorithm, scramble, and solve tables?**

**Dr McIvor:** "Most of the features are now available in these windows, but one comment I would give is that the size of the text is a bit too small."

**Does the tutorial mode operate properly?**

**Dr McIvor:** "It works better than expected and has good response messages when the problems are solved. Make sure to write tutorials for all steps, not just the cross."

**Any other comments?**

**Dr McIvor:** "More validation checks need to be added to the solve table window so that invalid data can't be entered into the table."

## Interview 6

This interview was carried out on 10 February 2013 after I had made the changes requested by the user, added windows for the member table, the competition table, and the member-competition table.

The image displays three separate windows from a Windows application:

- Competition Table:** A window titled "Competition Table" showing a table with two rows. The first row has ID 1 and Date 25/12/2013. The second row has ID 2 and Date 25/12/2000. Below the table are buttons for "Add Competition", "Edit Competition", "View Rankings", and "Delete Competition".
- Member-Competition:** A window titled "Member-Competition" showing a table with five rows. The columns are Competition ID, Member ID, Rank, Average, Time 1, Time 2, Time 3, Time 4, and Time 5. Below the table are buttons for "Add", "Edit", and "Delete".
- Member Table:** A window titled "Member Table" showing a table with three rows. The columns are ID, Forenames, Surname, Gender, Date of Birth, Email, and Form Class. Below the table are buttons for "Add Member", "Edit Member", and "Delete Member".

What do you think of the three new windows?

**User:** “The member table and the competition table display exactly how I want them to, but the member competition table could be made a bit better by not repeating the numbers in the competition ID column because all the numbers in that column will be the same. Also, please show the names in the member-competition table beside the member IDs so that it is easier to identify the member.”

## Source Code

### Class Algorithm

```
package jCube;

/**
 * @author Kelsey McKenna
 */
public class Algorithm {
    /**
     * The ID of the algorithm in the database
     */
    private int algorithmID;
    /**
     * The moves of the algorithm
     */
    private String moveSequence;
    /**
     * The comment associated with the algorithm. This is usually a name, such as
     * 'Sledgehammer'
     */
    private String comment;

    /**
     * Constructor allowing the fields of the class to be initialised
     *
     * @param algorithmID
     *         an integer representing the algorithmID of the algorithm
     * @param moveSequence
     *         a string holding the moves in the algorithm
     * @param comment
     *         a string holding the comment for the algorithm - this should
     *         usually be a name such as 'sledgehammer'
     */
    public Algorithm(int algorithmID, String moveSequence, String comment) {
        this.algorithmID = algorithmID;
        this.moveSequence = moveSequence;
        this.comment = comment;
    }

    /**
     * @return an integer representing the ID of the algorithm
     */
}
```

```
public int getAlgorithmID() {
    return algorithmID;
}

/**
 * @param algorithmID
 *         an integer representing the ID of the algorithm
 */
public void setAlgorithmID(int algorithmID) {
    this.algorithmID = algorithmID;
}

/**
 * @return a string containing the moves of the algorithm
 */
public String getMoveSequence() {
    return moveSequence;
}

/**
 * @param moveSequence
 *         a string containing the moves of the algorithm
 */
public void setMoveSequence(String moveSequence) {
    this.moveSequence = moveSequence;
}

/**
 * @return a string containing the comment for the algorithm
 */
public String getComment() {
    return "" + comment;
}

/**
 * @param comment
 *         a string containing the comment for the algorithm
 */
public void setComment(String comment) {
    this.comment = "" + comment;
}

}
```

## Class AlgorithmDatabaseConnection

```
package jCube;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * @author Kelsey McKenna
 */
public class AlgorithmDatabaseConnection {

    /**
     * Stores the SQL update-strings required to insert the preset algorithms
     * into the 'algorithm' table
     */
    public static final String[] PRESET_ALGORITHM_UPDATES = {
        String.format("INSERT INTO algorithm(moveSequence, comment) " + "VALUES(\"%s\", \"%s\")", "R U R' U'",
                      "Fast Move (uneditable)"),
        String.format("INSERT INTO algorithm(moveSequence, comment) " + "VALUES(\"%s\", \"%s\")", "R' F R F'",
                      "Sledgehammer (uneditable)"),
        String.format("INSERT INTO algorithm(moveSequence, comment) " + "VALUES(\"%s\", \"%s\")",
                      "R U R' U R U2 R'", "Sune (uneditable)"),
    };

    /**
     * The try/catch block is invoked if the table does not exist or the query
     * is invalid
     *
     * @param query
     *         the SQLite query to be performed on 'algorithm' table
     * @return an array of Algorithms representing the result of the specified
     *         query
     * @throws SQLException
     *         if the query is invalid
     * @throws ClassNotFoundException
     *         if SQLite classes are missing
     */
    public static Algorithm[] executeQuery(String query) throws SQLException, ClassNotFoundException {
        Algorithm[] algorithms = null;
```

```
try {
    algorithms = executeSafeQuery(query);
} catch (SQLException e) {
    initTable();
    algorithms = executeSafeQuery(query);
}

return algorithms;
}

/**
 * This method will only be called once the table exists
 *
 * @param query
 *          the SQLite query to be performed on 'algorithm' table
 * @return an array of Algorithms representing the result of the specified
 *         query
 * @throws ClassNotFoundException
 *          if SQLite classes are missing
 * @throws SQLException
 *          if the table does not exist or the query is invalid
 */
private static Algorithm[] executeSafeQuery(String query) throws ClassNotFoundException, SQLException {
    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement;
    ResultSet rs;
    Algorithm[] algorithms;
    int numRecords = 0;

    connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
    statement = connection.createStatement();
    statement.setQueryTimeout(30); // set timeout to 30 sec.

    rs = statement.executeQuery(query);

    while (rs.next())
        ++numRecords;

    rs.close();
    rs = statement.executeQuery(query);

    algorithms = new Algorithm[numRecords];
```

```
for (int i = 0; i < numRecords; ++i) {
    rs.next();
    algorithms[i] = new Algorithm(rs.getInt("algorithmID"), rs.getString("moveSequence"), ""
        + rs.getString("comment"));
}

try {
    if (connection != null)
        connection.close();
} catch (SQLException e) {
}

return algorithms;
}

/**
 * Executes the specified update on the 'algorithm' table
 *
 * @param update
 *         the update to be performed on the table
 * @throws ClassNotFoundException
 *         if SQLite classes are missing
 * @throws SQLException
 *         if the query is invalid
 */
public static void executeUpdate(String update) throws ClassNotFoundException, SQLException {
    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement = null;

    connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
    statement = connection.createStatement();
    statement.setQueryTimeout(30); // set timeout to 30 sec.

    statement.executeUpdate(update);

    try {
        if (connection != null)
            connection.close();
    } catch (SQLException e2) {
    }
}
```

```
/**  
 * Initialises the table in the database. This method will be called if the  
 * executeQuery(...) method cannot find the table in the database.  
 *  
 * @throws ClassNotFoundException  
 *         if SQLite classes are missing  
 */  
private static void initTable() throws ClassNotFoundException {  
    Class.forName("org.sqlite.JDBC");  
    Connection connection = null;  
    Statement statement = null;  
  
    try {  
        connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");  
        statement = connection.createStatement();  
        statement.setQueryTimeout(30); // set timeout to 30 sec.  
  
        statement.executeUpdate("DROP TABLE IF EXISTS algorithm");  
        statement.executeUpdate("CREATE TABLE algorithm(" + "algorithmID INTEGER PRIMARY KEY AUTOINCREMENT,"  
            + "moveSequence TEXT," + "comment TEXT" + ");");  
        for (int i = 0; i < PRESET_ALGORITHM_UPDATES.length; ++i)  
            statement.execute(PRESET_ALGORITHM_UPDATES[i]);  
  
    } catch (SQLException e) {  
        try {  
            if (connection != null)  
                connection.close();  
        } catch (SQLException e2) {}  
    }  
}  
  
/**  
 * Resets the IDs in the 'algorithm' table so that they are continuous, e.g.  
 * if the IDs were 1, 2, 5, 6, 7, 12, 13 then they would be reset to 1, 2,  
 * 3, 4, 5, 6, 7  
 *  
 * @throws ClassNotFoundException  
 *         if SQLite classes are missing  
 * @throws SQLException  
 *         if the table does not exist etc.  
 */  
public static void resetIDs() throws ClassNotFoundException, SQLException {  
    Class.forName("org.sqlite.JDBC");
```

```
// This is used to establish a connection with the database.  
Connection connection = null;  
// This is used to perform updates and queries on the database  
Statement statement = null;  
// This stores the records returned from a query.  
ResultSet rs;  
  
try {  
    connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");  
    statement = connection.createStatement();  
    statement.setQueryTimeout(30); // set timeout to 30 sec.  
  
    rs = statement.executeQuery("SELECT * FROM algorithm");  
  
    statement = connection.createStatement();  
    statement.executeUpdate("DROP TABLE IF EXISTS algorithmCopy");  
    statement.executeUpdate("CREATE TABLE algorithmCopy(" + "algorithmID INTEGER PRIMARY KEY AUTOINCREMENT,"  
        + "moveSequence TEXT," + "comment TEXT" + ");");  
  
    while (rs.next()) {  
        String moveSequence = rs.getString("moveSequence"), comment = rs.getString("comment");  
        statement.executeUpdate(String.format(  
            "INSERT INTO algorithmCopy(moveSequence, comment) VALUES ('%s', '%s');", moveSequence,  
            comment));  
    }  
  
    statement.executeUpdate("DROP TABLE algorithm");  
    statement.executeUpdate("ALTER TABLE algorithmCopy RENAME TO algorithm");  
} catch (SQLException e) {  
    System.out.println(e.getMessage());  
  
    if (connection != null)  
        connection.close();  
  
    throw new SQLException();  
}  
}
```

## Class AlgorithmDatabasePopUp

```
package jCube;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.SQLException;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;

/**
 * @author Kelsey McKenna
 */
public class AlgorithmDatabasePopUp extends JFrame {
    /**
     * Default serialVersionUID
     */
    private static final long serialVersionUID = 1L;
    /**
     * This indicates the initial width of the 'Algorithm Table' window
     */
    private static final int WIDTH = 700;

    /**
     * The list of algorithms is stored inside this panel
     */
    private JPanel algorithmListPanel;
    /**
     * algorithmTable is placed inside this so that when the size of the table
     * exceeds the size of the window, the rest of the table can be viewed by
     * scrolling
    
```

```
/*
private JScrollPane tableContainer;
/**
 * This stores the contents of the table
 */
private final JTable algorithmTable;
/**
 * By setting the model of the algorithmTable to 'model', certain cells of
 * the table can be made uneditable, and the columns can be given names
 */
private final DefaultTableModel model;
/**
 * The buttons are placed in this panel so that they are grouped together
 */
private JPanel buttonPanel;

/**
 * Clicking this button opens an input dialog so that a new algorithm can be
 * entered
 */
private JButton addAlgorithmButton;
/**
 * Clicking this button applies the reverse of the selected algorithm to the
 * virtual cube in the main window
 */
private JButton applyReverseAlgorithmButton;
/**
 * Clicking this button performs the algorithm in real-time on the virtual
 * cube in the main window
 */
private JButton viewExecutionButton;
/**
 * Clicking this button deletes the selected rows from the table
 */
private JButton deleteAlgorithmButton;

/**
 * Constructor - sets up the window
 */
public AlgorithmDatabasePopUp() {
    super("Algorithm Table");

    setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
    setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

```
setLayout(new BorderLayout());
setPreferredSize(new Dimension(WIDTH, 400));
setVisible(false);

algorithmListPanel = new JPanel();
algorithmListPanel.setOpaque(true);

final String[] columnNames = { "ID", "Algorithm", "Comment", };

model = new DefaultTableModel() {
    private static final long serialVersionUID = 1L;

    public int getColumnCount() {
        return 3;
    }

    public String getColumnName(int col) {
        return columnNames[col];
    }

    public boolean isCellEditable(int row, int col) {
        return ((col != 0) && (row >= AlgorithmDatabaseConnection.PRESET_ALGORITHM_UPDATES.length));
    }
};

model.addTableModelListener(new TableModelListener() {
    @Override
    public void tableChanged(TableModelEvent e) {
        Main.resizeColumnWidths(algorithmTable);
        int row = e.getFirstRow();

        TableModel model = (TableModel) e.getSource();
        try {
            AlgorithmDatabaseConnection.executeUpdate(String.format("UPDATE algorithm "
                + "SET moveSequence = \"%s\", " + "comment = \"%s\" " + "WHERE algorithmID = %d", ""
                + model.getValueAt(row, 1), "" + model.getValueAt(row, 2),
                Integer.valueOf("") + model.getValueAt(row, 0)));
        } catch (Exception exc) {
        }
    }
});

algorithmTable = new JTable();
algorithmTable.setModel(model);
algorithmTable.setColumnSelectionAllowed(false);
```

```
algorithmTable.setPreferredScrollableViewportSize(new Dimension(WIDTH, 70));
algorithmTable.setFillsViewportHeight(true);
algorithmTable.setAutoscrolls(true);
algorithmTable.getTableHeader().setReorderingAllowed(false);
algorithmTable.setFont(new Font("Arial", 0, 15));
algorithmTable.setRowHeight(20);

buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(1, 3));
buttonPanel.setPreferredSize(new Dimension(WIDTH, 40));
buttonPanel.setSize(WIDTH, 50);

addAlgorithmButton = new JButton("Add Algorithm");
addAlgorithmButton.setFocusable(false);
addAlgorithmButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            addRow();
        } catch (SQLException exc) {
            JOptionPane
                .showMessageDialog(null, "Could not access database", "Error", JOptionPane.ERROR_MESSAGE);
        } catch (ClassNotFoundException exc) {
            JOptionPane
                .showMessageDialog(null, "Could not access database", "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
});;

viewExecutionButton = new JButton("View Execution");
viewExecutionButton.setFocusable(false);
viewExecutionButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int selectedRow = algorithmTable.getSelectedRow();
        if (selectedRow != -1) {
            String algorithm = "" + model.getValueAt(selectedRow, 1);
            Main.performRealTimeSolving(SolveMaster.getReverseStringMoves(algorithm), algorithm);
        }
    }
});;

applyReverseAlgorithmButton = new JButton("Apply Reverse");
applyReverseAlgorithmButton.setFocusable(false);
applyReverseAlgorithmButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
```

```
int selectedRow = algorithmTable.getSelectedRow();
if (selectedRow != -1) {
    // Main.performRealTimeSolving("", 
    // SolveMaster.getReverseStringMoves("") + 
    // model.getValueAt(algorithmTable.getSelectedRow(), 1)));
    Main.handleScramble(SolveMaster.getReverseStringMoves("") + model.getValueAt(selectedRow, 1)));
}
});

deleteAlgorithmButton = new JButton("Delete Algorithm");
deleteAlgorithmButton.setFocusable(false);
deleteAlgorithmButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        try {
            Object[] options = { "Yes", "No" };
            int choice = -1;

            if (algorithmTable.getSelectedRows().length > 5) {
                choice = JOptionPane.showOptionDialog(null, "Are you sure you want to delete?", "Warning",
                    JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[1]);
            } else
                choice = 0;

            if (choice == 0)
                deleteRow();
        } catch (ClassNotFoundException | SQLException e) {
            JOptionPane.showMessageDialog(algorithmTable, "Unable to delete record from database", "Error",
                JOptionPane.ERROR_MESSAGE);
        }
    }
});

buttonPanel.add(addAlgorithmButton);
buttonPanel.add(applyReverseAlgorithmButton);
buttonPanel.add(viewExecutionButton);
buttonPanel.add(deleteAlgorithmButton);

tableContainer = new JScrollPane(algorithmTable);
tableContainer.setViewportView(algorithmTable, null);

Main.resizeColumnWidths(algorithmTable);
populateCellsWithDatabaseData();
```

```
    add(tableContainer, null);
    add(buttonPanel, BorderLayout.SOUTH);
    pack();
}

/**
 * Retrieves the data from the 'algorithm' table and adds it to the table in
 * the window
 */
private void populateCellsWithDatabaseData() {
    int rowCount = model.getRowCount();

    for (int i = 0; i < rowCount; ++i) { // Remove all rows from table
        model.removeRow(0);
    }

    Algorithm[] algorithms = new Algorithm[0];

    try {
        algorithms = AlgorithmDatabaseConnection.executeQuery("SELECT * " + "FROM algorithm;");
    } catch (SQLException e) {
        JOptionPane.showMessageDialog(this, "Could not load algorithms", "Error", JOptionPane.ERROR_MESSAGE);
    } catch (ClassNotFoundException e) {
        JOptionPane.showMessageDialog(this, "Could not load algorithms", "Error", JOptionPane.ERROR_MESSAGE);
    }

    if (algorithms != null) {
        for (int i = 0; i < algorithms.length; ++i) {
            model.addRow(new Object[] { "" + algorithms[i].getAlgorithmID(), algorithms[i].getMoveSequence(),
                algorithms[i].getComment() });
        }
    }

    Main.resizeColumnWidths(algorithmTable);
}

/**
 * Adds a row to the table and to the database
 *
 * @throws ClassNotFoundException
 *         if SQLite classes are missing
 * @throws SQLException
 *         if the table does not exist etc.
*/
```

```
/*
private void addRow() throws ClassNotFoundException, SQLException {
    AlgorithmDatabaseConnection.executeUpdate("INSERT INTO algorithm(moveSequence, comment) VALUES ('\\\"', '\\\"')");
    model.addRow(new Object[] { model.getRowCount() + 1, "", "" });

    Main.resizeColumnWidths(algorithmTable);
    algorithmTable.setRowSelectionInterval(algorithmTable.getRowCount() - 1, algorithmTable.getRowCount() - 1);
}

/**
 * Deletes the selected rows from the table, and deletes the corresponding
 * records from the database
 *
 * @throws ClassNotFoundException
 *         if SQLite classes are missing
 * @throws SQLException
 *         if the table does not exist or the updates fail
 */
private void deleteRow() throws ClassNotFoundException, SQLException {
    int[] selectedIndices = algorithmTable.getSelectedRows();

    if (selectedIndices.length == 0)
        return;

    for (int i = 0; i < selectedIndices.length; ++i) {
        if (selectedIndices[i] >= AlgorithmDatabaseConnection.PRESET_ALGORITHM_UPDATES.length)
            AlgorithmDatabaseConnection.executeUpdate("DELETE FROM algorithm WHERE algorithmID = "
                + ("'" + model.getValueAt(selectedIndices[i], 0)) + ";");
    }

    for (int i = selectedIndices.length - 1; i >= 0; --i) {
        if (selectedIndices[i] >= AlgorithmDatabaseConnection.PRESET_ALGORITHM_UPDATES.length)
            model.removeRow(selectedIndices[i]);
    }

    for (int i = selectedIndices[0]; i < model.getRowCount(); ++i)
        model.setValueAt("'" + (i + 1), i, 0);

    AlgorithmDatabaseConnection.resetIDs();

    try {
        algorithmTable.setRowSelectionInterval(selectedIndices[selectedIndices.length - 1],
            selectedIndices[selectedIndices.length - 1]);
    } catch (Exception e) {
```

```
        algorithmTable.setRowSelectionInterval(Math.max(algorithmTable.getRowCount() - 1, 0),
                                                Math.max(algorithmTable.getRowCount() - 1, 0));
    }
}
```

## Class ColorSelection

```
package jCube;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingConstants;

/**
 * @author Kelsey McKenna
 */
public class ColorSelection extends JFrame implements MouseListener {
    /**
     * Default serialVersionUID
     */
    private static final long serialVersionUID = 1L;

    /**
     * The contents of the window are placed in this JPanel
     */
    private JPanel contentPane;

    /**
     * This label is displayed at the bottom of the window with a black
     * background and shows the text 'COLOR'. The foreground of the text
     * indicates the selected colour
     */
    private JLabel selectedColorLabel;

    /**
     * This is displayed as a white rectangle on the screen. When clicked, the
     * selected colour is changed to white.
     */
    private JTextField whiteButton;
    /**
     * This is displayed as a yellow rectangle on the screen. When clicked, the
```

```
* selected colour is changed to yellow.  
*/  
private JTextField yellowButton;  
/**  
 * This is displayed as a red rectangle on the screen. When clicked, the  
 * selected colour is changed to red.  
 */  
private JTextField redButton;  
/**  
 * This is displayed as an orange rectangle on the screen. When clicked, the  
 * selected colour is changed to orange.  
 */  
private JTextField orangeButton;  
/**  
 * This is displayed as a green rectangle on the screen. When clicked, the  
 * selected colour is changed to green.  
 */  
private JTextField greenButton;  
/**  
 * This is displayed as a blue rectangle on the screen. When clicked, the  
 * selected colour is changed to blue.  
 */  
private JTextField blueButton;  
  
/**  
 * This stores the selected color  
 */  
private Color selectedColor = Color.white;  
  
/**  
 * This indicates the width of the window  
 */  
private static final int WIDTH = 215;  
/**  
 * This indicates the height of the window  
 */  
private static final int HEIGHT = 270;  
/**  
 * This indicates the height of each of the rectangles in the window  
 */  
private static final int BUTTON_HEIGHT = HEIGHT / 7 - 3;  
  
/**  
 * Constructor - sets up the window
```

```
/*
public ColorSelection() {
    super("Color Selection");

    int y = 0;

    contentPane = new JPanel();
    contentPane.setLayout(null);

    setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
    setContentPane(contentPane);
    setPreferredSize(new Dimension(WIDTH, HEIGHT));
    setSize(new Dimension(WIDTH, HEIGHT));
    setResizable(false);
    setFocusable(false);
    setAlwaysOnTop(true);

    whiteButton = new JTextField();
    whiteButton.setEditable(false);
    whiteButton.addMouseListener(this);

    yellowButton = new JTextField();
    yellowButton.setEditable(false);
    yellowButton.addMouseListener(this);

    redButton = new JTextField();
    redButton.setEditable(false);
    redButton.addMouseListener(this);

    orangeButton = new JTextField();
    orangeButton.setEditable(false);
    orangeButton.addMouseListener(this);

    greenButton = new JTextField();
    greenButton.setEditable(false);
    greenButton.addMouseListener(this);

    blueButton = new JTextField();
    blueButton.setEditable(false);
    blueButton.addMouseListener(this);

    selectedColorLabel = new JLabel("COLOR", SwingConstants.CENTER);
    selectedColorLabel.setForeground(Color.BLACK);
    selectedColorLabel.setBackground(selectedColor.brighter());
```

```
selectedColorLabel.setOpaque(true);
selectedColorLabel.setFont(new Font("Lucida Sans", 0, 25));

whiteButton.setBackground(Color.white);
yellowButton.setBackground(Color.yellow);
redButton.setBackground(Color.red);
orangeButton.setBackground(Cubie.orange);
greenButton.setBackground(Color.green);
blueButton.setBackground(Color.blue);

whiteButton.setSize(WIDTH, BUTTON_HEIGHT);
whiteButton.setLocation(0, y);
y += BUTTON_HEIGHT;

yellowButton.setSize(WIDTH, BUTTON_HEIGHT);
yellowButton.setLocation(0, y);
y += BUTTON_HEIGHT;

redButton.setSize(WIDTH, BUTTON_HEIGHT);
redButton.setLocation(0, y);
y += BUTTON_HEIGHT;

orangeButton.setSize(WIDTH, BUTTON_HEIGHT);
orangeButton.setLocation(0, y);
y += BUTTON_HEIGHT;

greenButton.setSize(WIDTH, BUTTON_HEIGHT);
greenButton.setLocation(0, y);
y += BUTTON_HEIGHT;

blueButton.setSize(WIDTH, BUTTON_HEIGHT);
blueButton.setLocation(0, y);
y += BUTTON_HEIGHT;

selectedColorLabel.setSize(WIDTH, BUTTON_HEIGHT);
selectedColorLabel.setLocation(0, y);

contentPane.add(whiteButton);
contentPane.add(yellowButton);
contentPane.add(redButton);
contentPane.add(orangeButton);
contentPane.add(greenButton);
contentPane.add(blueButton);
contentPane.add(selectedColorLabel);
```

```
    setVisible(false);
}

/**
 * @return the selected colour to be painted on the cube in the main window
 */
public Color getSelectedColor() {
    return selectedColor;
}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.MouseListener#mouseClicked(java.awt.event.MouseEvent)
 */
@Override
public void mouseClicked(MouseEvent arg0) {
    Object source = arg0.getSource();

    selectedColorLabel.setForeground(Color.BLACK);

    if (source == whiteButton)
        selectedColor = (Color.white);
    else if (source == yellowButton)
        selectedColor = (Color.yellow);
    else if (source == redButton)
        selectedColor = (Color.red);
    else if (source == orangeButton)
        selectedColor = (Cubie.orange);
    else if (source == greenButton)
        selectedColor = (Color.green);
    else if (source == blueButton) {
        selectedColorLabel.setForeground(Color.white);
        selectedColor = (Color.blue);
    }

    selectedColorLabel.setBackground(selectedColor.brighter());
    Main.requestCubePanelFocus();
}

/**
 * (non-Javadoc)
 *
```

```
* @see java.awt.event.MouseListener#mouseEntered(java.awt.event.MouseEvent)
*/
@Override
public void mouseEntered(MouseEvent arg0) {

}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.MouseListener#mouseExited(java.awt.event.MouseEvent)
 */
@Override
public void mouseExited(MouseEvent arg0) {

}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.MouseListener#mousePressed(java.awt.event.MouseEvent)
 */
@Override
public void mousePressed(MouseEvent arg0) {

}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.MouseListener#mouseReleased(java.awt.event.MouseEvent)
 */
@Override
public void mouseReleased(MouseEvent arg0) {

}

}
```

## Class Competition

```
package jCube;

import java.text.SimpleDateFormat;

/**
 * @author Kelsey McKenna
 */
public class Competition {

    /**
     * The ID of the competition
     */
    private int competitionID;
    /**
     * The date of the competition
     */
    private String date;

    /**
     * Constructor to initialise fields
     *
     * @param competitionID
     *         the ID of the competition
     * @param date
     *         the date of the competition
     */
    public Competition(int competitionID, String date) {
        this.setID(competitionID);
        this.setDate(date);
    }

    /**
     * @return the ID of the competition
     */
    public int getID() {
        return competitionID;
    }

    /**
     * @param competitionID
     *         the ID of the competition
     */
}
```

```
public void setID(int competitionID) {
    this.competitionID = competitionID;
}

/**
 * @return the date of the competition
 */
public String getDate() {
    return date;
}

/**
 * @param date
 *          the date of the competition
 */
public void setDate(String date) {
    this.date = date;
}

/**
 * Returns a value indicating whether the argument is a valid date for a
 * competition
 *
 * @param dateString
 *          the date to be analysed
 * @return true if the argument is valid and is in the format dd/MM/yyyy;
 *         false otherwise
 */
public static boolean isValidDate(String dateString) {
    try {
        /*
         * This will throw a ParseException if dateString is not in the
         * correct format. No assignment statement is needed; this statement
         * just confirms that dateString is in the correct format to
         * proceed.
        */
        new SimpleDateFormat("dd/MM/yyyy").parse(dateString);

        // Stores an integer representation of the day in the month.
        int day;
        // Stores an integer representation of the month in the year.
        int month;
        // Stores an integer representation of the year.
        int year;
    }
}
```

```
day = Integer.valueOf(dateString.substring(0, dateString.indexOf("/")));
dateString = dateString.substring(dateString.indexOf("/") + 1);

month = Integer.valueOf(dateString.substring(0, dateString.indexOf("/")));
year = Integer.valueOf(dateString.substring(dateString.indexOf("/") + 1));

/*
 * This means the method will return false if a month greater than
 * '12' is indicated. It will also return false if the day indicated
 * is greater than the number of days in the specified month.
 */
if ((month > 12) || (getNumDaysInMonth(month, year) < day))
    return false;
else
    return true;
} catch (Exception e) {
    return false;
}
}

/**
 * @param month
 *          the month of the date in question
 * @param year
 *          the year of the date in question
 * @return the number of days in the month
 */
private static int getNumDaysInMonth(int month, int year) {
    switch (month) {
        case 2:
            /*
             * This is a leap year checker. A year is a leap year if it is
             * divisible by four and not divisible by 100 unless it is also
             * divisible by 400.
             */
            if (((year % 4 == 0) && ((year % 100 == 0) ? (year % 400 == 0) : true))) {
                return 29;
            } else {
                return 28;
            }
        case 4:
        case 6:
        case 7:
```

```
    case 11:  
        return 30;  
    default:  
        return 31;  
    }  
}  
}
```

## Class CompetitionDatabaseConnection

```
package jCube;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * @author Kelsey McKenna
 */
public class CompetitionDatabaseConnection {

    /**
     * The try/catch block is invoked if the table does not exist or the query
     * is invalid
     *
     * @param query
     *         the SQLite query to be performed on 'competition' table
     * @return an array of Competitions representing the result of the specified
     *         query
     * @throws SQLException
     *         if the query is invalid
     * @throws ClassNotFoundException
     *         if SQLite classes are missing
     */
    public static Competition[] executeQuery(String query) throws SQLException, ClassNotFoundException {
        Competition[] competitions = null;

        try {
            competitions = executeSafeQuery(query);
        } catch (SQLException e) {
            initTable();
            competitions = executeSafeQuery(query);
        }

        return competitions;
    }

    /**
     * @param query
     *         the SQLite query to be performed on 'competition' table

```

```
* @return an array of Competitions representing the result of the specified
*         query
* @throws ClassNotFoundException
*         if SQLite classes are missing
* @throws SQLException
*         if the table does not exist or the query is invalid
*/
private static Competition[] executeSafeQuery(String query) throws ClassNotFoundException, SQLException {
    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement;
    ResultSet rs;
    Competition[] competitions;
    int numRecords = 0;

    connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
    statement = connection.createStatement();
    statement.setQueryTimeout(30); // set timeout to 30 sec.

    rs = statement.executeQuery(query);

    while (rs.next())
        ++numRecords;

    rs.close();
    rs = statement.executeQuery(query);

    competitions = new Competition[numRecords];

    for (int i = 0; i < numRecords; ++i) {
        rs.next();
        competitions[i] = new Competition(rs.getInt("competitionID"), rs.getString("competitionDate"));
    }

    try {
        if (connection != null)
            connection.close();
    } catch (SQLException e) {
    }

    return competitions;
}

/**
```

```
* Executes the specified update on the 'competition' table
*
* @param update
*         the update to be performed on the table
* @throws ClassNotFoundException
*         if SQLite classes are missing
* @throws SQLException
*         if the query is invalid
*/
public static void executeUpdate(String update) throws ClassNotFoundException, SQLException {
    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement = null;

    connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
    statement = connection.createStatement();
    statement.setQueryTimeout(30); // set timeout to 30 sec.

    statement.executeUpdate(update);

    try {
        if (connection != null)
            connection.close();
    } catch (SQLException e2) {
    }
}

/**
 * Initialises the table in the database. This method will be called if the
 * executeQuery(...) method cannot find the table in the database.
 *
* @throws ClassNotFoundException
*         if SQLite classes are missing
*/
private static void initTable() throws ClassNotFoundException {
    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement = null;

    try {
        connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
        statement = connection.createStatement();
        statement.setQueryTimeout(30); // set timeout to 30 sec.
        statement.executeUpdate("DROP TABLE IF EXISTS competition");
```

```
statement.executeUpdate("CREATE TABLE competition(" + "competitionID INTEGER PRIMARY KEY AUTOINCREMENT,"  
+ "competitionDate TEXT" + ");");  
  
} catch (SQLException e) {  
    try {  
        if (connection != null)  
            connection.close();  
    } catch (SQLException e2) {  
    }  
}  
}  
}
```

## Class CompetitionDatabasePopUp

```
package jCube;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.sql.SQLException;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.DefaultTableModel;

/**
 * @author Kelsey McKenna
 */
public class CompetitionDatabasePopUp extends JFrame {
    /**
     * Default serialVersionUID
     */
    private static final long serialVersionUID = 1L;
    /**
     * The initial width of the window
     */
    private static final int WIDTH = 700;

    /**
     * The list of competitions is stored inside this panel
     */
    private JPanel competitionListPanel;
    /**
     * competitionTable is placed inside this so that when the size of the table
     * exceeds the size of the window, the rest of the table can be viewed by

```

```
* scrolling
*/
private JScrollPane tableContainer;
/**
 * Stores the contents of the table
 */
private final JTable competitionTable;
/**
 * By setting the model of the algorithmTable to 'model', certain cells of
 * the table can be made uneditable, and the columns can be given names
 */
private final DefaultTableModel model;
/**
 * The buttons are placed in this panel so that they are grouped together
 */
private JPanel buttonPanel;

/**
 * Clicking this button adds a row to the table
 */
private JButton addCompetitionButton;
/**
 * Clicking this button opens an input dialog so that the date of the
 * competition can be edited
 */
private JButton editCompetitionButton;
/**
 * Clicking this button deletes the selected competitions from the table and
 * from the database
 */
private JButton deleteCompetitionButton;
/**
 * Clicking this button opens the 'Member-Competition Table' window and
 * displays the rankings for the selected competition
 */
private JButton viewRankingsButton;

/**
 * This is the window that opens when the viewRankingsButton is clicked
 */
private MemberCompetitionDatabasePopUp memberCompetitionDatabasePopUp;

/**
 * Constructor - sets up the window
*/
```

```
/*
public CompetitionDatabasePopUp() {
    super("Competition Table");

    setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
    setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    setLayout(new BorderLayout());
    setPreferredSize(new Dimension(WIDTH, 400));
    setVisible(false);

    memberCompetitionDatabasePopUp = new MemberCompetitionDatabasePopUp();

    competitionListPanel = new JPanel();
    competitionListPanel.setOpaque(true);

    final String[] columnNames = { "ID", "Date" };

    model = new DefaultTableModel() {
        private static final long serialVersionUID = 1L;

        public int getColumnCount() {
            return 2;
        }

        public String getColumnName(int col) {
            return columnNames[col];
        }

        public boolean isCellEditable(int row, int col) {
            return false;
        }
    };
    model.addTableModelListener(new TableModelListener() {
        @Override
        public void tableChanged(TableModelEvent e) {
            Main.resizeColumnWidths(competitionTable);
            /*
            * int row = e.getFirstRow();
            *
            * TableModel model = (TableModel)e.getSource(); try {
            * CompetitionDatabaseConnection.executeUpdate( String.format(
            * "UPDATE competition " + "SET competitionDate = \"%s\", " +
            * "WHERE competitionID = %d", "" + model.getValueAt(row, 1), ""
            * + model.getValueAt(row, 2)) ); } catch (Exception exc) { }
            */
        }
    });
}
```

```
        */
    });

competitionTable = new JTable();
competitionTable.setModel(model);
competitionTable.setColumnSelectionAllowed(false);
competitionTable.setPreferredScrollableViewportSize(new Dimension(WIDTH, 70));
competitionTable.setFillsViewportHeight(true);
competitionTable.setAutoScrolls(true);
competitionTable.getTableHeader().setReorderingAllowed(false);
competitionTable.setFont(new Font("Arial", 0, 15));
competitionTable.setRowHeight(20);
competitionTable.addMouseListener(new MouseListener() {

    @Override
    public void mouseClicked(MouseEvent arg0) {
        if (arg0.getClickCount() == 2)
            viewRankingsButtonFunction();
    }

    @Override
    public void mouseEntered(MouseEvent arg0) {
    }

    @Override
    public void mouseExited(MouseEvent arg0) {
    }

    @Override
    public void mousePressed(MouseEvent arg0) {
    }

    @Override
    public void mouseReleased(MouseEvent arg0) {
    }
});

buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(1, 3));
buttonPanel.setPreferredSize(new Dimension(WIDTH, 40));
buttonPanel.setSize(WIDTH, 50);
```

```
addCompetitionButton = new JButton("Add Competition");
addCompetitionButton.setFocusable(false);
addCompetitionButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            addRow();
        } catch (SQLException exc) {
            exc.printStackTrace();
            JOptionPane
                .showMessageDialog(null, "Could not access database", "Error", JOptionPane.ERROR_MESSAGE);
        } catch (ClassNotFoundException exc) {
            JOptionPane
                .showMessageDialog(null, "Could not access database", "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
});

editCompetitionButton = new JButton("Edit Competition Date");
editCompetitionButton.setFocusable(false);
editCompetitionButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        editButtonFunction();
    }
});

deleteCompetitionButton = new JButton("Delete Competition");
deleteCompetitionButton.setFocusable(false);
deleteCompetitionButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        try {
            Object[] options = { "Yes", "No" };
            int choice = -1;

            if (competitionTable.getSelectedRows().length == 0)
                return;

            choice = JOptionPane.showOptionDialog(null, "Are you sure you want to delete?", "Warning",
                JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[1]);

            if (choice == 0)
                deleteRow();
        } catch (ClassNotFoundException | SQLException e) {
            JOptionPane.showMessageDialog(competitionTable, "Unable to delete record from database", "Error",

```

```
        JOptionPane.ERROR_MESSAGE);
    }
});
});

viewRankingsButton = new JButton("View Rankings");
viewRankingsButton.setFocusable(false);
viewRankingsButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        viewRankingsButtonFunction();
    }
});

buttonPanel.add(addCompetitionButton);
buttonPanel.add(editCompetitionButton);
buttonPanel.add(viewRankingsButton);
buttonPanel.add(deleteCompetitionButton);

tableContainer = new JScrollPane(competitionTable);
tableContainer.setViewportView(null);

Main.resizeColumnWidths(competitionTable);
populateCellsWithDatabaseData();

add(tableContainer, null);
add(buttonPanel, BorderLayout.SOUTH);
pack();
}

/**
 * Sets up the memberCompetitionDatabasePopUp window so that it is visible
 * and the cells are populated with data relating to the selecting row in
 * the table
 */
private void viewRankingsButtonFunction() {
    if ((competitionTable.getSelectedRow() > -1)) {
        memberCompetitionDatabasePopUp.setCurrentCompetitionID(Integer.valueOf("")
            + competitionTable.getValueAt(competitionTable.getSelectedRow(), 0)));
        memberCompetitionDatabasePopUp.populateCellsWithDatabaseData();
        memberCompetitionDatabasePopUp.setVisible(true);
    }
}
```

```
/**  
 * Performs the operations required to allow the selected row to be edited  
 * and validation to be performed on the data entered  
 */  
private void editButtonFunction() {  
    // Stores the string entered by the user.  
    String date;  
    // Stores the index of the row in the table selected by the user.  
    int row = competitionTable.getSelectedRow();  
  
    if (row != -1) {  
        // Stores the date-string shown in the selected row so that if  
        // editing, the existing date can be shown in the input window.  
        String originalDate = "" + competitionTable.getValueAt(row, 1);  
        date = JOptionPane.showInputDialog(null, "Enter Date of Competition",  
            (originalDate.equals("")) ? "dd/MM/YYYY" : originalDate);  
  
        if (date == null)  
            return;  
  
        date += " ";  
        date = date.trim();  
  
        if ((date.equals("")) || (!Competition.isValidDate(date))) {  
            JOptionPane.showMessageDialog(this, "Invalid date", "Error", JOptionPane.ERROR_MESSAGE);  
        } else {  
            try {  
                CompetitionDatabaseConnection.executeUpdate(String.format("UPDATE competition "  
                    + "SET competitionDate = \'%s\' " + "WHERE competitionID = %d", date,  
                    Integer.valueOf("") + model.getValueAt(competitionTable.getSelectedRow(), 0)));  
                competitionTable.setValueAt(date, row, 1);  
            } catch (Exception exc) {  
                exc.printStackTrace();  
            }  
        }  
    }  
}  
  
/**  
 * Populates the cells of the table with the data from the database  
 */  
private void populateCellsWithDatabaseData() {  
    int rowCount = model.getRowCount();
```

```
for (int i = 0; i < rowCount; ++i) { // Remove all rows from table
    model.removeRow(0);
}

Competition[] competitions = new Competition[0];

try {
    competitions = CompetitionDatabaseConnection.executeQuery("SELECT * " + "FROM competition;");
} catch (SQLException e) {
    JOptionPane.showMessageDialog(this, "Could not load algorithms", "Error", JOptionPane.ERROR_MESSAGE);
} catch (ClassNotFoundException e) {
    JOptionPane.showMessageDialog(this, "Could not load algorithms", "Error", JOptionPane.ERROR_MESSAGE);
}

if (competitions != null) {
    for (int i = 0; i < competitions.length; ++i) {
        model.addRow(new Object[] { "" + competitions[i].getID(), competitions[i].getDate() });
    }
}

Main.resizeColumnWidths(competitionTable);
}

/**
 * Adds a row to the table and a blank record to the 'competition' table in
 * the database
 *
 * @throws ClassNotFoundException
 *         if SQLite classes are missing
 * @throws SQLException
 *         if the table does not exist etc.
 */
private void addRow() throws ClassNotFoundException, SQLException {
    CompetitionDatabaseConnection.executeUpdate("INSERT INTO competition(competitionDate) VALUES ('\"\"')");
    model.addRow(new Object[] {
        ""
        + CompetitionDatabaseConnection
            .executeQuery("SELECT * FROM competition ORDER BY competitionID DESC LIMIT 1") [0]
            .getID(), "", "" });

    Main.resizeColumnWidths(competitionTable);
    competitionTable
        .setRowSelectionInterval(competitionTable.getRowCount() - 1, competitionTable.getRowCount() - 1);
}
```

```
}

/**
 * Deletes the selected rows from competitionTable
 *
 * @throws ClassNotFoundException
 *         if SQLite classes are missing
 * @throws SQLException
 *         if the table does not exist etc.
 */
private void deleteRow() throws ClassNotFoundException, SQLException {
    int[] selectedIndices = competitionTable.getSelectedRows();

    for (int i = 0; i < selectedIndices.length; ++i) {
        CompetitionDatabaseConnection.executeUpdate("DELETE FROM competition WHERE competitionID = "
            + ("'" + model.getValueAt(selectedIndices[i], 0)) + "'");
    }

    for (int i = selectedIndices.length - 1; i >= 0; --i) {
        model.removeRow(selectedIndices[i]);
    }

    try {
        competitionTable.setRowSelectionInterval(selectedIndices[selectedIndices.length - 1],
            selectedIndices[selectedIndices.length - 1]);
    } catch (Exception e) {
        if (competitionTable.getRowCount() > 0)
            competitionTable.setRowSelectionInterval(competitionTable.getRowCount() - 1,
                competitionTable.getRowCount() - 1);
    }
}
}
```

## Class Corner

```
package jCube;

import static java.awt.Color.blue;
import static java.awt.Color.green;
import static java.awt.Color.red;
import static java.awt.Color.white;
import static java.awt.Color.yellow;

import java.awt.Color;
import java.util.Arrays;

/**
 * @author Kelsey McKenna
 */
public class Corner extends Cubie {

    /**
     * Stores the initial permutation of the corners on a solved cube with white
     * and green centres on top and front respectively.
     */
    private static final Color[][] INITIAL_CORNERS = { { white, orange, blue }, { white, blue, red },
        { white, red, green }, { white, green, orange }, { yellow, red, blue }, { yellow, blue, orange },
        { yellow, orange, green }, { yellow, green, red } };

    /**
     * Constructor
     */
    public Corner() {
    }

    /**
     * @param stickers
     *          a three-element array containing the colors for the Corner
     */
    public Corner(Color[] stickers) {
        super(stickers);
    }

    /**
     * @param zero
     *          the first sticker on the Corner
     * @param one
     */
}
```

```
*           the second sticker on the Corner
 * @param two
 *           the third sticker on the Corner
 */
public Corner(Color zero, Color one, Color two) {
    setStickers(zero, one, two);
}

/**
 * Constructor - acts as a clone
 *
 * @param corner
 *           the corner to be copied
 */
public Corner(Corner corner) {
    super(corner.getStickers());
    setOrientation(corner.getOrientation());
}

/**
 * @return the INITIAL_CORNERS array
 */
public static Color[][] getAllInitialStickers() {
    return Arrays.copyOf(INITIAL_CORNERS, 8);
}

/**
 * Returns the stickers for the i-th corner
 *
 * @param index
 *           the index of the corner whose stickers are to be returned
 * @return the stickers of the specified corner
 */
public static Color[] getInitialStickers(int index) {
/*
 * for (int i = 0; i < 3; ++i) {
 * System.out.print(Cubie.getColorToWord(INITIAL_CORNERS[index][i])); }
 */

    return Arrays.copyOf(INITIAL_CORNERS[index], 3);
}

/**
 * Sets the three stickers of the Corner
```

```
*  
* @param zero  
*          the first sticker  
* @param one  
*          the second sticker  
* @param two  
*          the third sticker  
*/  
public void setStickers(Color zero, Color one, Color two) {  
    Color[] stickers = { zero, one, two };  
    super.setStickers(stickers);  
}  
  
/**  
 * Changes the saved orientation and stickers accordingly  
*  
* @param direction  
*          the direction in which to twist the corner. <br>  
*          <b>1</b> if clockwise; <br>  
*          <b>-1</b> if anti-clockwise  
*/  
public void twist(int direction) {  
    Color temp;  
    Color[] stickersCopy = super.getStickers();  
  
    temp = stickersCopy[0];  
  
    if (direction > 0) {  
        stickersCopy[0] = stickersCopy[2];  
        stickersCopy[2] = stickersCopy[1];  
        stickersCopy[1] = temp;  
    } else {  
        stickersCopy[0] = stickersCopy[1];  
        stickersCopy[1] = stickersCopy[2];  
        stickersCopy[2] = temp;  
    }  
  
    setOrientation(((getOrientation() + direction + 4) % 3) - 1);  
}  
}
```

## Class CornerSolver

```
package jCube;

import java.awt.Color;
import java.util.LinkedList;

/**
 * @author Kelsey McKenna
 */
public class CornerSolver extends SolveMaster {

    /**
     * @author Kelsey McKenna
     */
    private class SolveCandidate {
        /**
         * This stores the index of the corner on the cube
         */
        int index = -1000;
        /**
         * This stores the number of moves required to solve the corner
         */
        int score = -1000;
    }

    /**
     * This stores Corner objects representing the white-red-green,
     * white-green-orange, white-orange-blue, and white-blue-red corners.
     */
    private static Corner[] fLCorners = new Corner[4];
    /**
     * This stores the remaining first layer Corner objects that need to be
     * solved.
     */
    private SolveCandidate[] solveCandidates;

    /**
     * Constructor - sets the cube to be solved
     *
     * @param cube
     *          to be used to generate a solution
     */
    public CornerSolver(Cube cube) {
```

```
super(cube);

for (int i = 0; i < 4; ++i) {
    fLCorners[i] = new Corner(Corner.getInitialStickers(i));
}

/*
 * public void solveFirstLayerCorners() { Corner corner;
 * rotateToTop(Color.yellow); int numCornersToSolve = 4; int
 * solveCandidatesIndex; int cornerIndex; Color[] stickers;
 *
 * while (!firstLayerCornersSolved()) { solveCandidatesIndex = 0;
 * solveCandidates = new SolveCandidate[numCornersToSolve--];
 *
 * for (int i = 0; i < 8; ++i) { corner = cube.getCorner(i); if
 * (isFLCorner(corner) && (!pieceSolved(corner))) {
 * solveCandidates[solveCandidatesIndex] = new SolveCandidate();
 * solveCandidates[solveCandidatesIndex].index = i;
 * solveCandidates[solveCandidatesIndex].score = getScore(i);
 * ++solveCandidatesIndex; } }
 *
 * cornerIndex =
 * solveCandidates[getIndexOfMinScore(solveCandidatesIndex)].index; stickers
 * = cube.getCorner(cornerIndex).getStickers(); solutionExplanation +=
 * String.format("Corners - %s-%s-%s corner:\n",
 * Cubie.getColorWord(stickers[0]), Cubie.getColorWord(stickers[1]),
 * Cubie.getColorWord(stickers[2]));
 * solveCorner(solveCandidates[getIndexOfMinScore
 * (solveCandidatesIndex)].index); } }
 */

/**
 * Solves the corners in the first layer of the cube and records the
 * solution and explanation at the same time
 */
public void solveFirstLayerCorners() {
    // Stores the properties of the corner to be solved.
    Corner corner;
    rotateToTop(Color.yellow);
    // Stores the index of the unsolved corner being examined.
    int solveCandidatesIndex;
    // Stores the index of the corner to solve.
    int indexOfCornerToSolve;
```

```
// Stores the stickers of the corner so that a better explanation can be
// generated.
Color[] stickers;

solveCandidates = new SolveCandidate[4];
for (int i = 0; i < 4; ++i)
    // Initialise solveCandidates
    solveCandidates[i] = new SolveCandidate();

while (!firstLayerCornersSolved()) {
    solveCandidatesIndex = 0;

    for (int i = 0; i < 8; ++i) {
        corner = cube.getCorner(i);
        if (isFLCorner(corner) && (!isPieceSolved(corner))) {
            solveCandidates[solveCandidatesIndex].index = i;
            solveCandidates[solveCandidatesIndex].score = getScore(i);
            ++solveCandidatesIndex;
        }
    }

    indexOfCornerToSolve = solveCandidates[getIndexOfMinScore(solveCandidatesIndex)].index;

    stickers = cube.getCorner(indexOfCornerToSolve).getStickers();
    solutionExplanation += String.format("Corners - %s-%s-%s corner:\n",
                                         Cubie.getColorToWord(stickers[0]),
                                         Cubie.getColorToWord(stickers[1]),
                                         Cubie.getColorToWord(stickers[2]));

    solveCorner(indexOfCornerToSolve);
}

try {
    // Removes the last newline character
    solutionExplanation = solutionExplanation.substring(0, solutionExplanation.lastIndexOf("\n"));
} catch (IndexOutOfBoundsException e) {
}
}

/**
 * @param originalMoves
 *         the moves to be examined
 * @return <b>true</b> if the last move (after simplification and ignoring
 *         rotations) was U; <br>
 *         <b>false</b> otherwise
 */
```

```
private boolean lastMoveWasU(LinkedList<String> originalMoves) {
    // Stores a copy of originalMoves because 'moves' is later altered to
    // check for cancellations.
    LinkedList<String> moves = new LinkedList<>();
    int size = originalMoves.size();

    for (int i = 0; i < size; ++i)
        moves.add(originalMoves.get(i));

    simplifyMoves(moves, SolveMaster.CANCELLATIONS);

    if (moves.size() == 0)
        return false;

    else {
        for (int i = moves.size() - 1; i >= 0; --i) {
            if ("xyz".contains(moves.get(i).substring(0, 1)))
                continue;
            else if (!moves.get(i).contains("U"))
                return false;
            else
                return true;
        }
    }

    return false;
}

/**
 * Solves the specified corner, and records the solution and explanation at
 * the same time
 *
 * @param currentIndex
 *         the index of the corner to be solved
 */
public void solveCorner(int currentIndex) {
    // Stores a copy of the properties of the corner to be solved.
    Corner corner = cube.getCorner(currentIndex);
    // Stores the orientation (-1, 0, 1) of the corner.
    int orientation = corner.getOrientation();
    // Stores the index of the setup location for the corner.
    int overCornerIndex = getIndexOfDestination(corner);
    overCornerIndex += (overCornerIndex % 2 == 0) ? -3 : -5;
```

```
if (isPieceSolved(corner))
    return;

/*
 * The cube will perform y rotations until the corner is at URF (cubie
 * index 2) or URD (cubie index 7) (
 */
while (cube.getCorner((currentIndex >= 4) ? 7 : 2).compareTo(corner) == -1) {
    cube.rotate("y");
    catalogMoves("y");
}

// i.e. if the corner is in the bottom layer
if (currentIndex >= 4) {
    if (orientation > 0) {
        cube.performAbsoluteMoves("R U' R'");
        catalogMoves("R U' R'");
        solutionExplanation += "The corner is trapped in the bottom layer, so remove it using R U' R'";
    } else {
        cube.performAbsoluteMoves("R U R' U'");
        catalogMoves("R U R' U'");
        solutionExplanation += "The corner is trapped in the bottom layer, so remove it using R U R'";
    }
    solutionExplanation += "\n";
    // The corner is now at cubie index 2, so recursively call the
    // method to solve the corner from this index
    solveCorner(2);
}
/*
 * overCornerIndex is basically the setup position for the corner. If
 * the corner is at this index, then only one more sequence of moves
 * needs to be performed in order to solve the corner
 */
else if (currentIndex == overCornerIndex) {
    if (lastMoveWasU(getCatalogMoves()))
        solutionExplanation += "Bring the corner over its destination. ";

    if (orientation == 0) {
        cube.performAbsoluteMoves("R U2 R' U' R U R'");
        catalogMoves("R U2 R' U' R U R'");
        solutionExplanation += "White is facing top, so perform R U2 R' U' R U R'";
    } else if (orientation == 1) {
        cube.performAbsoluteMoves("R U R'");
        catalogMoves("R U R'");
}
```

```
        solutionExplanation += "White is facing right, so perform R U R'";  
    } else {  
        cube.performAbsoluteMoves("F' U' F");  
        catalogMoves("F' U' F");  
        solutionExplanation += "White is facing front, so perform F' U' F";  
    }  
    solutionExplanation += "\n\n";  
}  
/*  
 * If this block is reached, then it means the corner is in the top  
 * layer, is at cubie index 2, and needs to be moved (using U) to its  
 * setup/overCornerIndex position. This method performs U until the  
 * corner is at its setup position then rotates the cube so that the  
 * corner is still at cubie index 2.  
 */  
else {  
    for (int i = 0; i < ((overCornerIndex - currentIndex) + 4) % 4; ++i) {  
        cube.performAbsoluteMoves("U");  
        catalogMoves("U");  
    }  
  
    for (int i = 0; i < ((overCornerIndex - currentIndex) + 4) % 4; ++i) {  
        cube.rotate("y'");  
        catalogMoves("y'");  
    }  
    solveCorner(2);  
}  
}  
  
/**  
 * @param corner  
 *      the corner to be analysed  
 * @return <b>true</b> if the corner belongs to the first layer; <br>  
 *      false otherwise  
 */  
public static boolean isFLCorner(Corner corner) {  
    Color[] stickers = corner.getStickers();  
  
    if ((stickers[0].equals(Color.white)) || (stickers[1].equals(Color.white)) || (stickers[2].equals(Color.white)))  
        return true;  
    else  
        return false;  
}
```

```
/*
 * public boolean firstLayerCornersSolved() { //I think the bottom bit about
 * AUF is unnecessary since AUF isn't required for corners boolean solved;
 * boolean aufPerformed = false;
 *
 * for (int aUF = 0; aUF < 4; ++aUF) { solved = true; for (int i = 0; i < 4;
 * ++i) { if (!isPieceSolved(fLCorners[i])) { solved = false; break; } }
 *
 * if (solved) { // Catalog only necessary AUF moves if (aufPerformed) for
 * (int i = 0; i < aUF; ++i) { catalogMoves("U"); } return true; } else { //
 * Perform AUF cube.performAbsoluteMoves("u"); aufPerformed = true; }
 *
 * } return false; }
 */

/***
 * @return <b>true</b> if the first-layer corners are solved; <b>false</b>
 * otherwise
 */
public boolean firstLayerCornersSolved() {
    for (int i = 0; i < 4; ++i)
        if (!isPieceSolved(fLCorners[i]))
            return false;

    return true;
}

/*
 * private int getScore(Corner corner) { int score = 0; int index =
 * getIndexOf(corner); int orientation = corner.getOrientation(); int
 * overCornerIndex = getIndexOfDestination(corner); overCornerIndex +=
 * (overCornerIndex % 2 == 0) ? -3 : -5;
 *
 * if (index >= 4) { score -= 6; index = ((index % 2 == 0) ? -3 : -5); }
 * else { if (orientation == 0) score -= 7; else score -= 3; }
 *
 * score -= getShortestOffset(overCornerIndex, index);
 *
 * return score;
 *
 *
 * }
 */
```

```
/**  
 * @param index  
 *          the index of the corner to be examined  
 * @return the number of moves required to solve the specified corner  
 */  
private int getScore(int index) {  
    int score;  
  
    CornerSolver cs = new CornerSolver(cube);  
    cs.solveCorner(index);  
    simplifyMoves(cs.getCatalogMoves(), SolveMaster.CORNER_EDGE);  
    score = cs.getCatalogMoves().size();  
    cube.performAbsoluteMoves(getReverseStringMoves(cs.getCatalogMoves()));  
    cs.clearMoves();  
  
    return score;  
}  
  
/**  
 * @param length  
 *          the number of unsolved corners remaining  
 * @return the index in the solveCandidates array which represents the  
 *         corner that requires the fewest moves to solve  
 */  
private int getIndexOfMinScore(int length) {  
    int min = solveCandidates[0].score;  
    int indexOfMin = 0;  
  
    for (int i = 1; i < length; ++i) {  
        if (solveCandidates[i].score < min) {  
            indexOfMin = i;  
            min = solveCandidates[i].score;  
        }  
    }  
  
    return indexOfMin;  
}  
}
```

## Class CrossSolver

```
package jCube;

import java.awt.Color;
import java.util.LinkedList;

/**
 * @author Kelsey McKenna
 */
public class CrossSolver extends SolveMaster {
    /**
     * Stores the Edge objects representing the white-green, white-red,
     * white-blue, and white-orange edges.
     */
    private static Edge[] crossEdges = new Edge[4];

    /**
     * Constructor - assigns an object to the parent class's 'cube' field
     *
     * @param cube
     *         the cube to be solved
     */
    public CrossSolver(Cube cube) {
        super(cube);

        for (int i = 0; i < 4; ++i) {
            crossEdges[i] = new Edge(Edge.getInitialStickers(i));
        }
    }

    /**
     * Solves the cross of the cube in the main window, and records the solution
     * and explanation at the same time
     */
    public void solveCross() {
        // Stores the index of the current cross edge in the current slice.
        int indexOfCrossEdgeInSlice;
        // Stores the index of the target for the current cross edge in the E
        // slice.
        int workingEdgeTarget;
        // Counts the number of rotations made so that no more than four are
        // performed.
        int count;
```

```
// Stores the expected offset between the current cross edge and a
// solved cross edge in the top slice.
int expectedOffset;
// Stores true if a cross edge is found in the E slice; false otherwise.
boolean workingEdgeFound;

rotateToTop(Color.white);
clearMoves();
catalogMoves("Holding " + Cubie.getColorToWord(cube.getSlice(0).getCentre()) + " top and "
    + Cubie.getColorToWord(cube.getSlice(4).getCentre()) + " front:");

while (!isCrossSolved()) {
    count = 0;
    workingEdgeFound = false;

    /*
     * This searches the middle-layer/E-slice for a cross edge, which is
     * then designated as the 'working' edge. If a cross edge is found,
     * it will be at the FR position because the cube is rotated between
     * each comparison.
     */
    for (count = 0; count < 4; ++count) {
        if (LinearSearch.linearSearch(cube.getSlice(4).getEdge(1).getStickers(), Color.white) != -1) {
            workingEdgeFound = true;
            break;
        } else {
            cube.performAbsoluteMoves("y");
            catalogMoves("y");
        }
    }
}

if (workingEdgeFound) {
    /*
     * If orientation = 0, then the target will be UR, otherwise it
     * will be UF. This is because if the edge's orientation is 0,
     * then you would place it in the top face by performing R, i.e.
     * by placing it at UR, and if the orientation was 1, then you
     * would place it in the top face by performing F, i.e. by
     * placing it at UF
     */
    workingEdgeTarget = cube.getSlice(4).getEdge(1).getOrientation() + 1;
    // Stores a copy of workingEdgeTarget for later manipulation
    int helperTarget = workingEdgeTarget;
```

```
if /* there is a cross edge in U-slice */(((indexOfCrossEdgeInSlice = getIndexOfCrossEdgeInSlice(0, 0)) != -2)
    || ((indexOfCrossEdgeInSlice = getIndexOfCrossEdgeInSlice(0, 1)) != -2))) {
    if /* the orientation of the found cross edge is 0 */(cube.getSlice(0)
        .getEdge(indexOfCrossEdgeInSlice).getOrientation() == 0) {
        /*
         * This calculates the expected offset (relative
         * distance) between the working edge and the existing
         * cross edge. See getExpectedOffset(...) for more
         * information.
        */
        expectedOffset = getExpectedOffset(
            cube.getSlice(0).getEdge(indexOfCrossEdgeInSlice).getStickers()[1],
            cube.getSlice(4).getEdge(1).getStickers()[(cube.getSlice(4).getEdge(1).getOrientation() + 1) % 2]);

        indexOfCrossEdgeInSlice = (indexOfCrossEdgeInSlice == 3) ? -1 : indexOfCrossEdgeInSlice;

        /*
         * This moves the existing cross edge to the correct
         * position.
        */
        while ((workingEdgeTarget - indexOfCrossEdgeInSlice != expectedOffset)
            && (workingEdgeTarget - indexOfCrossEdgeInSlice != (expectedOffset + 4) % 4)) {
            cube.performAbsoluteMoves("U");
            catalogMoves("U");
            indexOfCrossEdgeInSlice = ((indexOfCrossEdgeInSlice + 2) % 4) - 1;
        }
    }
}
/*
 * Reaching this block means there is an unoriented edge in
 * the top slice, so you can remove this from the top slice
 * and solve the working edge at the same time.
*/
else {
    indexOfCrossEdgeInSlice = (indexOfCrossEdgeInSlice == 3) ? -1 : indexOfCrossEdgeInSlice;
    for (int i = 0; i < (workingEdgeTarget - indexOfCrossEdgeInSlice + 4) % 4; ++i) {
        cube.performAbsoluteMoves("U");
        catalogMoves("U");
    }
}
} else if /* there is an edge in bottom slice */((indexOfCrossEdgeInSlice = getIndexOfCrossEdgeInSlice(
    1, 0)) != -2) || ((indexOfCrossEdgeInSlice = getIndexOfCrossEdgeInSlice(1, 1)) != -2)) {
    indexOfCrossEdgeInSlice = (indexOfCrossEdgeInSlice == 3) ? -1 : indexOfCrossEdgeInSlice;
}
/*
```

```
* If the bottom slice contains an unoriented edge, then you
* want to place this edge in the plane of motion of the
* working edge so that you can solve the working edge and
* fix the bad edge in the bottom layer at the same time.
*/
if (sliceContainsCrossEdgeOriented(1, 1)) {
    helperTarget = (helperTarget + 2) % 4;

    while (cube.getSlice(1).getEdge(helperTarget).getStickers()[1] != Color.white) {
        cube.performAbsoluteMoves("D");
        catalogMoves("D");
    }
} else {
    while (isCrossEdge(cube.getSlice(1).getEdge((helperTarget)))) {
        cube.performAbsoluteMoves("D");
        catalogMoves("D");
    }
}
} else { // All edges are in the E-Slice

    if /* an oriented edge will become misoriented */(cube.getSlice(helperTarget + 1).getEdge(1)
        .getOrientation() == (helperTarget % 2)) {
        cube.performAbsoluteMoves((helperTarget == 1) ? "B'" : "L");
        catalogMoves((workingEdgeTarget == 1) ? "B'" : "L");
    }
}

cube.performAbsoluteMoves((workingEdgeTarget == 1) ? "R" : "F'");
catalogMoves((workingEdgeTarget == 1) ? "R" : "F'");
} else { // There are no edges in the top or bottom layers
// Stores the index of the cross edge's destination in the top
// slice.
int uSliceIndex = 0;
// Stores the location of the cross edge in the bottom slice.
int dSliceIndex;
count = 0;

if /* there is an oriented cross edge in the bottom slice */((dSliceIndex = getIndexOfCrossEdgeInSlice(
    1, 0)) != -2) {
    if (dSliceIndex == 1)
        uSliceIndex = 3;
    else if (dSliceIndex == 3)
        uSliceIndex = 1;
    else
```

```
uSliceIndex = dSliceIndex;

/*
 * Move any unoriented cross edges in the top slice to the
 * target for the edge found in the bottom slice so that two
 * operations can be performed at once.
 */
if (sliceContainsCrossEdgeOriented(0, 1))
    while (cube.getSlice(0).getEdge(uSliceIndex).getStickers()[1] != Color.white) {
        cube.performAbsoluteMoves("u");
        catalogMoves("U");
    }
else {
    /*
     * Move any existing (and oriented) cross edges out of
     * the way
     */
    while (isCrossEdge(cube.getSlice(0).getEdge(uSliceIndex))) {
        cube.performAbsoluteMoves("u");
        catalogMoves("U");
    }
}
/*
 * If there is an unoriented cross edge in the top slice and no
 * oriented edges in the bottom slice
 */
else if ((uSliceIndex = getIndexOfCrossEdgeInSlice(0, 1)) != -2) {
    if (uSliceIndex == 1)
        dSliceIndex = 3;
    else if (uSliceIndex == 3)
        dSliceIndex = 1;
    else
        dSliceIndex = uSliceIndex;

    /*
     * If there is an unoriented cross edge in the bottom
     * slice...
     */
    if (sliceContainsCrossEdgeOriented(1, 1)) {
        while (cube.getSlice(1).getEdge(dSliceIndex).getStickers()[1] != Color.white) {
            cube.performAbsoluteMoves("d");
            catalogMoves("D");
        }
    }
}
```

```
        } else {
            while (isCrossEdge(cube.getSlice(1).getEdge(dSliceIndex))) {
                cube.performAbsoluteMoves("d");
                catalogMoves("D");
            }
        }
    /*
     * If there is an unoriented cross edge in the bottom slice and
     * no unoriented cross edges in the top slice
     */
    else {
        dSliceIndex = getIndexOfCrossEdgeInSlice(1, 1);
        count = 0;

        if (dSliceIndex == 1)
            uSliceIndex = 3;
        else if (dSliceIndex == 3)
            uSliceIndex = 1;
        else
            uSliceIndex = dSliceIndex;

        try {
            /*
             * Move oriented cross edges out of the way
             */
            while ((count < 4) && (isCrossEdge(cube.getSlice(0).getEdge(uSliceIndex)))) {
                ++count;
                dSliceIndex = (dSliceIndex + 1) % 4;

                if (dSliceIndex == 1)
                    uSliceIndex = 3;
                else if (dSliceIndex == 3)
                    uSliceIndex = 1;
                else
                    uSliceIndex = dSliceIndex;

                cube.performAbsoluteMoves("d");
                catalogMoves("D");
            }
        }
    /*
     * This exception is thrown if all cross edges are in the
     * top layer, hence uSliceIndex will be negative
    
```

```
        */
        catch (ArrayIndexOutOfBoundsException e) {
            if (edgesAreOpposite(cube.getEdge(0), cube.getEdge(2))) {
                cube.performAbsoluteMoves("R U2 R' U2 R");
                catalogMoves("R U2 R' U2 R");
            } else if (edgesAreOpposite(cube.getEdge(0), cube.getEdge(1))) {
                cube.performAbsoluteMoves("R' U' R U'");
                catalogMoves("R' U' R U'");
            } else {
                cube.performAbsoluteMoves("L U L' U");
                catalogMoves("L U L' U");
            }
        }
    }

/*
 * Perform the move required to get the edge in the top slice.
 */
switch (uSliceIndex) {
case 0:
    cube.performAbsoluteMoves("B");
    catalogMoves("B");
    break;
case 1:
    cube.performAbsoluteMoves("R");
    catalogMoves("R");
    break;
case 2:
    cube.performAbsoluteMoves("F");
    catalogMoves("F");
    break;
case 3:
    cube.performAbsoluteMoves("L");
    catalogMoves("L");
    break;
default:
    break;
}
}

// Stores the y moves in the solution.
LinkedList<String> temp = new LinkedList<>();
LinkedList<String> catalogMoves = getCatalogMoves();
```

```
for (int i = 0; i < catalogMoves.size(); ++i) {
    if (catalogMoves.get(i).contains("y"))
        temp.add(catalogMoves.get(i));
}

simplifyMoves(temp, SolveMaster.CANCELLATIONS);

if (temp.size() > 0) {
    if (temp.get(0).contains("'")) {
        // The cross doesn't require any
        // rotations, so undo any
        // rotations performed.
        cube.rotate("y");
    } else if (temp.get(0).contains("2")) {
        cube.rotate("y2");
    } else {
        cube.rotate("y'");
    }
}

cube.rotate("z2");
catalogMoves("z2");
}

/**
 * @param edgeOne
 *          this Edge is compared to the second Edge
 * @param edgeTwo
 *          this Edge is compared to the first Edge
 * @return <b>true</b> if the edges are opposite each other and are on the
 *         same face on a solved cube; <br>
 *         <b>false</b> otherwise
 */
private boolean edgesAreOpposite(Edge edgeOne, Edge edgeTwo) {
    Color colorOne, colorTwo;

    if ((colorOne = edgeOne.getStickers()[1]).equals(Color.white))
        colorOne = edgeOne.getStickers()[0];
    if ((colorTwo = edgeTwo.getStickers()[1]).equals(Color.white))
        colorOne = edgeTwo.getStickers()[0];

    int one = -1, two = -1;
```

```
for (int i = 0; i < 4; ++i) {
    if (crossEdges[i].getStickers()[1].equals(colorOne))
        one = i;
    else if (crossEdges[i].getStickers()[1].equals(colorTwo))
        two = i;
}

return ((one + two) % 2 == 0);
}

/**
 * @param sliceIndex
 *         the index of the slice to be analysed
 * @param orientation
 *         the orientation to be found
 * @return <b>true</b> if the specified slice contains a cross edge with the
 *         specified orientation; <br>
 *         <b>false</b> otherwise
 */
private boolean sliceContainsCrossEdgeOriented(int sliceIndex, int orientation) {
    Slice slice = cube.getSlice(sliceIndex);
    Edge edge;

    for (int i = 0; i < 4; ++i) {
        edge = slice.getEdge(i);
        if ((edge.getOrientation() == orientation)
            && ((edge.getStickers()[1] == Color.white || edge.getStickers()[1] == Color.white)))
            return true;
    }

    return false;
}

/**
 * @param edge
 *         the edge to be analysed
 * @return <b>true</b> if the specified edge is a cross edge, i.e. the edge
 *         contains a white sticker; <br>
 *         <b>false</b> otherwise
 */
public static boolean isCrossEdge(Edge edge) {
    Color[] stickers = edge.getStickers();
    if ((stickers[0] == Color.white) || (stickers[1] == Color.white))
        return true;
```

```
        else
            return false;
    }

    /**
     * @param relativeSolvedEdge
     *         the secondary colour of a cross edge that is solved
     *         (relatively) to other pieces
     * @param workingEdge
     *         the secondary colour of a cross edge that is not solved at all
     * @return the expected offset between the two cross edges on a solved cube
     */
    private/* public */int getExpectedOffset(Color relativeSolvedEdge, Color workingEdge) {
        int end = 0, start = 0;

        for (int i = 0; i < 4; ++i) {
            if (crossEdges[i].getSecondaryColor().equals(relativeSolvedEdge))
                start = i;
            else if (crossEdges[i].getSecondaryColor().equals(workingEdge))
                end = i;
        }

        if (end - start <= -2)
            return (end - start) + 4;
        else if (end - start == 3)
            return -1;
        else
            return end - start;
    }

    /**
     * @param sliceIndex
     *         the index of the slice whose edges are to be searched
     * @param orientation
     *         the orientation of the found cross edge must be of this
     *         orientation
     * @return the index of a cross edge in the specified slice with the
     *         specified orientation
     */
    private int getIndexOfCrossEdgeInSlice(int sliceIndex, int orientation) {
        Slice slice = cube.getSlice(sliceIndex);
        Color[] stickers;
        for (int i = 0; i < 4; ++i) {
            stickers = slice.getEdge(i).getStickers();
```

```
        if (stickers[orientation].equals(Color.white))
            return i;
    }

    return -2;
}

/*
 * private int getNumCrossEdgesOriented() { int numOriented = 4; Edge[]
 * edges = cube.getEdges(); Color[] stickers;
 *
 * for (int i = 0; i < 12; ++i) { stickers = edges[i].getStickers(); if
 * (stickers[0].equals(Color.white) || stickers[1].equals(Color.white))
 * numOriented -= edges[i].getOrientation(); }
 *
 * return numOriented; }
 */

/**
 * Indicates whether or not the cross is solved. Solves AUF at the same
 * time.
 *
 * @return <b>true</b> if the cross is solved; <br>
 *         <b>false</b> otherwise
 */
public boolean isCrossSolved() {
    boolean solved;

    for (int count = 0; count < 4; ++count) {
        solved = true;
        for (int i = 0; i < 4; ++i) {
            if (!isPieceSolved(crossEdges[i])) {
                solved = false;
                break;
            }
        }

        if (solved)
            return true;
        else { // Check AUF
            cube.performAbsoluteMoves("U");
            catalogMoves("U");
        }
    }
}
```

```
        }
        return false;
    }

    /*
     * private boolean uFaceChanged(Slice newSlice, Edge[] originalSliceEdges) {
     * Edge currentEdge; boolean found = false;
     *
     * for (int i = 0; i < 4; ++i) { found = false; currentEdge =
     * originalSliceEdges[i];
     *
     * if (isCrossEdge(currentEdge)) { for (int j = 0; j < 4; ++j) { if
     * (newSlice.getEdge(j).compareTo(currentEdge) == 0) { found = true; break;
     * } } if (!found) return true; else continue; }
     *
     * currentEdge = newSlice.getEdge(i); if (isCrossEdge(currentEdge)) { for
     * (int j = 0; j < 4; ++j) { if
     * (originalSliceEdges[j].compareTo(currentEdge) == 0) { found = true;
     * break; } } if (!found) return true; } }
     *
     * return false; }
     *
     * public String getSolutionExplanation() { String solutionExplanation =
     * super.solutionExplanation; LinkedList<String> moves = getCatalogMoves();
     * int size = moves.size(); Cube tempCube = new Cube(); int edgeIndex = 0,
     * orientation = 0; Color[] stickers; String currentMove; Edge[] oldEdges =
     * new Edge[4];
     *
     * if (size == 0) return "";
     *
     * for (int i = 0; i < 4; ++i) oldEdges[i] = new Edge();
     *
     * tempCube.rotate("z2");
     *
     * while (!tempCube.getSlice(4).getCentre().equals(originalCentre))
     * tempCube.rotate("y");
     *
     * tempCube.performAbsoluteMoves(getReverseStringMoves(moves));
     *
     * for (int i = 0; i < size; ++i) { currentMove = moves.get(i);
     *
     * for (int j = 0; j < 4; ++j)
     * oldEdges[j].setStickers(Arrays.copyOf(tempCube.getEdge(j).getStickers(),
     * 2)); }
```

```
*  
* if (!"xyz".contains(currentMove.substring(0, 1)))  
* tempCube.performAbsoluteMoves(currentMove);  
*  
* if (uFaceChanged(tempCube.getSlice(0), oldEdges)) { switch  
* (moves.get(i).substring(0, 1)) { case "F": edgeIndex = 2; orientation =  
* 1; break; case "B": edgeIndex = 0; orientation = 1; break; case "R":  
* edgeIndex = 1; orientation = 0; break; case "L": edgeIndex = 3;  
* orientation = 0; break; default: break; }  
*  
* stickers = tempCube.getEdge(edgeIndex).getStickers();  
*  
* solutionExplanation +=  
* String.format("Insert the %s-%s edge into the U face%n",  
* Cubie.getColorToWord(stickers[orientation]),  
* Cubie.getColorToWord(stickers[(orientation + 1) % 2])); } }  
*  
* return solutionExplanation; }  
*/  
}
```

## Class Cube

```
package jCube;

import java.awt.Color;

/*
 * Initial orientation = White-top Green-front
 */

/***
 * @author Kelsey McKenna
 */
public class Cube {
    /**
     * NUM_SLICES is used instead of 6 throughout for readability
     */
    private static final int NUM_SLICES = 6;
    /**
     * NUM_EDGES is used instead of 12 throughout for readability
     */
    private static final int NUM_EDGES = 12;
    /**
     * NUM_CORNERS is used instead of 8 throughout for readability
     */
    private static final int NUM_CORNERS = 8;
    /**
     * This two-dimensional array stores the indices of the slices involved in a
     * corresponding rotation. The first element stores the slices involved in
     * 'x' rotations, the second stores those involved in 'y' rotations, and the
     * third stores those involve in 'z' rotations.
     */
    private static int[][] rotationSlices = { { 0, 5, 1, 4 }, // x
                                              { 2, 4, 3, 5 }, // y
                                              { 0, 2, 1, 3 }, // z
                                            };
    /**
     * The ith element of this array stores the cubie indices for the ith slice.
     * The cubie indices indicate the index of a slice's cubies in relation to
     * the rest of the cube.
     */
    private static int[][] cubieIndices = { { 0, 1, 2, 3, 0, 1, 2, 3 }, { 4, 5, 6, 7, 8, 9, 10, 11 },
                                           { 2, 1, 4, 7, 1, 5, 11, 6 }, { 0, 3, 6, 5, 3, 7, 9, 4 }, { 3, 2, 7, 6, 2, 6, 10, 7 },
                                           { 1, 0, 5, 4, 0, 4, 8, 5 }, };
}
```

```
/**  
 * This stores the Edges of the cube.  
 */  
private Edge[] edges = new Edge[NUM_EDGES];  
/**  
 * This stores the corners of the cube.  
 */  
private Corner[] corners = new Corner[NUM_CORNERS];  
/**  
 * This stores the slices of the cube. U slice -> slices[0], D slice ->  
 * slices[1], R slice -> slices[2], L slice -> slices[3], F slice ->  
 * slices[4], B slice -> slices[5]  
 */  
private Slice[] slices = new Slice[NUM_SLICES];  
  
/**  
 * Constructor - sets up cube for display etc.  
 */  
public Cube() {  
    resetCube();  
}  
  
/**  
 * Constructor - acts as a clone  
 *  
 * @param cube  
 *          the cube to be cloned  
 */  
public Cube(Cube cube) {  
    for (int i = 0; i < NUM_EDGES; ++i) {  
        edges[i] = new Edge(cube.getEdge(i));  
    }  
    for (int i = 0; i < NUM_CORNERS; ++i) {  
        corners[i] = new Corner(cube.getCorner(i));  
    }  
    for (int i = 0; i < NUM_SLICES; ++i) {  
        slices[i] = new Slice(edges, corners);  
    }  
}  
  
/**  
 * Resets the indices of the cubies to their original state  
 */  
private void resetCubieIndices() {
```

```
for (int i = 0; i < 8; ++i)
    corners[i].setCubieIndex(i);

for (int i = 0; i < 12; ++i)
    edges[i].setCubieIndex(i);

for (int slice = 0; slice < NUM_SLICES; ++slice) {
    slices[slice].setCubieIndices(cubieIndices[slice]);
}
}

/**
 * @param index
 *          the index of the slice to be returned
 * @return the <b>index</b>th slice
 */
public Slice getSlice(int index) {
    return slices[index];
}

/**
 * @param face
 *          the name of the face which represents the slice to be returned
 * @return the slice which is associated with the specified face
 */
public Slice getSlice(String face) {
    switch (face) {
    case "U":
        return slices[0];
    case "D":
        return slices[1];
    case "R":
        return slices[2];
    case "L":
        return slices[3];
    case "F":
        return slices[4];
    case "B":
        return slices[5];
    default:
        return null;
    }
}
```

```
/**  
 * Performs the specified move on the cube  
 *  
 * @param m  
 *         the move to be performed in WCA notation  
 */  
private void performAbsoluteMove(String m) {  
    m = m.toLowerCase();  
  
    switch (m) {  
        case "u":  
            performMove('j');  
            break;  
        case "u'":  
            performMove('f');  
            break;  
        case "u2":  
            performMove('j');  
            performMove('j');  
            break;  
        case "d":  
            performMove('s');  
            break;  
        case "d'":  
            performMove('l');  
            break;  
        case "d2":  
            performMove('l');  
            performMove('l');  
            break;  
        case "r":  
            performMove('i');  
            break;  
        case "r'":  
            performMove('k');  
            break;  
        case "r2":  
            performMove('k');  
            performMove('k');  
            break;  
        case "l":  
            performMove('d');  
            break;  
        case "l'":
```

```
    performMove('e');
    break;
case "l2":
    performMove('e');
    performMove('e');
    break;
case "f":
    performMove('h');
    break;
case "f'":
    performMove('g');
    break;
case "f2":
    performMove('g');
    performMove('g');
    break;
case "b":
    performMove('w');
    break;
case "b'":
    performMove('o');
    break;
case "b2":
    performMove('o');
    performMove('o');
    break;
case "rw":
    performMove('u');
    break;
case "rw'":
    performMove('m');
    break;
case "lw":
    performMove('v');
    break;
case "lw'":
    performMove('r');
    break;
case "m":
    performMove('x');
    break;
case "m'":
    performMove('.');
    break;
```

```
        case "m2":
            performMove('.');
            performMove('.');
            break;
        default:
            rotate(m);
            break;
    }
}

/**
 * Performs the specified moves on the cube
 *
 * @param moves
 *         the moves to be performed in WCA notation
 */
public void performAbsoluteMoves(String moves) {
    if (moves == null)
        return;

    moves = moves.trim();
    if (moves.equals(""))
        return;

    String currentMove;
    int i = 0;
    int indexOfSpace;
    String remainingMoves;

    while (i < moves.length()) {
        remainingMoves = moves.substring(i);
        indexOfSpace = remainingMoves.indexOf(" ");

        if (indexOfSpace == -1)
            indexOfSpace = remainingMoves.length();

        currentMove = remainingMoves.substring(0, indexOfSpace).trim();
        // if ("xyz".contains(currentMove.substring(0, 1)))
        // rotate(currentMove);
        // else

        performAbsoluteMove(currentMove); // perform absolute move handles
                                         // rotations
```

```
        i += indexOfSpace + 1;
    }
}

/**
 * Performs the specified move on the cube
 *
 * @param m
 *         the move to be performed represented by a key on keyboard
 *         (i.e. not WCA notation)
 */
public void performMove(char m) {
    String move = Character.toString(m).toLowerCase();
    switch (move) {
        case "j": // U
            slices[0].performMove(1);
            updateCubies(0);
            break;
        case "f": // U'
            slices[0].performMove(-1);
            updateCubies(0);
            break;
        case "i": // R
            slices[2].performMove(1);
            slices[2].twistAllCorners(1);
            updateCubies(2);
            break;
        case "k": // R'
            slices[2].performMove(-1);
            slices[2].twistAllCorners(-1);
            updateCubies(2);
            break;
        case "d": // L
            slices[3].performMove(1);
            slices[3].twistAllCorners(1);
            updateCubies(3);
            break;
        case "e": // L'
            slices[3].performMove(-1);
            slices[3].twistAllCorners(-1);
            updateCubies(3);
            break;
        case "s": // D
            slices[1].performMove(1);
```

```
updateCubies(1);
break;
case "l": // D'
slices[1].performMove(-1);
updateCubies(1);
break;
case "h": // F
slices[4].performMove(1);
slices[4].flipAllEdges();
slices[4].twistAllCorners(1);
updateCubies(4);
break;
case "g": // F'
slices[4].performMove(-1);
slices[4].flipAllEdges();
slices[4].twistAllCorners(-1);
updateCubies(4);
break;
case "w": // B
slices[5].performMove(1);
slices[5].flipAllEdges();
slices[5].twistAllCorners(1);
updateCubies(5);
break;
case "o": // B'
slices[5].performMove(-1);
slices[5].flipAllEdges();
slices[5].twistAllCorners(-1);
updateCubies(5);
break;
case "u": // Rw
this.performMove('d');
this.rotate("x");
break;
case "m": // R'w
performMove('e');
rotate("x'");
break;
case "r": // Lw
performMove('k');
rotate("x");
break;
case "v": // L'w
performMove('i');
```

```
        rotate("x'");
        break;
    case "x": // M
        performMove('i');
        performMove('e');
        rotate("x'");
        break;
    case ".": // M'
        performMove('k');
        performMove('d');
        rotate("x");
        break;
    case ";":
        rotate("y");
        break;
    case "a":
        rotate("y'");
        break;
    case "y":
        rotate("x");
        break;
    case "n":
        rotate("x'");
        break;
    case "p":
        rotate("z");
        break;
    case "q":
        rotate("z'");
        break;
    default:
        break;
    }

    updateAll();
}

/**
 * Performs the specified rotation on the cube
 *
 * @param rotation
 *         the rotation to be performed - this should one of the
 *         following: <br>
 *         {x, x', y, y', z, z'}

```

```
/*
public void rotate(String rotation) {
    /*
     * If the argument is null, empty, or does not start with 'x', 'y', or
     * 'z', then nothing can be done, so just return.
     */
    if ((rotation == null) || (rotation.trim().length() == 0) || (!"xyz".contains(rotation.substring(0, 1))))
        return;

    if (rotation.equals("z")) {
        rot("y");
        rot("x'");
        rot("y'");
    } else if (rotation.equals("z'")) {
        rot("y'");
        rot("x'");
        rot("y");
    } else {
        if (rotation.contains("2")) {
            for (int i = 0; i < 2; ++i)
                rotate(rotation.substring(0, 1));
        } else {
            rot(rotation);
        }
    }
}

/**
 * Performs the specified rotation on the cube
 *
 * @param rotation
 *         the rotation to be performed - must be "x" or "y" without "2"s
 */
private void rot(String rotation) {
    // Stores the indices of the slices affected by the rotation.
    int[] sliceIndices = rotationSlices[((int) rotation.toCharArray()[0] - (int) 'x')];
    // Stores a Slice temporarily so that the cycle of Slices can be
    // completed.
    Slice tempSlice = slices[sliceIndices[0]];
    // Stores the index of the next slice to be changed.
    int nextIndex;
    // Stores the direction in which the slices should be cycled.
    int direction = (rotation.contains("") ? -1 : 1);
    // Stores the index of the element in slicesIndices that stores the
```

```
// index of the last slice that should be changed.
int end = (4 + direction) % 4;

/**
 * This cycles around the slices with the indices specified in
 * sliceIndices, e.g. <br>
 * 0 -> 5 <br>
 * 5 -> 1 <br>
 * 1 -> 4 <br>
 * 4 -> 0 <br>
 * would be the cycle order for an "x" rotation. <br>
 */
for (int i = 0; i != end; i = (i - direction + 4) % 4) {
    nextIndex = (i - direction + 4) % 4;
    slices[sliceIndices[i]] = slices[sliceIndices[nextIndex]];
    performRotationMaintenance(rotation, sliceIndices[i]);
    updateCubies(sliceIndices[i]);
}

/**
 * This is the final step of the cycle. This is similar to declaring a
 * 'temp' variable when performing a swap, e.g. <br>
 * temp <- a <br>
 * a <- b <br>
 * b <- temp
 */
slices[sliceIndices[end]] = tempSlice;
performRotationMaintenance(rotation, sliceIndices[end]);
updateCubies(sliceIndices[end]);

/*
 * This performs the necessary operations to ensure that the
 * corners/edges are twisted/flipped correctly after a particular
 * orientation.
 */
switch (rotation) {
case "x'":
case "x":
    slices[0].twistAllCorners(1);
    updateCubies(0);
    slices[1].twistAllCorners(-1);
    updateCubies(1);
    break;
case "y":
```

```
        case "Y'":
            for (int i = 0; i < 4; ++i) {
                slices[sliceIndices[i]].flipEdge(1);
                updateCubies(sliceIndices[i]);
            }

            break;
        default:
            break;
    }

    if (rotation.equals("x'")) {
        slices[0].performMove(2);
        updateCubies(0);
        slices[1].performMove(2);
        updateCubies(1);
    }
    /*
     * Update all cubies in each slice to the cubies in this instance of
     * Cube.
     */
    updateAll();
}

/*
 * private void rot(String move) { Edge tempEdge; Color tempCentre;
 *
 * switch(move) { case "x": tempEdge = edges[0]; tempCentre =
 * slices[0].getCentre();
 *
 * edges[0] = edges[2]; edges[2] = edges[10]; edges[10] = edges[8]; edges[8]
 * = tempEdge;
 *
 * slices[0].setCentre(slices[4].getCentre());
 * slices[4].setCentre(slices[1].getCentre());
 * slices[1].setCentre(slices[5].getCentre());
 * slices[5].setCentre(tempCentre);
 *
 * performAbsoluteMoves("R"); performAbsoluteMoves("L");
 *
 * edges[0].flip(); edges[2].flip(); edges[8].flip(); edges[10].flip();
 * break; case "x'": tempEdge = edges[0]; tempCentre =
 * slices[0].getCentre();
 *
```

```
* edges[0] = edges[8]; edges[8] = edges[10]; edges[10] = edges[2]; edges[2]
* = tempEdge;
*
* slices[0].setCentre(slices[5].getCentre());
* slices[5].setCentre(slices[1].getCentre());
* slices[1].setCentre(slices[4].getCentre());
* slices[4].setCentre(tempCentre);
*
* performAbsoluteMoves("R'"); performAbsoluteMoves("L");
*
* edges[0].flip(); edges[2].flip(); edges[8].flip(); edges[10].flip();
* break; case "y": tempEdge = edges[4]; tempCentre = slices[4].getCentre();
*
* edges[4] = edges[7]; edges[7] = edges[6]; edges[6] = edges[5]; edges[5] =
* tempEdge;
*
* slices[4].setCentre(slices[2].getCentre());
* slices[2].setCentre(slices[5].getCentre());
* slices[5].setCentre(slices[3].getCentre());
* slices[3].setCentre(tempCentre);
*
* performAbsoluteMoves("U"); performAbsoluteMoves("D'");
*
* edges[4].flip(); edges[5].flip(); edges[6].flip(); edges[7].flip();
* break; case "y'": tempEdge = edges[4]; tempCentre =
* slices[4].getCentre();
*
* edges[4] = edges[5]; edges[5] = edges[6]; edges[6] = edges[7]; edges[7] =
* tempEdge;
*
* slices[4].setCentre(slices[3].getCentre());
* slices[3].setCentre(slices[5].getCentre());
* slices[5].setCentre(slices[2].getCentre());
* slices[2].setCentre(tempCentre);
*
* performAbsoluteMoves("U'"); performAbsoluteMoves("D");
*
* edges[4].flip(); edges[5].flip(); edges[6].flip(); edges[7].flip();
* break; }
*
* if (move.contains("x")) { for (int i = 0; i < 12; ++i) { if ((i == 0) ||
* (i == 2) || (i == 8) || (i == 10)) continue;
*
* if (isTestLEdge(edges[i])) edges[i].flipOrientation(); } }
```

```
/*
 * else { for (int i = 0; i < 4; ++i) { if (isMLEdge(edges[i]))
 * edges[i].flipOrientation(); }
 *
 * for (int i = 8; i < 12; ++i) { if (isMLEdge(edges[i]))
 * edges[i].flipOrientation(); } }
 *
 *
 * updateAll(); }
 */

/*
 * @SuppressWarnings("unused") private boolean isMLEdge(Edge edge) { Color[]
 * stickers = edge.getStickers();
 *
 * if (!stickers[0].equals(slices[0].getCentre())) &&
 * (!stickers[1].equals(slices[0].getCentre())) &&
 * (!stickers[0].equals(slices[1].getCentre())) &&
 * (!stickers[1].equals(slices[1].getCentre()))) return true; else return
 * false; }
 */
/*
 * @SuppressWarnings("unused") private boolean isTestLEdge(Edge edge) {
 * Color[] stickers = edge.getStickers();
 *
 * if (!stickers[0].equals(slices[2].getCentre())) &&
 * (!stickers[1].equals(slices[2].getCentre())) &&
 * (!stickers[0].equals(slices[3].getCentre())) &&
 * (!stickers[1].equals(slices[3].getCentre()))) return true; else return
 * false; }
 */

/**
 * Fixes the the cubies after an x rotation
 *
 * @param rotation
 *          the rotation to be performed
 * @param i
 *          the index of the slice affected
 */
private void performRotationMaintenance(String rotation, int i) {
    if (rotation.equals("x")) {
        if (i >= 4) {
            slices[i].swapEdges(1, 3);
```

```
        slices[i].flipEdge(0);
        slices[i].flipEdge(2);
    }
} else if (rotation.equals("x'")) {
    if (i <= 1) {
        slices[i].flipEdge(0);
        slices[i].flipEdge(2);
    }
}
}

/**
 * This method acts as an interface between the cubies of each slice and the
 * all the cubies in this class. This means that when one slice is changed,
 * you can change any corresponding slices using the 'updateAll()' method
 *
 * @param sliceIndex
 *         the slice from which the updates should be taken
 */
public void updateCubies(int sliceIndex) {
    Edge cEdge;
    Corner cCorner;

    for (int cubieIndex = 0; cubieIndex < 4; ++cubieIndex) {
        cCorner = slices[sliceIndex].getCorner(cubieIndex);
        cEdge = slices[sliceIndex].getEdge(cubieIndex);

        this.corners[cubieIndices[sliceIndex][cubieIndex]] = cCorner;
        this.edges[cubieIndices[sliceIndex][cubieIndex + 4]] = cEdge;
    }
}

/**
 * Updates the cubies in each slice based on the cubies in this instance of
 * Cube.
 */
public void updateAll() {
    for (int sliceIndex = 0; sliceIndex < NUM_SLICES; ++sliceIndex) {
        for (int cubieIndex = 0; cubieIndex < 4; ++cubieIndex) {
            slices[sliceIndex].setCorner(cubieIndex, corners[cubieIndices[sliceIndex][cubieIndex]]);
            slices[sliceIndex].setEdge(cubieIndex, edges[(cubieIndices[sliceIndex][cubieIndex + 4])]);
        }
    }
}
```

```
    resetCubeIndices();  
}  
  
/**  
 * @return the edges of the cube  
 */  
public Edge[] getEdges() {  
    return edges;  
}  
  
/**  
 * @return the corners of the cube  
 */  
public Corner[] get Corners() {  
    return corners;  
}  
  
/**  
 * @return the slices of the cube  
 */  
public Slice[] getSlices() {  
    return slices;  
}  
  
/**  
 * @param index  
 *          the index of the edge to be returned  
 * @return the <b>index</b>th edge  
 */  
public Edge getEdge(int index) {  
    return edges[index];  
}  
  
/**  
 * @param index  
 *          the index of the corner to be returned  
 * @return the <b>index</b>th corner  
 */  
public Corner getCorner(int index) {  
    return corners[index];  
}  
  
/**  
 * Resets the cube to its initial solved state with white on top and green
```

```
* on front.  
*/  
public void resetCube() {  
    for (int i = 0; i < NUM_EDGES; ++i)  
        edges[i] = new Edge();  
  
    for (int i = 0; i < NUM_CORNERS; ++i)  
        corners[i] = new Corner();  
  
    for (int i = 0; i < NUM_SLICES; ++i)  
        slices[i] = new Slice(cubieIndices[i]);  
  
    slices[0].setCentre(Color.white);  
    slices[1].setCentre(Color.yellow);  
    slices[2].setCentre(Color.red);  
    slices[3].setCentre(Cubie.orange);  
    slices[4].setCentre(Color.green);  
    slices[5].setCentre(Color.blue);  
  
    for (int i = 0; i < 8; ++i) { // Give cube initial colors  
        edges[i].setStickers(Edge.getInitialStickers(i));  
        edges[i].setCubieIndex(i);  
        corners[i].setStickers(Corner.getInitialStickers(i));  
        corners[i].setCubieIndex(i);  
    }  
  
    for (int i = 8; i < 12; ++i) {  
        edges[i].setStickers(Edge.getInitialStickers(i));  
        edges[i].setCubieIndex(i);  
    }  
  
    updateAll();  
}  
}
```

## Class Cubie

```
package jCube;

import java.awt.Color;
import java.util.Arrays;

/**
 * @author Kelsey McKenna
 */
public class Cubie implements Comparable<Cubie> {
    /**
     * Custom colour definition for orange
     */
    public static final Color orange = new Color(255, 132, 10);
    /**
     * This indicates the orientation of the cubie. 0 = Oriented, 1 =
     * Flipped/Clockwise, -1 = Anticlockwise
     */
    private int orientation = 0;
    /**
     * This stores the stickers of the cubie.
     */
    private Color[] stickers;
    /**
     * This stores the index of the cubie on the cube in relation to the other
     * pieces.
     */
    private int cubieIndex;

    /**
     * Constructor - empty
     */
    public Cubie() {
    }

    /**
     * Constructor - assigns stickers to the cubie
     *
     * @param stickers
     *         the stickers to be assigned to the cube
     */
    public Cubie(Color[] stickers) {
        this.stickers = stickers;
    }
}
```

```
}

/**
 * Sets the stickers of the cubie to the <b>stickers</b> parameter
 *
 * @param stickers
 *         the stickers to be assigned to the cubie
 */
public void setStickers(Color[] stickers) {
    this.stickers = stickers;
}

/**
 * Sets the cubieIndex of the cubie to the parameter
 *
 * @param cubieIndex
 *         the new cubie index
 */
public void setCubieIndex(int cubieIndex) {
    this.cubieIndex = cubieIndex;
}

/**
 * Sets the orientation of the cubie to the parameter
 *
 * @param orientation
 *         the value to be assigned to the orientation
 */
public void setOrientation(int orientation) {
    this.orientation = orientation;
}

/**
 * Sets the <b>index</b>th sticker of the cubie to <b>color</b>
 *
 * @param index
 *         the index of the sticker to be changed
 * @param color
 *         the new color for the <b>index</b>th sticker
 */
public void setSticker(int index, Color color) {
    Color[] stickers = Arrays.copyOf(getStickers(), getStickers().length);
    stickers[index] = color;
    setStickers(stickers);
```

```
}

/**
 * @return the orientation of the cubie
 */
public int getOrientation() {
    return orientation;
}

/**
 * @return an array of colors representing the stickers of the cubie
 */
public Color[] getStickers() {
    return stickers;
}

/**
 * @return the cubie index associated with this cubie
 */
public int getCubieIndex() {
    return cubieIndex;
}

/**
 * @param otherCubie
 *          the cubie to which this cubie is compared
 * @return <b>0</b> if this cubie represents the same cubie as
 *         <b>otherCubie</b>; <br>
 *         <b>-1</b> otherwise
 */
public int compareTo(Cubie otherCubie) {
    // Stores true if the current sticker of the current cubie is found on
    // otherCubie; false otherwise.
    boolean found;

    /*
     * i.e. if this instance is a corner, and the other instance is an edge,
     * then they are obviously not the same.
     */
    if (otherCubie.getStickers().length != this.stickers.length)
        return -1;

    for (int i = 0; i < stickers.length; ++i) {
        found = false;
```

```
for (int j = 0; j < stickers.length; ++j) {
    /*
     * Each element of this object's stickers should be in the other
     * object's stickers. If the element is not found in the other
     * array, then the method will return false.
     */
    if (this.getStickers()[i].equals(otherCubie.getStickers()[j])) {
        found = true;
        break;
    }
}
/*
 * This means the one of the elements of this.stickers has not been
 * found in otherCubie.stickers, so their stickers are not the same
 */
if (!found)
    return -1;
}
return 0;
}

/**
 * @param otherCubie
 *          the cubie to which this cubie is compared
 * @return <b>0</b> if the cubies are the same and the stickers are in the
 *         same order (but not necessarily positions, e.g. the
 *         red-green-white corner will be shown to be the same as the
 *         green-white-red corner; <br>
 *         <b>-1</b> otherwise
 */
public int strictCompareTo(Cubie otherCubie) {
    // TODO Test this method to ensure that the new algorithm is valid

    int stickersLength = this.stickers.length;

    if (stickersLength != otherCubie.getStickers().length)
        return -1;

    /*
     * Color[] otherStickersCopy = Arrays.copyOf(otherCubie.getStickers(),
     * otherCubie.getStickers().length); boolean matching; Color current;
     *
     * for (int i = 0; i < length; ++i) { current = stickersCopy[0];
     */
```

```
* matching = true;
*
* for (int j = 0; j < length; ++j) { if
* (!stickersCopy[j].equals(this.stickers[j])) { matching = false;
* break; } }
*
* if (matching) return 0;
*
* for (int j = 0; j < length - 1; ++j) { // Cycle stickers
* stickersCopy[j] = stickersCopy[j + 1]; } stickersCopy[length - 1] =
* current;
*
* } return -1;
*/
int i = 0;

/*
 * Find the index of a matching sticker
 */
while ((i < stickersLength) && (!otherCubie.getStickers()[0].equals(this.getStickers()[i]))) {
    ++i;
}

/*
 * If no matching sticker is found then i == stickersLength, so return
 * -1
 */
if (i == stickersLength) {
    return -1;
}

/*
 * Compare each element until will the expected offset. If any elements
 * differ, then return -1.
 */
for (int j = i + 1; j < i + stickersLength; ++j) {
    if (!stickers[j % stickersLength]
        .equals(otherCubie.getStickers()[(j - i + stickersLength) % stickersLength])) {
        return -1;
    }
}

/*
```

```
        * No contradictions have been found, so return 0
        */
    return 0;
}

/**
 * @param color
 *         the colour to be analysed
 * @return the english word used for the colour; <br>
 *         <b>-1</b> otherwise
 */
public static String getColorToWord(Color color) {
    if (color.equals(Color.white))
        return "white";
    else if (color.equals(Color.yellow))
        return "yellow";
    else if (color.equals(Color.red))
        return "red";
    else if (color.equals(Cubie.orange))
        return "orange";
    else if (color.equals(Color.green))
        return "green";
    else if (color.equals(Color.blue))
        return "blue";

    return "-1";
}

/**
 * @param color
 *         the english word for the colour to be returned
 * @return a Color object with the associated characteristics of
 *         <b>color</b>
 */
public static Color getWordToColor(String color) {
    color = color.toLowerCase().substring(0, 1);

    switch (color) {
    case "w":
        return Color.white;
    case "y":
        return Color.yellow;
    case "r":
        return Color.red;
```

```
    case "o":  
        return Cubie.orange;  
    case "g":  
        return Color.green;  
    case "b":  
        return Color.blue;  
    default:  
        System.err.printf("\ngetWordToColor(String color) \\" -> color = %s%n", color);  
        return Color.black;  
    }  
}  
}
```

## Class CustomPdfWriter

```
package jCube;

import java.io.FileOutputStream;
import java.util.Date;

import javax.swing.JOptionPane;

import com.itextpdf.text.Document;
import com.itextpdf.text.DocumentException;
import com.itextpdf.text.Element;
import com.itextpdf.text.Font;
import com.itextpdf.text.Paragraph;
import com.itextpdf.text.Phrase;
import com.itextpdf.text.pdf.PdfPCell;
import com.itextpdf.text.pdf.PdfPTable;
import com.itextpdf.text.pdf.PdfWriter;

/**
 * @author Kelsey McKenna
 */
public class CustomPdfWriter {
    /**
     * This stores a font for the headings of the table stored when saving
     * statistics.
     */
    private static Font tableHeaderFont = new Font(Font.FontFamily.HELVETICA, 15, Font.BOLD);

    /**
     * Creates the statistics pdf
     *
     * @param filePath
     *         the path to which the pdf file should be saved
     * @param timeData
     *         the times from which the statistics are calculated
     */
    public static void createStatisticsPdf(String filePath, String[] timeData) {
        Document document = new Document();

        try {
            PdfWriter.getInstance(document, new FileOutputStream(filePath));
            document.open();
            addMetaData(document);
        }
```

```
    addTimesPage(document, timeData);
    document.close();
} catch (Exception e) {
    JOptionPane.showMessageDialog(null, "Error Writing to File", "Error", JOptionPane.ERROR_MESSAGE);
} finally {
    try {
        document.close();
    } catch (Exception e) {
    }
}
}

/**
 * Adds meta data to the pdf document
 *
 * @param document
 *         the document to which the meta data is to be added
 */
private static void addMetaData(Document document) {
    document.addTitle("Rubik's Cube Statistics - " + new Date());
    document.addAuthor(System.getProperty("user.name"));
    document.addKeywords("Rubik, Cube, Statistics");
}

/**
 * Adds the specific contents to the document
 *
 * @param document
 *         the document to which the contents are added
 * @param timeData
 *         the data from which the statistics are calculated
 * @throws DocumentException
 *         if there was an error writing to the document
 */
private static void addTimesPage(Document document, String[] timeData) throws DocumentException {
    Paragraph timesChapter = new Paragraph();
    timesChapter.add(new Paragraph(" ")); // Empty line/separator
    timesChapter.add(getStatisticsTable(timeData));
    document.add(timesChapter);
    document.newPage();
}

/**
 * @param timeData
```

```
*           the times from which statistics are calculated
* @return a formatted table containing the statistics
* @throws DocumentException
*           if there was an error writing to the document
*/
private static PdfPTable getStatisticsTable(String[] timeData) throws DocumentException {
    PdfPTable table = new PdfPTable(3);
    int[] widths = { 20, 10, 40 };
    table.setWidthPercentage(100);
    table.setWidths(widths);

    PdfPCell c1 = new PdfPCell(new Phrase("Average Type", tableHeaderFont));
    c1.setHorizontalAlignment(Element.ALIGN_LEFT);
    c1.setVerticalAlignment(Element.ALIGN_CENTER);
    table.addCell(c1);

    c1 = new PdfPCell(new Phrase("Average", tableHeaderFont));
    c1.setHorizontalAlignment(Element.ALIGN_LEFT);
    c1.setVerticalAlignment(Element.ALIGN_CENTER);
    table.addCell(c1);

    c1 = new PdfPCell(new Phrase("Times", tableHeaderFont));
    c1.setHorizontalAlignment(Element.ALIGN_LEFT);
    c1.setVerticalAlignment(Element.ALIGN_CENTER);
    table.addCell(c1);
    table.setHeaderRows(0);

    if (timeData != null) {
        for (int i = 0; i < timeData.length; ++i) {
            c1 = new PdfPCell(new Phrase(timeData[i]));
            c1.setPaddingTop(13F);
            c1.setPaddingBottom(20F);
            table.addCell(c1);
        }
    } else {
        table.addCell("No data to display");
        table.addCell("No data to display");
        table.addCell("No data to display");
    }
    return table;
}
```

## Class Edge

```
package jCube;

import static java.awt.Color.blue;
import static java.awt.Color.green;
import static java.awt.Color.red;
import static java.awt.Color.white;
import static java.awt.Color.yellow;

import java.awt.Color;
import java.util.Arrays;

/**
 * @author Kelsey McKenna
 */
public class Edge extends Cubie {
    /**
     * The initial edges of a solved cube with white on top and green on front.
     * The <b>i</b>th element represents the sticker for the cubie with
     * cubieIndex <b>i</b>
     */
    private static final Color[][] INITIAL_EDGES = { { white, blue }, { white, red }, { white, green },
        { white, orange }, { blue, orange }, { blue, red }, { green, red }, { green, orange }, { yellow, blue },
        { yellow, orange }, { yellow, green }, { yellow, red } };

    /**
     * Constructor - blank
     */
    public Edge() {
    }

    /**
     * Constructor - assigns colors to the stickers of the Cubie
     *
     * @param stickers
     *         the colors to be assigned to the stickers of the Edge
     */
    public Edge(Color[] stickers) {
        super(stickers);
    }

    /**
     * Constructor - assigns colors to the stickers of the Cubie
```

```
* @param one
*          the first sticker colour
* @param two
*          the second sticker colour
*/
public Edge(Color one, Color two) {
    Color[] stickers = { one, two };
    super.setStickers(stickers);
}

/**
 * Constructor - acts as a clone
 *
 * @param edge
 *          the edge to be cloned
 */
public Edge(Edge edge) {
    super(edge.getStickers());
    setOrientation(edge.getOrientation());
}

/**
 * @return the initial edges of a solved cube with white on top and green on
 *         front
 */
public static Color[][] getAllInitialStickers() {
    return Arrays.copyOf(INITIAL_EDGES, 12);
}

/**
 * @param index
 *          the index of the edge whose stickers to be returned
 * @return the <b>index</b>th Edge's stickers
 */
public static Color[] getInitialStickers(int index) {
    return Arrays.copyOf(INITIAL_EDGES[index], 2);
}

/**
 * @param zero
 *          the first sticker of the Edge
 * @param one
 *          the second sticker of the Edge
```

```
/*
public void setStickers(Color zero, Color one) {
    Color[] stickers = { zero, one };
    super.setStickers(stickers);
}

/**
 * Flip the positions of the stickers so that the first sticker in place of
 * the second sticker and vice-versa.
 */
private void flipStickers() {
    Color[] stickersCopy = Arrays.copyOf(super.getStickers(), 2);
    setStickers(stickersCopy[1], stickersCopy[0]);
}

/**
 * Flips the edge so that its appearance (<b>stickers</b>) and properties
 * (<b>orientation</b>) change
 */
public void flip() {
    flipStickers();
    flipOrientation();
}

/**
 * Change the orientation in the following mapping: <br>
 * 0 -> 1 <br>
 * 1 -> 0
 */
public void flipOrientation() {
    super.setOrientation((super.getOrientation() + 1) % 2);
}

/**
 * Returns the secondary colour of an edge, i.e. the colour that is not
 * white or yellow. If the edge does not contain white or yellow, then the
 * first sticker will be returned.
 *
 * @return <b>secondary colour</b> of the edge if the Edge has a yellow or
 *         white sticker; <br>
 *         <b>first sticker</b> otherwise
 */
public Color getSecondaryColor() {
    Color[] stickers = getStickers();
```

```
if ((stickers[0].equals(Color.white)) || (stickers[0].equals(Color.yellow)))
    return stickers[1];
else
    return stickers[0];
}
```

## Class EdgeSolver

```
package jCube;

import java.awt.Color;

/**
 * @author Kelsey McKenna
 */
public class EdgeSolver extends SolveMaster {

    /**
     * @author Kelsey McKenna
     */
    private class SolveCandidate {
        /**
         * This stores the index of the edge on the cube.
         */
        int index = -1000;
        /**
         * This stores the number of moves required to solve the edge.
         */
        int score = -1000;
    }

    /**
     * This stores the Edge objects representing the red-green, green-orange,
     * orange-blue, and blue-red edges.
     */
    private static Edge[] mLEdges = new Edge[4]; // The 'middle-layer' edges
    /**
     * This stores the remaining middle-layer Edge objects that need to be
     * solved.
     */
    private SolveCandidate[] solveCandidates;

    /**
     * Constructor - assigns an object to the parent class's cube field
     *
     * @param cube
     *          the cube to be solved
     */
    public EdgeSolver(Cube cube) {
        super(cube);
```

```
        for (int i = 4; i < 8; ++i) {
            mLEdges[i - 4] = new Edge(Edge.getInitialStickers(i));
        }
    }

/*
 * public void solveMiddleLayerEdges() { Edge edge;
 * rotateToTop(Color.yellow); int numEdgesToSolve = 4; int
 * solveCandidatesIndex; int edgeIndex; Color[] stickers;
 *
 * while (!middleLayerEdgesSolved()) { solveCandidatesIndex = 0;
 * solveCandidates = new SolveCandidate[numEdgesToSolve--];
 *
 * for (int i = 0; i < 8; ++i) { edge = cube.getEdge(i); if (isMLEdge(edge)
 * && (!pieceSolved(edge))) { solveCandidates[solveCandidatesIndex] = new
 * SolveCandidate(); solveCandidates[solveCandidatesIndex].index = i;
 * solveCandidates[solveCandidatesIndex].score = getScore(i);
 * ++solveCandidatesIndex; } }
 *
 * edgeIndex =
 * solveCandidates[getIndexOfMinScore(solveCandidatesIndex)].index; stickers
 * = cube.getEdge(edgeIndex).getStickers(); solutionExplanation +=
 * String.format("Edges - %s-%s edge:\n", Cubie.getColorWord(stickers[0]),
 * Cubie.getColorWord(stickers[1])); solveEdge(edgeIndex); } }
 */

/**
 * Solves the edges in the middle layer, and records the solution and
 * explanation at the same time.
 */
public void solveMiddleLayerEdges() {
    // Stores the properties of the edge to be solved.
    Edge edge;
    rotateToTop(Color.yellow);
    // Stores the index of the unsolved corner being examined.
    int solveCandidatesIndex;
    // Stores the index of the corner to solve.
    int edgeIndex;
    // Stores the stickers of the edge so that a better explanation can be
    // generated.
    Color[] stickers;

    solveCandidates = new SolveCandidate[4];
    for (int i = 0; i < 4; ++i)
```

```
// initialise solveCandidates
solveCandidates[i] = new SolveCandidate();

while (!middleLayerEdgesSolved()) {
    solveCandidatesIndex = 0;

    for (int i = 0; i < 8; ++i) {
        edge = cube.getEdge(i);
        if (isMLEdge(edge) && (!isPieceSolved(edge))) {
            solveCandidates[solveCandidatesIndex].index = i;
            solveCandidates[solveCandidatesIndex].score = getScore(i);
            ++solveCandidatesIndex;
        }
    }

    edgeIndex = solveCandidates[getIndexOfMinScore(solveCandidatesIndex)].index;

    stickers = cube.getEdge(edgeIndex).getStickers();
    solutionExplanation += String.format("Edges - %s-%s edge:\n", Cubie.getColorToWord(stickers[0]),
                                         Cubie.getColorToWord(stickers[1]));

    solveEdge(edgeIndex);
}

try {
    // Remove last newline character
    solutionExplanation = solutionExplanation.substring(0, solutionExplanation.lastIndexOf("\n"));
} catch (IndexOutOfBoundsException e) {
}
}

/**
 * Solves the Edge at the specified index, and record the solution and
 * explanation at the same time.
 *
 * @param currentIndex
 *         the index of the Edge to be solved.
 */
public void solveEdge(int currentIndex) {
    rotateToTop(Color.yellow);
    // Stores a copy of the edge to be solved.
    Edge edge = new Edge(cube.getEdge(currentIndex).getStickers());

    if (isPieceSolved(edge))
```

```
return;

if /* edge is in top layer */(currentIndex < 4) {
    while (!edgeInSetupPosition(edge)) {
        cube.performAbsoluteMoves("U");
        catalogMoves("U");
    }

    while (cube.getEdge(2).compareTo(edge) == -1) {
        cube.rotate("y");
        catalogMoves("y");
    }

    if (edge.getStickers()[0].equals(cube.getSlice(2).getCentre())) {
        cube.performAbsoluteMoves("U R U' R' U' F' U F");
        catalogMoves("U R U' R' U' F' U F");
        solutionExplanation += "The edge needs to go to the right, so set up the edge then perform R U' R' U' F' U F\n\n";
    } else {
        cube.performAbsoluteMoves("U' L' U L U F U' F'");
        catalogMoves("U' L' U L U F U' F'");
        solutionExplanation += "The edge needs to go to the left, so set up the edge then perform L' U L U F U' F'\n\n";
    }
} else /* edge is in E slice/middle layer */{
    // Rotate the cube until the edge is at FR
    while (cube.getEdge(6).compareTo(edge) == -1) {
        cube.rotate("y");
        catalogMoves("y");
    }

    cube.performAbsoluteMoves("R U' R' U' F' U F");
    catalogMoves("R U' R' U' F' U F");
    solutionExplanation += "The edge is trapped in the middle layer, so remove it using R U' R' U' F' U F.\n";

    // The edge is now at UB, so solve the edge at this index
    solveEdge(0);
}

/**
 * @param edge
 *      the edge to be analysed
 * @return <b>true</b> if the specified Edge is in the correct setup
 *         position for solving; <br>
 *         <b>false</b> otherwise
 */
```

```
/*
private boolean edgeInSetupPosition(Edge edge) {
    int index = getIndexOf(edge);

    return (cube.getSlice(sliceEdgeSharing[index][1]).getCentre().equals(edge.getStickers()[1]));
}

/**
 * @param edge
 *          the edge to be analysed
 * @return <b>true</b> if the specified edge belongs in the middle layer,
 *          i.e. it does not have a yellow or white sticker <br>
 *          <b>false</b> otherwise
 */
public static boolean isMLEdge(Edge edge) {
    Color[] stickers = edge.getStickers();

    if ((!stickers[0].equals(Color.white)) && (!stickers[1].equals(Color.white))
        && (!stickers[0].equals(Color.yellow)) && (!stickers[1].equals(Color.yellow)))
        return true;
    else
        return false;
}

/**
 * @return <b>true</b> if the Edges in the middle layer are solved;
 *          <b>false</b> otherwise
 */
public boolean middleLayerEdgesSolved() {
    for (int i = 4; i < 8; ++i)
        if (!isPieceSolved(cube.getEdge(i)))
            return false;

    return true;
}

/*
 * private int getScore(Edge edge) { int score = 0; int index =
 * getIndexOf(edge); int orientation = edge.getOrientation(); int
 * overEdgeIndex = ((getIndexOfDestination(edge) - 4) + ((orientation == 0)
 * ? -2 : 1) + 4) % 4;
 *
 * if (index < 4) score -= getShortestOffset(overEdgeIndex, index); else {
 * score -= 7; if (pieceIsInCorrectPosition(edge) && (orientation == 1))
 * 
```

```
* score -= 4; }
*
* return score; }
*/
/***
 * This determines how many moves are required to solve the edge at the
 * specified index.
 *
 * @param index
 *          the index of Edge to be analysed
 * @return the number of moves required to solve the Edge at the specified
 *         <b>index</b>
 */
private int getScore(int index) {
/*
 * This determines the number of moves to solve the edge at the
 * specified index by solving the piece, counting the moves, then
 * reverses the moves so that the cube is in its initial state before
 * the method was invoked.
 */
    // Stores the number of moves to solve the edge.
    int score;

    EdgeSolver es = new EdgeSolver(cube);
    es.solveEdge(index);
    simplifyMoves(es.getCatalogMoves(), SolveMaster.CORNER_EDGE);
    score = es.getCatalogMoves().size();
    cube.performAbsoluteMoves(getReverseStringMoves(es.getCatalogMoves()));
    es.clearMoves();

    return score;
}

/***
 * Returns the index of the element in the <b>solveCandidates</b> array that
 * requires the fewest number of moves to solve
 *
 * @param length
 *          the number of remaining unsolved edges
 * @return the index of the element in <b>solveCandidates</b> that requires
 *         the fewest number of moves to solve
 */
*/
```

```
private int getIndexOfMinScore(int length) {  
    int min = solveCandidates[0].score;  
    int indexOfMin = 0;  
  
    for (int i = 1; i < length; ++i) {  
        if (solveCandidates[i].score < min) {  
            indexOfMin = i;  
            min = solveCandidates[i].score;  
        }  
    }  
  
    return indexOfMin;  
}  
}
```

## Class LinearSearch

```
package jCube;

import java.awt.Color;
import java.util.LinkedList;

/**
 * @author Kelsey McKenna
 */
public class LinearSearch {

    /**
     * Suppresses default constructor, ensuring non-instantiability.
     */
    private LinearSearch() {
    }

    /**
     * Searches an array of Colors for a specified Color
     *
     * @param list
     *         the list to be searched
     * @param element
     *         the element to be found
     * @return the index of <b>element</b> in <b>list</b>. <br>
     *         If the index is 2, then <b>-1</b> is returned. <br>
     *         If the element cannot be found then <b>-2</b> is returned.
     */
    public static int linearSearchCornerOrientation(Color[] list, Color element) {
        for (int i = 0; i < list.length; ++i) {
            if (list[i].equals(element))
                return (i == 2) ? -1 : i;
        }
        return -2; // Not found
    }

    /**
     * Searches an array of Colors for a specified Color
     *
     * @param list
     *         the list to be searched
     * @param element
     *         the element to be found
```

```
* @return the index of <b>element</b> in <b>list</b> if the element can be
*         found; <br>
*         <b>-1</b> otherwise
*/
public static int linearSearch(Color[] list, Color element) {
    for (int i = 0; i < list.length; ++i) {
        if (list[i].equals(element))
            return i;
    }

    return -1; // Not found
}

/**
 * Searches an array of Strings for a specified String
 *
 * @param list
 *         the list to be searched
 * @param element
 *         the element to be found
 * @return the index of <b>element</b> in <b>list</b> if the element can be
 *         found; <br>
 *         <b>-1</b> otherwise
 */
public static int linearSearch(String[] list, String element) {
    for (int i = 0; i < list.length; ++i) {
        if (list[i].equals(element))
            return i;
    }

    return -1; // Not found
}

/**
 * Searches a linked list of strings for a specified element
 *
 * @param list
 *         the list to be searched
 * @param element
 *         the element to be found
 * @return the index of <b>element</b> in <b>list</b>
 */
public static int linearSearch(LinkedList<String> list, String element) {
    for (int i = 0; i < list.size(); ++i) {
        if (list.get(i).equals(element))
```

```
        return i;
    }
    return -1; // Not found
}

/**
 * Searches an array of Strings for a String that starts with the specified
 * element, e.g. "Hello" starts with "He"
 *
 * @param list
 *          the list to be searched
 * @param element
 *          the element to be found
 * @return the index of <b>element</b> in <b>list</b>
 */
public static int linearSearchStartsWith(String[] list, String element) {
    for (int i = 0; i < list.length; ++i) {
        if (list[i].startsWith(element))
            return i;
    }
    return -1; // Not found
}

/**
 * Searches an array of Integers to see if it contains a specified element
 * (index is not important)
 *
 * @param list
 *          the list to be searched
 * @param element
 *          to be found
 * @return <b>true</b> if the list contains the specified element; <br>
 *         <b>false</b> otherwise
 */
public static boolean linearSearchContains(Integer[] list, Integer element) {
    for (int i = 0; i < list.length; ++i) {
        if (list[i].equals(element))
            return true;
    }
    return false;
}
```

## Class Main

```
package jCube;

import java.awt.BasicStroke;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Component;
import java.awt.Cursor;
import java.awt.Desktop;
import java.awt.Dimension;
import java.awt.Event;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GridLayout;
import java.awt.Image;
import java.awt.Point;
import java.awt.RenderingHints;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import java.awt.geom.Rectangle2D;
import java.io.File;
import java.io.IOException;
import java.util.LinkedList;

import javax.imageio.ImageIO;
import javax.swing.BorderFactory;
import javax.swing.BoxLayout;
import javax.swing.DefaultListModel;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JMenu;
```

```
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextArea;
import javax.swing.KeyStroke;
import javax.swing.ListSelectionModel;
import javax.swing.SwingUtilities;
import javax.swing.Timer;
import javax.swing.border.EtchedBorder;
import javax.swing.table.TableCellRenderer;
import javax.swing.table.TableColumnModel;

import org.jfree.data.category.DefaultCategoryDataset;

/**
 * @author Kelsey McKenna
 */
public class Main extends JPanel implements KeyListener {

    /**
     * @author Kelsey McKenna This is used to run the visible incrementing timer
     *         in the main window
     */
    private static class TimerListener implements ActionListener {
        /**
         * @see java.awt.event.ActionListener#actionPerformed(java.awt.event.ActionEvent)
         */
        public void actionPerformed(ActionEvent evt) {
            /*
             * This routine will be called every x milliseconds (defined by
             * Timer constructor) So, we want to increase elapsedTime
             * accordingly and change the text of timeLabel to show elapsedTime
             * to 2 decimal places
             */
            elapsedTime += timingDisplayInterval;
            if (elapsedTime >= 60.0) {
                ++elapsedMinutes;
                elapsedTime = 0;
            }
            if (elapsedMinutes > 0)
```

```
timeLabel.setText(String.format("%d:%02d.%02d", elapsedMinutes, (int) elapsedTimingSeconds,
                               (int) (100 * (elapsedTimingSeconds - (int) elapsedTimingSeconds))));
else
    timeLabel.setText(String.format("%d.%02d", (int) elapsedTimingSeconds,
                                   (int) (100 * (elapsedTimingSeconds - (int) elapsedTimingSeconds))));

}
}

/**
 * @author Kelsey McKenna This is used to run the inspection timer in the
 *         main window
 */
private static class InspectionTimerListener implements ActionListener {

    /**
     * @see java.awt.event.ActionListener#actionPerformed(java.awt.event.ActionEvent)
     */
    public void actionPerformed(ActionEvent e) {
        inspectionTimeRemaining -= 1;

        if (inspectionTimeRemaining > 0)
            timeLabel.setText("" + inspectionTimeRemaining);
        else if (inspectionTimeRemaining > -2) {
            timeLabel.setText("+2");
            currentPenalty = "+2";
        } else {
            timeLabel.setText("DNF");
            currentPenalty = "DNF";
            inspectionTimer.stop();
            inspectionTimer = null;
        }
    }
}

/**
 * @author Kelsey McKenna This is used to perform real-time animations
 */
private static class RealTimeTimerListener implements ActionListener {
    /**
     * Stores the move count of the solution
     */
    private static int moveCount;
    /**
```

```
* The index of the current move being applied
*/
private static int index;
/**
 * Stores the moves to be performed
 */
private static LinkedList<String> movesToPerform;

/**
 * @param moves
 *      the moves to perform
 */
public RealTimeTimerListener(LinkedList<String> moves) {
    moveCount = moves.size();
    movesToPerform = new LinkedList<>();

    for (int i = 0; i < moveCount; ++i) {
        movesToPerform.add(moves.get(i));
    }

    index = 0;
}

/**
 * @see java.awt.event.ActionListener#actionPerformed(java.awt.event.ActionEvent)
 */
public void actionPerformed(ActionEvent e) {
    if (index < moveCount) {
        cube.performAbsoluteMoves(movesToPerform.get(index));
        ++index;
        cubePanel.repaint();
    } else {
        try {
            realTimeSolutionTimer.stop();
            realTimeSolutionTimer = null;
        } catch (Exception exc) {
        }
        /*
         * MenuBar.setRandomScrambleEnabled(true); boolean enabled =
         * !customPaintingInProgress && !tutorialIsRunning;
         * MenuBar.setRandomScrambleEnabled(enabled);
         * MenuBar.clickToSolveItem.setEnabled(!tutorialIsRunning);
         * MenuBar.paintCustomStateItem.setEnabled(!tutorialIsRunning);
         * MenuBar.solveCubeItem.setEnabled(!tutorialIsRunning);
        */
    }
}
```

```
        * timingDetailsStartNewSolveButton.setEnabled(true);
        */
    MenuBar.clickToSolveItem.setSelected(false);
    MenuBar.clearStickersItem.setEnabled(customPaintingInProgress);
    movesToPerform.clear();
    movesAllowed = !customPaintingInProgress;
}
}

/**
 * @author Kelsey McKenna This WindowListener is used for most windows so
 *         that the all windows are updated in the appropriate manner.
 */
public static class PopUpWindowListener implements WindowListener {
    /**
     * @see java.awt.event.WindowListener#windowActivated(java.awt.event.WindowEvent)
     */
    @Override
    public void windowActivated(WindowEvent arg0) {
        /*
         * try { MouseListener[] m = cubePanel.getMouseListeners(); for (int
         * i = 0; i < m.length; ++i)
         * cubePanel.removeMouseListener(cubePanel.getMouseListeners()[i]);
         * } catch (Exception e) { //All mouse listeners are removed }
         */
    }

    /**
     * Updates the scramble label and gives the cube panel back the
     * appropriate MouseListeners
     *
     * @see java.awt.event.WindowListener#windowClosed(java.awt.event.WindowEvent)
     */
    @Override
    public void windowClosed(WindowEvent arg0) {
        scrambleLabel.setFont(new Font("Courier New", 0, preferencesPopUp.getScrambleFontSize()));
        cubePanel.addMouseListener(getCubePanelMouseListener());
    }

    /**
     * @see java.awt.event.WindowListener#windowClosing(java.awt.event.WindowEvent)
     */
    @Override
```

```
public void windowClosing(WindowEvent arg0) {  
}  
  
/**  
 * Updates the times and statistics  
 *  
 * @see java.awt.event.WindowListener#windowDeactivated(java.awt.event.WindowEvent)  
 */  
@Override  
public void windowDeactivated(WindowEvent arg0) {  
    cubePanel.addMouseListener(getCubePanelMouseListener());  
    copyAllTimesToDisplay();  
    refreshTimeList();  
    refreshTimeGraph(false);  
}  
  
/**  
 * @see java.awt.event.WindowListener#windowDeiconified(java.awt.event.WindowEvent)  
 */  
@Override  
public void windowDeiconified(WindowEvent arg0) {  
}  
  
/**  
 * @see java.awt.event.WindowListener#windowIconified(java.awt.event.WindowEvent)  
 */  
@Override  
public void windowIconified(WindowEvent arg0) {  
}  
  
/**  
 * @see java.awt.event.WindowListener#windowOpened(java.awt.event.WindowEvent)  
 */  
@Override  
public void windowOpened(WindowEvent arg0) {  
    /*  
     * try { MouseListener[] m = cubePanel.getMouseListeners(); for (int  
     * i = 0; i < m.length; ++i)  
     * cubePanel.removeMouseListener(cubePanel.getMouseListeners()[i]);  
     * } catch (Exception e) { //All mouse listeners are removed }  
     */  
}  
}
```

```
/**  
 * @author Kelsey McKenna  
 */  
private static class MenuBar {  
    /**  
     * Stores the contents of the menu bar  
     */  
    private static JMenuBar menuBar;  
    /**  
     * Stores the contents of the file menu  
     */  
    private static JMenu fileMenu;  
    /**  
     * Stores the contents of the edit menu  
     */  
    private static JMenu editMenu;  
    /**  
     * Stores the contents of the tools menu  
     */  
    private static JMenu toolMenu;  
    /**  
     * Stores the contents of the view menu  
     */  
    private static JMenu viewMenu;  
    /**  
     * Stores the contents of the options menu  
     */  
    private static JMenu optionsMenu;  
    /**  
     * Stores the contents of the tutorial menu  
     */  
    private static JMenu tutorialMenu;  
    /**  
     * Stores the contents of the help menu  
     */  
    private static JMenu helpMenu;  
  
    // File  
    /**  
     * Clicking this menu item opens a save dialog, allowing a location to  
     * be chosen to save a text file containing a state-string for the  
     * current state.  
     */  
    private static JMenuItem saveCubeStateItem;
```

```
/**  
 * Clicking this menu item opens a load dialog, allowing a location to  
 * be chosen to load a text file containing a state-string for the  
 * current state.  
 */  
private static JMenuItem loadCubeStateItem;  
/**  
 * Clicking this menu item opens a load dialog so that solve information  
 * can be loaded from a specified location.  
 */  
private static JMenuItem loadSolveInfoItem;  
/**  
 * Clicking this menu item opens a save dialog so that the statistics  
 * can be saved as a pdf to a specified location.  
 */  
private static JMenuItem saveStatsToFileItem;  
/**  
 * Clicking this menu item closes the system  
 */  
private static JMenuItem exitProgramItem;  
/**  
 * Clicking this menu item saves the times in the list, i.e. the  
 * session, in the main window to the database.  
 */  
private static JMenuItem saveSessionToDatabaseItem;  
  
// Edit  
/**  
 * Clicking this menu item opens the Solve Editor window  
 */  
private static JMenuItem manuallyEnterSolveInfoItem;  
/**  
 * Clicking this menu item opens the Solve Editor window with the  
 * selected solves information in its fields  
 */  
private static JMenuItem editSelectedSolveItem;  
/**  
 * Clicking this menu item deletes the selected solve from the list  
 */  
private static JMenuItem deleteSelectedSolveItem;  
  
// View  
/**  
 * Clicking this menu item shows the Time Graph window
```

```
/*
private static JMenuItem showTimeGraphItem;
/**
 * Clicking this menu item shows the Scramble List window
 */
private static JMenuItem showScrambleListItem;
/**
 * Clicking this menu item toggles whether the current scramble is shown
 * at the top of the main window
 */
private static JCheckBoxMenuItem showScrambleItem;
/**
 * Clicking this menu item shows the Algorithm Table window
 */
private static JMenuItem showAlgorithmTableItem;
/**
 * Clicking this menu item shows the Solve Table window
 */
private static JMenuItem showSolveTableItem;
/**
 * Clicking this menu item shows the statistics at the bottom of the
 * main window
 */
private static JMenuItem showStatisticsItem;
/**
 * Clicking this menu item shows the Competition Table window
 */
private static JMenuItem showCompetitionTableItem;
/**
 * Clicking this menu item shows the Member Table window
 */
private static JMenuItem showMemberTableItem;

// Options
/**
 * Clicking this menu item opens the Preferences window
 */
private static JMenuItem preferencesItem;
/**
 * Clicking this menu item toggles whether or not the scrambles in the
 * Scramble List window are used to scramble the cube for each solve
 */
private static JCheckBoxMenuItem useScramblesInListItem;
```

```
// Tools
/**
 * Clicking this menu item cancels all timers, animations etc.
 */
private static JMenuItem cancelSolveItem;
/**
 * Clicking this menu item clears all colours on the cube so that all
 * stickers show grey.
 */
private static JMenuItem clearStickersItem;
/**
 * Clicking this menu item allows a custom state to be painted on cube.
 */
private static JCheckBoxMenuItem paintCustomStateItem;
/**
 * Clicking this menu item generates a solution for the cube, displays
 * this solution, and starts solving the cube in real time
 */
private static JMenuItem solveCubeItem;
/**
 * Clicking this menu item allows the user to click on a piece on the
 * cube in order to generate a solution for that piece.
 */
private static JCheckBoxMenuItem clickToSolveItem;

// Scramble
/**
 * Clicking this menu item generates a random scramble and applies it to
 * the cube, showing the scramble in the main window
 */
private static JMenuItem applyRandomScrambleItem;

// Tutorial
/**
 * Clicking this menu item opens the controls tutorial
 */
private static JMenuItem controlsTutorialItem;
/**
 * This menu holds the cross tutorials
 */
private static JMenu crossTutorialMenu;
/**
 * Clicking this menu item opens the first cross tutorial
 */
```

```
private static JMenuItem crossTutorialOneItem;
/**
 * Clicking this menu item opens the second cross tutorial tutorial
 */
private static JMenuItem crossTutorialTwoItem;
/**
 * This menu holds the corner tutorials
 */
private static JMenu cornerTutorialMenu;
/**
 * Clicking this menu item opens the corner tutorial
 */
private static JMenuItem cornerTutorialOneItem;
/**
 * This menu holds the edge tutorials
 */
private static JMenu edgeTutorialMenu;
/**
 * Clicking this menu item opens the edge tutorial
 */
private static JMenuItem edgeTutorialOneItem;
/**
 * This menu holds the orientation tutorials
 */
private static JMenu orientationTutorialMenu;
/**
 * Clicking this menu item opens the orientation tutorial
 */
private static JMenuItem orientationTutorialOneItem;
/**
 * This menu holds the permutation tutorials
 */
private static JMenu permutationTutorialMenu;
/**
 * Clicking this menu item opens the permutation tutorial
 */
private static JMenuItem permutationTutorialOneItem;
/**
 * Clicking this menu item opens a load dialog so that a tutorial can be
 * loaded from the specified location
 */
private static JMenuItem loadTutorialFromFileItem;
/**
 * Clicking this menu item switches from tutorial mode to timing mode
```

```
/*
private static JMenuItem exitTutorialItem;

// Help
/**
 * Clicking this menu item opens the Terminology.pdf
 */
private static JMenuItem terminologyItem;

/**
 * This variable is used to select a location to save or load text
 * files.
 */
private JFileChooser fileChooser = new JFileChooser(System.getProperty("user.home") + "/Desktop") {
    private static final long serialVersionUID = 1L;

    @Override
    public void approveSelection() {
        // String filePath = getSelectedFile() + ".txt";
        // filePath = filePath.substring(0, filePath.indexOf(".txt")) +
        // ".txt";
        setSelectedFile(new File((( "" + getSelectedFile().replace("\\\\", "\\").replace("\\\\", "\\") + ".txt")));
        File f = new File(getSelectedFile().toString());

        if ((f.exists()) && (getDialogType() == SAVE_DIALOG)) {
            int result = JOptionPane.showConfirmDialog(this, String.format(
                "The file %s already exists. Do you want to overwrite?", getSelectedFile().toString()),
                "Existing File", JOptionPane.YES_NO_OPTION);
            switch (result) {
                case JOptionPane.YES_OPTION:
                    super.approveSelection();
                    return;
                case JOptionPane.NO_OPTION:
                    return;
                case JOptionPane.CLOSED_OPTION:
                    return;
            }
        }
        super.approveSelection();
    }
};

/**
 * This variable is used to select a location to save or load pdf files
```

```
* (statistics)
*/
private JFileChooser pdfFileChooser = new JFileChooser(System.getProperty("user.home") + "/Desktop") {
    private static final long serialVersionUID = 1L;

    @Override
    public void approveSelection() {
        // String filePath = getSelectedFile() + ".txt";
        // filePath = filePath.substring(0, filePath.indexOf(".txt")) +
        // ".txt";
        setSelectedFile(new File(((("") + getSelectedFile()).replaceAll("\\\\pdf", "")) + ".pdf"));
        File f = new File(getSelectedFile().toString());

        if ((f.exists()) && (getDialogType() == SAVE_DIALOG)) {
            int result = JOptionPane.showConfirmDialog(this, String.format(
                "The file %s already exists. Do you want to overwrite?", getSelectedFile().toString()),
                "Existing File", JOptionPane.YES_NO_OPTION);
            switch (result) {
                case JOptionPane.YES_OPTION:
                    super.approveSelection();
                    return;
                case JOptionPane.NO_OPTION:
                    return;
                case JOptionPane.CLOSED_OPTION:
                    return;
            }
        }
        super.approveSelection();
    }
};

/**
 * @return the menu bar for the main window
 */
public JMenuBar createMenuBar() {
    menuBar = new JMenuBar();
    ImageIcon icon;

    /***** FILE *****/
    fileMenu = new JMenu("File");

    icon = createImageIcon("res/images/SaveIcon.png");
    saveCubeStateItem = new JMenuItem("Save Cube-State - txt", icon);
    saveCubeStateItem.addActionListener(new ActionListener() {
```

```
public void actionPerformed(ActionEvent e) {
    if (!isValidCubeState(true)) {
        cubePanel.repaint();
        JOptionPane.showMessageDialog(totalFrame, "This is not a valid state", "Error",
            JOptionPane.ERROR_MESSAGE);
    } else {
        String cubeStateString = solveMaster.getStateString();

        if (fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
            TextFile currentFile = new TextFile();

            try {
                currentFile.setFilePath(fileChooser.getSelectedFile().toString());
                currentFile.setIO(TextFile.WRITE);
                currentFile.write(cubeStateString);
            } catch (Exception exc) {
                JOptionPane.showMessageDialog(totalFrame, "Could not save to file", "Error",
                    JOptionPane.ERROR_MESSAGE);
            } finally {
                currentFile.close();
            }
        }
    }
    cubePanel.repaint();
}
});

icon = createImageIcon("res/images/OpenIcon.png");
loadCubeStateItem = new JMenuItem("Load Cube-State - txt", icon);
loadCubeStateItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Check that nothing else is running

        if (fileChooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            TextFile currentFile = new TextFile();
            String cubeStateString;

            try {
                currentFile.setFilePath(fileChooser.getSelectedFile().toString());
                currentFile.setIO(TextFile.READ);
                cubeStateString = currentFile.readLine();
            }
            try {
```

```
        solveMaster.applyStateString(cubeStateString);
    } catch (Exception exc) {
        JOptionPane.showMessageDialog(totalFrame, "Invalid File", "Error",
            JOptionPane.ERROR_MESSAGE);
    }

    if (!isValidCubeState(false)) {
        resetCube();
        JOptionPane.showMessageDialog(totalFrame, "Invalid Cube-State", "Error",
            JOptionPane.ERROR_MESSAGE);
    }
} catch (Exception exc) {
    JOptionPane.showMessageDialog(totalFrame, "Could not open file", "Error",
        JOptionPane.ERROR_MESSAGE);
} finally {
    currentFile.close();
}

cubePanel.repaint();
cubePanel.requestFocus();
}
}
});

loadCubeStateItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O, Event.CTRL_MASK));

icon = createImageIcon("res/images/OpenIcon.png");
loadSolveInfoItem = new JMenuItem("Load Solve Information - txt", icon);
loadSolveInfoItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        cancelSolve();
        if (fileChooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            TextFile currentFile = new TextFile();
            String[] data;
            int numLines;

            try {
                currentFile.setFilePath(fileChooser.getSelectedFile().toString());
                currentFile.setIO(TextFile.READ);
                numLines = currentFile.getNumLines();

                if (numLines != 5) {
                    throw new Exception();
                }
            }
        }
    }
});
```

```
        data = new String[numLines];

        for (int i = 0; i < numLines; ++i)
            data[i] = currentFile.readLine();

        currentFile.close();

        solves.add(new Solve(data[0], data[1], data[2], data[3], data[4]));
        copyAllTimesToDisplay();
        listHolder.setSelectedIndex(solves.size() - 1);
    } catch (Exception exc) {
        JOptionPane.showMessageDialog(totalFrame, "Invalid File", "Error",
            JOptionPane.ERROR_MESSAGE);
    }
}

refreshStatistics();
});
});

icon = createImageIcon("res/images/SaveIcon.png");
saveStatsToFileItem = new JMenuItem("Save Statistics - pdf", icon);
saveStatsToFileItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S, Event.CTRL_MASK));
saveStatsToFileItem.addActionListener(new ActionListener() {

    @Override
    public void actionPerformed(ActionEvent arg0) {
        if (pdfFileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
            System.out.println(solves.size());
            CustomPdfWriter.createStatisticsPdf(pdfFileChooser.getSelectedFile().toString(),
                Statistics.getFormattedStatisticsArray(solves));
        }
    }
});

icon = createImageIcon("res/images/SaveIcon.png");
saveSessionToDatabaseItem = new JMenuItem("Save Session to Database", icon);
saveSessionToDatabaseItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String dateAdded = SolveDatabaseConnection.getCurrentTimeInSQLFormat();
        Solve currentSolve;

        totalFrame.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    }
});
```

```
try {
    int size = solves.size();
    for (int i = 0; i < size; ++i) {
        currentSolve = solves.get(i);

        SolveDatabaseConnection.executeUpdate(String.format(
            "INSERT INTO solve(solveTime, penalty, comment, scramble, solution, dateAdded) "
            + "VALUES ('%s', '%s', '%s', '%s', '%s', '%s')",
            currentSolve.getStringTime(), currentSolve.getPenalty(), currentSolve.getComment(),
            currentSolve.getScramble(), currentSolve.getSolution(), dateAdded));
    }

    solveDatabasePopUp.populateCellsWithDatabaseData();
} catch (Exception exc) {
    JOptionPane.showMessageDialog(totalFrame, "Could not access database", "Error",
        JOptionPane.ERROR_MESSAGE);
}

totalFrame.setCursor(Cursor.getDefaultCursor());
});

exitProgramItem = new JMenuItem("Exit");
exitProgramItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});

fileMenu.add(saveCubeStateItem);
fileMenu.add(saveStatsToFileItem);
fileMenu.add(saveSessionToDatabaseItem);
fileMenu.add(loadCubeStateItem);
fileMenu.add(loadSolveInfoItem);
fileMenu.addSeparator();
fileMenu.add(exitProgramItem);
/********** END FILE *****/

/********** EDIT *****/
editMenu = new JMenu("Edit");

manuallyEnterSolveInfoItem = new JMenuItem("Add Solve");
manuallyEnterSolveInfoItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_D, Event.CTRL_MASK));
```

```
manuallyEnterSolveInfoItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        solves.add(new Solve("DNF", "0", "*"));
        copyAllTimesToDisplay();
        listHolder.setSelectedIndex(solves.size() - 1);
        timeListPopUp.setSolve(solves.get(listHolder.getSelectedIndex()));
        timeListPopUp.selectAllTimeText();
        timeListPopUp.setVisible(true);
    }
});

editSelectedSolveItem = new JMenuItem("Edit Selected Solve");
editSelectedSolveItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_E, Event.CTRL_MASK));
editSelectedSolveItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        int selectedIndex = listHolder.getSelectedIndex();

        if (selectedIndex == -1) {
            JOptionPane.showMessageDialog(totalFrame, "No Solve Selected", "Error",
                JOptionPane.ERROR_MESSAGE);
            return;
        }

        timeListPopUp.setSolve(solves.get(selectedIndex));
        timeListPopUp.selectAllTimeText();
        timeListPopUp.setVisible(true);
    }
});

deleteSelectedSolveItem = new JMenuItem("Delete Selected Solve");
deleteSelectedSolveItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        int selectedIndex = listHolder.getSelectedIndex();

        if (selectedIndex == -1) {
            JOptionPane.showMessageDialog(totalFrame, "No Solve Selected", "Error",
                JOptionPane.ERROR_MESSAGE);
            return;
        }
    }
});
```

```
solves.get(selectedIndex).setStringTime("-1");
refreshTimeList();
refreshStatistics();
copyAllTimesToDisplay();
refreshTimeGraph(true);

if (solves.size() > 0)
    listHolder.setSelectedIndex((selectedIndex < solves.size()) ? selectedIndex : selectedIndex - 1);

listHolder.requestFocus();
}
});

editMenu.add(manuallyEnterSolveInfoItem);
editMenu.add(editSelectedSolveItem);
editMenu.add(deleteSelectedSolveItem);
***** END EDIT *****

***** VIEW *****
viewMenu = new JMenu("View");

showTimeGraphItem = new JMenuItem("Show Time Graph");
showTimeGraphItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        timeGraph.setDataset(getDatasetOfSelectedTimes());
        timeGraph.draw();
        timeGraph.setVisible(true);
    }
});

showScrambleItem = new JCheckBoxMenuItem("Show Current Scramble");
showScrambleItem.setSelected(true);
showScrambleItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        scrambleLabel.setVisible(showScrambleItem.isSelected());
    }
});

showScrambleListItem = new JMenuItem("Show Scramble List");
showScrambleListItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        scramblePopUp.setVisible(true);
    }
});
```

```
});  
  
showAlgorithmTableItem = new JMenuItem("Show Algorithm Table");  
showAlgorithmTableItem.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        algorithmDatabasePopUp.setVisible(true);  
    }  
});  
  
showSolveTableItem = new JMenuItem("Show Solve Table");  
showSolveTableItem.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        solveDatabasePopUp.setVisible(true);  
    }  
});  
  
showStatisticsItem = new JMenuItem("Show Statistics");  
showStatisticsItem.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent arg0) {  
        refreshStatistics();  
    }  
});  
  
showCompetitionTableItem = new JMenuItem("Show Competition Table");  
showCompetitionTableItem.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent arg0) {  
        competitionDatabasePopUp.setVisible(true);  
    }  
});  
  
showMemberTableItem = new JMenuItem("Show Member Table");  
showMemberTableItem.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        memberDatabasePopUp.setVisible(true);  
    }  
});  
  
viewMenu.add(showTimeGraphItem);  
viewMenu.add(showScrambleItem);  
viewMenu.add(showScrambleListItem);  
viewMenu.add(showAlgorithmTableItem);
```

```
viewMenu.add(showSolveTableItem);
viewMenu.add(showStatisticsItem);
viewMenu.addSeparator();
viewMenu.add(showCompetitionTableItem);
viewMenu.add(showMemberTableItem);
/**************** END VIEW *****/

/**************** TOOLS *****/
toolMenu = new JMenu("Tools");

cancelSolveItem = new JMenuItem("Cancel Solve");
cancelSolveItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        Main.cancelSolve();
    }
});

cancelSolveItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Q, Event.CTRL_MASK));

paintCustomStateItem = new JCheckBoxMenuItem("Paint Custom State");
paintCustomStateItem.setSelected(false);
paintCustomStateItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        boolean isSelected = paintCustomStateItem.isSelected();

        if ((tutorialIsRunning) || ((inspectionTimer != null)) || (incTimer != null)
            || (realTimeSolutionTimer != null) || (clickToSolve)) {
            cubePanel.requestFocus();
            paintCustomStateItem.setSelected(false);
            colorSelection.setVisible(false);
            return;
        }
        if (!isSelected && !isValidCubeState(true)) {
            int choice = MouseSelectionSolver
                .getQuestionDialogResponse("This is not a valid state. Would you like to reset the cube?");
            if (choice == 0)
                resetCube();
            else {
                paintCustomStateItem.setSelected(true);
                colorSelection.setVisible(true);
                cubePanel.repaint();
                cubePanel.requestFocus();
                return;
            }
        }
    }
});
```

```
        }
    }
    customPaintingInProgress = !customPaintingInProgress;
    timerHasPermissionToStart = !timerHasPermissionToStart;
    movesAllowed = !isSelected;
    if ((realTimeSolutionTimer != null) && (realTimeSolutionTimer.isRunning()))
        movesAllowed = false;

    colorSelection.setVisible(isSelected);
    /*
     * applyRandomScrambleItem.setEnabled(!isSelected);
     * timingDetailsStartNewSolveButton.setEnabled(!isSelected
     * && (!clickToSolveItem.isSelected()));
     */
    clearStickersItem.setEnabled(customPaintingInProgress);
    cubePanel.repaint();
    cubePanel.requestFocus();
}
});

clearStickersItem = new JMenuItem("Clear Stickers");
clearStickersItem.setEnabled(false);
clearStickersItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (customPaintingInProgress && (realTimeSolutionTimer == null)) {
            for (int i = 0; i < 8; ++i)
                cube.getCorner(i).setStickers(Color.LIGHT_GRAY, Color.LIGHT_GRAY, Color.LIGHT_GRAY);
            for (int i = 0; i < 12; ++i)
                cube.getEdge(i).setStickers(Color.LIGHT_GRAY, Color.LIGHT_GRAY);
        }

        cubePanel.repaint();
        cubePanel.requestFocus();
    }
});
solveCubeItem = new JMenuItem("Solve Cube");
solveCubeItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (customPaintingInProgress) {
            solveCubeItem.setSelected(false);
            return;
        }
    }
});
```

```
cancelSolve();
clickToSolveItem.setSelected(false);
clickToSolve = false;

if (!isValidCubeState(true)) {
    cubePanel.repaint();
    // colorSelection.setAlwaysOnTop(false);
    JOptionPane.showMessageDialog(colorSelection,
        "Some of the stickers are incorrect. Please try again.", "Error",
        JOptionPane.ERROR_MESSAGE);
    // colorSelection.setAlwaysOnTop(true);

    cubePanel.requestFocus();
} else {
    /*
     * applyRandomScrambleItem.setEnabled(false);
     * clickToSolveItem.setEnabled(false);
     * clearStickersItem.setEnabled(false);
     * paintCustomStateItem.setEnabled(false);
     * solveCubeItem.setEnabled(false);
     */
    timingDetailsTextArea.setText(getFormattedCubeSolution());
    timingDetailsTextArea.setCaretPosition(0);
    performRealTimeSolving();
}

cubePanel.repaint();
cubePanel.requestFocus();
}
);
solveCubeItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_R, Event.CTRL_MASK)); // delete
// for
// final
// program

clickToSolveItem = new JCheckBoxMenuItem("Solve Piece");
clickToSolveItem.setSelected(false);
clickToSolveItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        if ((customPaintingInProgress) || (realTimeSolutionTimer != null) || (inspectionTimer != null)
            || (incTimer != null)) {
            clickToSolveItem.setSelected(false);
            return;
        }
    }
});
```

```
        }

        clickToSolve = clickToSolveItem.isSelected();

        if (clickToSolveItem.isSelected()) {
            cancelSolve();

            if (!isValidCubeState(true)) {
                cubePanel.repaint();
                // colorSelection.setAlwaysOnTop(false);
                JOptionPane.showMessageDialog(colorSelection,
                    "Some of the stickers are incorrect. Please try again.", "Error",
                    JOptionPane.ERROR_MESSAGE);
                // colorSelection.setAlwaysOnTop(true);
                cubePanel.requestFocus();
                clickToSolveItem.setSelected(false);
                clickToSolve = false;
                return;
            }
        }

        if (preferencesPopUp.solvePieceWarningEnabled()) {
            // colorSelection.setAlwaysOnTop(false);
            JOptionPane.showMessageDialog(colorSelection, "Click on a piece to view solution",
                "Click to Solve", JOptionPane.INFORMATION_MESSAGE);
            // colorSelection.setAlwaysOnTop(true);
        }
        // timingDetailsStartNewSolveButton.setEnabled(false);
    } else {
        // timingDetailsStartNewSolveButton.setEnabled(!paintCustomStateItem.isSelected());
    }
}
});

applyRandomScrambleItem = new JMenuItem("Apply Random Scramble");
applyRandomScrambleItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        if ((inspectionTimer == null) && (realTimeSolutionTimer == null) && (inctimer == null)
            && (!customPaintingInProgress)) {
            currentScramble = Scramble.generateScramble();
            cube.performAbsoluteMoves(currentScramble);
            scrambleLabel.setText("Scramble: " + currentScramble);
            cubePanel.repaint();
            cubePanel.requestFocus();
        }
    }
});
```

```
        }
    });
}

toolMenu.add(cancelSolveItem);
toolMenu.add(clearStickersItem);
toolMenu.add(paintCustomStateItem);
toolMenu.add(solveCubeItem);
toolMenu.add(clickToSolveItem);
toolMenu.add(applyRandomScrambleItem);
/********** END TOOLS *****/

/********** OPTIONS *****/
optionsMenu = new JMenu("Options");

preferencesItem = new JMenuItem("Preferences");
preferencesItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        // preferencesPopUp.populateFieldsWithValues();
        preferencesPopUp.setVisible(true);
    }
});

useScramblesInListItem = new JCheckBoxMenuItem("Use Scrambles in List");
useScramblesInListItem.setSelected(false);

optionsMenu.add(preferencesItem);
optionsMenu.add(useScramblesInListItem);
/********** END OPTIONS ****/

/********** TUTORIAL ****/
tutorialMenu = new JMenu("Tutorial");
crossTutorialMenu = new JMenu("Cross");
cornerTutorialMenu = new JMenu("Corners");
edgeTutorialMenu = new JMenu("Edges");
orientationTutorialMenu = new JMenu("Orientation");
permutationTutorialMenu = new JMenu("Permutation");

controlsTutorialItem = new JMenuItem("Controls");
controlsTutorialItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        switchToTutorialView();
        try {
```

```
        tutorial.loadTutorial("res/Controls.txt");
    } catch (Exception exc) {
        JOptionPane
            .showMessageDialog(null, "Invalid Tutorial File", "Error", JOptionPane.ERROR_MESSAGE);
        switchToTimingView();
    }

    setUpTutorial();
    cubePanel.repaint();
    cubePanel.requestFocus();
}
});

crossTutorialOneItem = new JMenuItem("Introduction");
crossTutorialOneItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        switchToTutorialView();
        try {
            tutorial.loadTutorial("res/CrossIntroduction.txt");
        } catch (Exception exc) {
            JOptionPane
                .showMessageDialog(null, "Invalid Tutorial File", "Error", JOptionPane.ERROR_MESSAGE);
            switchToTimingView();
        }

        setUpTutorial();
        cubePanel.repaint();
        cubePanel.requestFocus();
    }
});

crossTutorialTwoItem = new JMenuItem("Tutorial Two");
crossTutorialTwoItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        switchToTutorialView();
        try {
            tutorial.loadTutorial("res/CrossTutorial.txt");
        } catch (Exception e1) {
            JOptionPane
                .showMessageDialog(null, "Invalid Tutorial File", "Error", JOptionPane.ERROR_MESSAGE);
            switchToTimingView();
        }
    }
});
```

```
        setUpTutorial();
        cubePanel.repaint();
        cubePanel.requestFocus();
    }
});

crossTutorialMenu.add(crossTutorialOneItem);
crossTutorialMenu.add(crossTutorialTwoItem);

cornerTutorialOneItem = new JMenuItem("Introduction");
cornerTutorialOneItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        switchToTutorialView();
        try {
            tutorial.loadTutorial("res/CornersIntroduction.txt");
        } catch (Exception exc) {
            JOptionPane
                .showMessageDialog(null, "Invalid Tutorial File", "Error", JOptionPane.ERROR_MESSAGE);
            switchToTimingView();
        }

        setUpTutorial();
        cubePanel.repaint();
        cubePanel.requestFocus();
    }
});

cornerTutorialMenu.add(cornerTutorialOneItem);

edgeTutorialOneItem = new JMenuItem("Introduction");
edgeTutorialOneItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        switchToTutorialView();
        try {
            tutorial.loadTutorial("res/EdgesIntroduction.txt");
        } catch (Exception exc) {
            JOptionPane
                .showMessageDialog(null, "Invalid Tutorial File", "Error", JOptionPane.ERROR_MESSAGE);
            switchToTimingView();
        }

        setUpTutorial();
        cubePanel.repaint();
    }
});
```

```
        cubePanel.requestFocus();
    }
});

edgeTutorialMenu.add(edgeTutorialOneItem);

orientationTutorialOneItem = new JMenuItem("Introduction");
orientationTutorialOneItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        switchToTutorialView();
        try {
            tutorial.loadTutorial("res/OrientationIntroduction.txt");
            setUpTutorial();
        } catch (Exception exc) {
            JOptionPane
                .showMessageDialog(null, "Invalid Tutorial File", "Error", JOptionPane.ERROR_MESSAGE);
            switchToTimingView();
        }

        cubePanel.repaint();
        cubePanel.requestFocus();
    }
});

orientationTutorialMenu.add(orientationTutorialOneItem);

permutationTutorialOneItem = new JMenuItem("Introduction");
permutationTutorialOneItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        switchToTutorialView();
        try {
            tutorial.loadTutorial("res/PermutationIntroduction.txt");
        } catch (Exception exc) {
            JOptionPane
                .showMessageDialog(null, "Invalid Tutorial File", "Error", JOptionPane.ERROR_MESSAGE);
            switchToTimingView();
        }

        setUpTutorial();
        cubePanel.repaint();
        cubePanel.requestFocus();
    }
});
```

```
});

permutationTutorialMenu.add(permutationTutorialOneItem);

exitTutorialItem = new JMenuItem("Exit Tutorial");
exitTutorialItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        switchToTimingView();
    }
});

loadTutorialFromFileItem = new JMenuItem("Load Tutorial from File - txt");
loadTutorialFromFileItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        if (fileChooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            TextFile currentFile = new TextFile();
            try {
                switchToTutorialView();
                currentFile.setFilePath(fileChooser.getSelectedFile().toString());
                currentFile.setIO(TextFile.READ);

                tutorial.loadTutorial(fileChooser.getSelectedFile().toString());
                setUpTutorial();
            } catch (Exception exc) {
                JOptionPane.showMessageDialog(totalFrame, "Invalid File", "Error",
                    JOptionPane.ERROR_MESSAGE);
                switchToTimingView();
            } finally {
                currentFile.close();
            }
        }
    }
});

tutorialMenu.add(controlsTutorialItem);
tutorialMenu.add(crossTutorialMenu);
tutorialMenu.add(cornerTutorialMenu);
tutorialMenu.add(edgeTutorialMenu);
tutorialMenu.add(orientationTutorialMenu);
tutorialMenu.add(permutationTutorialMenu);
tutorialMenu.add(loadTutorialFromFileItem);
```

```
tutorialMenu.addSeparator();
tutorialMenu.add(exitTutorialItem);
/**************** END TUTORIAL *****/
/* HELP */
helpMenu = new JMenu("Help");

terminologyItem = new JMenuItem("Terminology");
terminologyItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        if (Desktop.isDesktopSupported()) {
            try {
                File currentFile = new File("res/Terminology.pdf");
                Desktop.getDesktop().open(currentFile);
            } catch (IOException e) {
                JOptionPane.showMessageDialog(totalFrame, "No application installed to open PDFs", "Error",
                    JOptionPane.ERROR_MESSAGE);
            }
        }
    }
});

helpMenu.add(terminologyItem);
/**************** END HELP *****/
menuBar.add(fileMenu);
menuBar.add(editMenu);
menuBar.add(viewMenu);
menuBar.add(optionsMenu);
menuBar.add(toolMenu);
menuBar.add(tutorialMenu);
menuBar.add(helpMenu);

return menuBar;
}

/**
 * Switches the main window from timing mode to tutorial mode. The text
 * area at the bottom of the window displays the correct information.
 */
private void setUpTutorial() {
    resetCube();
    solveMaster.rotateToTopFront(Color.white, Color.green);
```

```
solveMaster.clearMoves();
tutorialIsRunning = true;
tutorialTextArea.setText(tutorial.getDescription());
tutorialTextArea.setCaretPosition(0);
cube.performAbsoluteMoves(tutorial.getScramble());
movesAllowed = tutorial.requiresUserAction();
tutorialBackButton.setEnabled(!tutorial.isFirstSubTutorial());
tutorialNextButton.setEnabled(!tutorial.isLastSubTutorial());
tutorialHintButton.setEnabled(tutorial.requiresUserSolution());
tutorialShowSolutionButton.setEnabled(tutorial.requiresUserSolution());
}

/**
 * Switches the main window from timing mode to tutorial mode. The
 * buttons at the bottom of the window are changed and the appropriate
 * fields are changed. Menu items are enabled and disabled accordingly
 */
private void switchToTutorialView() {
    timingDetailsComponentsPanel.setVisible(false);
    tutorialComponentsPanel.setVisible(true);
    tutorialIsRunning = true;
    timeToBeRecorded = false;
    cubeSolved = false;
    timerHasPermissionToStart = false;
    movesToBeRecorded = false;
    clickToSolve = false;
    customPaintingInProgress = false;
    customTimerRunning = false;
    movesAllowed = false;

    trackingMoves.clear();

    solveCubeItem.setEnabled(false);
    clickToSolveItem.setEnabled(false);
    clearStickersItem.setEnabled(false);
    paintCustomStateItem.setEnabled(false);
    paintCustomStateItem.setSelected(false);
    colorSelection.setVisible(false);
    cancelSolveItem.setEnabled(false);
    loadCubeStateItem.setEnabled(false);
    applyRandomScrambleItem.setEnabled(false);
    showStatisticsItem.setEnabled(false);

    scrambleLabel.setText("Tutorial");
}
```

```
cancelSolve();
cubePanel.repaint();
}

/**
 * Switches the main window from tutorial mode to timing mode. The
 * buttons at the bottom of the window are changed and the appropriate
 * fields are changed. Menu items are enabled and disabled accordingly
 */
private static void switchToTimingView() {
    timingDetailsComponentsPanel.setVisible(true);
    tutorialComponentsPanel.setVisible(false);
    tutorialIsRunning = false;
    timeToBeRecorded = false;
    cubeSolved = false;
    timerHasPermissionToStart = true;
    movesToBeRecorded = false;
    clickToSolve = false;
    customPaintingInProgress = false;
    customTimerRunning = false;
    movesAllowed = true;

    trackingMoves.clear();

    solveCubeItem.setEnabled(true);
    clickToSolveItem.setEnabled(true);
    paintCustomStateItem.setEnabled(true);
    cancelSolveItem.setEnabled(true);
    loadCubeStateItem.setEnabled(true);
    applyRandomScrambleItem.setEnabled(true);
    showStatisticsItem.setEnabled(true);
    paintCustomStateItem.setEnabled(true);
    timingDetailsStartNewSolveButton.setEnabled(true);
    scrambleLabel.setText("Scramble: ");
    cubePanel.repaint();
}

/**
 * Determines whether <code>clickToSolveItem</code> is enabled or not
 *
 * @param state
 *          the resulting state for the <code>clickToSolve</code> item
 */
```

```
public static void setSolvePieceSelected(boolean state) {
    clickToSolveItem.setSelected(state);
    clickToSolve = state;
}

/**
 * Determines whether <code>applyRandomScrambleItem</code> is enabled or
 * not
 *
 * @param state
 *      the resulting state for the
 *      <code>applyRandomScramble</code> item
 */
public static void setRandomScrambleEnabled(boolean state) {
    applyRandomScrambleItem.setEnabled(state);
}

/**
 * @return <b>true</b> if <code>useScramblesInListItem</code> is
 *         selected; <br>
 *         <b>false</b> otherwise
 */
public static boolean isUsingScramblesInList() {
    return useScramblesInListItem.isSelected();
}

}

/**
 * Default serialVersionUID
 */
private static final long serialVersionUID = 1L;
/**
 * This stores the thickness of the strokes used to paint the visual cube
 */
final static float strokeThickness = 2.0f;
/**
 * This stroke is used to paint the lines on the visual cube
 */
final static BasicStroke stroke = new BasicStroke(strokeThickness);

/**
 * The cube shown in the main window
 */
```

```
private static Cube cube;
/**
 * This variable allows useful methods to be accessed to operate on the cube
 */
private static SolveMaster solveMaster;
/**
 * This allows a solution for the cross to be generated
 */
private static CrossSolver crossSolver;
/**
 * This allows a solution for the corners to be generated
 */
private static CornerSolver cornerSolver;
/**
 * This allows a solution for the edges to be generated
 */
private static EdgeSolver edgeSolver;
/**
 * This allows a solution for the orientation to be generated
 */
private static OrientationSolver orientationSolver;
/**
 * This allows a solution for the permutation to be generated
 */
private static PermutationSolver permutationSolver;
/**
 * This is used to load tutorials and access their features
 */
private static Tutorial tutorial;
/**
 * This is used to solve a piece that is clicked when in Click-to-Solve mode
 */
private static MouseSelectionSolver selectionSolver;
/**
 * This represents the Color Selection window
 */
private static ColorSelection colorSelection;
/**
 * This represents the Preferences window
 */
private static Preferences preferencesPopUp;
/**
 * This represents the Scramble List window
 */
```

```
private static ScramblePopUp scramblePopUp;
/**
 * This represents the Algorithm Table window
 */
private static AlgorithmDatabasePopUp algorithmDatabasePopUp;
/**
 * This represents the Solve Table window
 */
private static SolveDatabasePopUp solveDatabasePopUp;
/**
 * This represents the Competition Table window
 */
private static CompetitionDatabasePopUp competitionDatabasePopUp;
/**
 * This represents the Member Table window
 */
private static MemberDatabasePopUp memberDatabasePopUp;

/**
 * This array stores the facelet/sticker colours for each sticker on the
 * cube. The first dimension stores the stickers for the top face,
 */
private static Color[][][] faceletColors = new Color[3][3][3];
/**
 * This specifies the order in which the edges should be painted when using
 * the for loops later
 */
private static int[] edgePaintOrder = { 0, 3, 1, 2 };
/**
 * This specifies the order in which the corners should be painted when
 * using the for loops later
 */
private static int[] cornerPaintOrder = { 0, 1, 3, 2 };
/**
 * This stores the number of times that should be graphed
 */
private static int numTimesToGraph = 12;
/**
 * This is used as the display-timer used to record times for solves
 */
private static Timer incTimer;
/**
 * This is used as the count-down timer for inspection
 */
```

```
private static Timer inspectionTimer;
/**
 * This is used to regulate the speed of animations (real-time solving etc.)
 */
private static Timer realTimeSolutionTimer;
/**
 * Stores the number of elapsed seconds while timing
 */
private static float elapsedTimingSeconds = 0.00F;
/**
 * Stores the number of elapsed minutes while timing
 */
private static int elapsedMinutes = 0;
/**
 * Stores the number of seconds remaining during inspection time
 */
private static int inspectionTimeRemaining;

/**
 * This indicates whether the system is in tutorial mode or not
 */
private static boolean tutorialIsRunning = false;
/**
 * This indicates whether the time/solve should be recorded when the timer
 * stops
 */
private static boolean timeToBeRecorded;
/**
 * Indicates whether or not the user is allowed to perform moves.
 */
private static boolean movesAllowed = true;
/**
 * Indicates whether or not the cube is solved
 */
private static boolean cubeSolved = false;
/**
 * Indicates whether or not <code>incTimer</code> has permission to start
 */
private static boolean timerHasPermissionToStart = false;
/**
 * Indicates whether or not the moves being performed by the user should be
 * recorded
 */
private static boolean movesToBeRecorded = false;
```

```
/**  
 * Indicates whether or not the user can click to solve a piece  
 */  
private static boolean clickToSolve = false;  
/**  
 * Indicates whether or not the user can paint a custom state on the cube  
 */  
private static boolean customPaintingInProgress = false;  
/**  
 * Indicates whether or not incTimer is running for a time that is being  
 * performed by the user manually (i.e. the 'Start New Solve' button was not  
 * clicked; spacebar was pressed to start)  
 */  
private static boolean customTimerRunning = false;  
  
/**  
 * This specifies how many seconds to wait before rendering the next time on  
 * the display.  
 */  
private static float timingDisplayInterval = 0.053F;  
/**  
 * This can store moves performed by the user in different situations  
 */  
private static LinkedList<String> trackingMoves = new LinkedList<>();  
/**  
 * This stores the data for the solves listed at the right-hand side of the  
 * main window  
 */  
private static LinkedList<Solve> solves = new LinkedList<>();  
/**  
 * Stores the last scramble used to store the cube  
 */  
private static String currentScramble;  
  
/**  
 * Stores the penalty to apply to the current solve  
 */  
private static String currentPenalty = "0";  
  
/**  
 * Clicking this button resets the cube to the solve state white on top and  
 * green on front  
 */  
private static JButton resetCubeButton;
```

```
/**  
 * This can be used to calculate statistics for the current session  
 */  
private static Statistics statistics;  
/**  
 * This represents the Solve Editor window  
 */  
private static TimeListPopUp timeListPopUp;  
/**  
 * This represents the Time Graph window  
 */  
private static TimeGraph timeGraph;  
  
/**  
 * This label shows the current time of <code>incTimer</code> or  
 * <code>inspectionTimer</code> at the top-left of the window  
 */  
private static JLabel timeLabel;  
/**  
 * This label shows the current scramble being used  
 */  
private static JTextArea scrambleLabel;  
/**  
 * This panel stores the components at the top of the main window  
 */  
private static JPanel topComponentsPanel;  
  
/**  
 * This list can be placed in the panel at the right-hand side of the screen  
 * so that the list can be displayed  
 */  
private static JList<String> listHolder;  
/**  
 * This list stores the contents of the displayed elements in the time list  
 * at the right-hand side of the screen  
 */  
private static DefaultListModel<String> timeDisplayList;  
  
/**  
 * This stores the listed times  
 */  
private static JPanel timeListPanel;  
/**  
 * This panel holds the painted components of the main window (such as
```

```
* <code>cubePanel</code>
*/
private static JPanel totalPaintingPanel;
/**
 * Stores the components unique to timing mode
 */
private static JPanel timingDetailsComponentsPanel;
/**
 * Stores the buttons unique to timing mode
 */
private static JPanel timingDetailsButtonPanel;
/**
 * Stores the components at the bottom of the main window
 */
private static JPanel bottomPanel;
/**
 * The total frame/window to be shown
 */
private static JFrame totalFrame;
/**
 * Stores the buttons unique to tutorial mode
 */
private static JPanel tutorialButtonsPanel;
/**
 * Stores the components unique to tutorial mode
 */
private static JPanel tutorialComponentsPanel;

/**
 * The text for sub-tutorials are shown in this text area
 */
private static JTextArea tutorialTextArea;
/**
 * The text used in timing mode, such as statistics or solutions to
 * cube-states, is shown in this text area.
 */
private static JTextArea timingDetailsTextArea;

/**
 * This scroll pane allows all contents of the time-list to be viewed when
 * its size exceeds the size of the panel in which it is placed
 */
private static JScrollPane timeListScrollPane;
/**
```

```
* This scroll pane allows all text in the tutorial text area to be viewed
* when the length of the text exceeds the size of the text area
*/
private static JScrollPane tutorialScrollPane;
/**
 * This scroll pane allows all text in the timing-details text area to be
 * viewed when the length of the text exceeds the size of the text area
*/
private static JScrollPane timingDetailsScrollPane;

/**
 * Clicking this button clears all times in the list at the right-hand side
 * of the main window
*/
private static JButton timingDetailsResetSessionButton;
/**
 * Clicking this button starts a new solve
*/
private static JButton timingDetailsStartNewSolveButton;
/**
 * Clicking this button resets the cube to the solved state with white on
 * top and green on front
*/
private static JButton tutorialResetStateButton;
/**
 * Clicking this button loads the next sub-tutorial
*/
private static JButton tutorialNextButton;
/**
 * Clicking this button loads the previous sub-tutorial
*/
private static JButton tutorialBackButton;
/**
 * Clicking this button loads the next hint in the sub-tutorial
*/
private static JButton tutorialHintButton;
/**
 * Clicking this button shows the description for the current sub-tutorial
*/
private static JButton tutorialShowDescriptionButton;
/**
 * Clicking this button shows the solution for the current sub-tutorial
*/
private static JButton tutorialShowSolutionButton;
```

```
/**  
 * The visual cube is stored in this panel  
 */  
final static Main cubePanel = new Main();  
  
/**  
 * Constructor - initialises the fields  
 */  
public Main() {  
    super(new BorderLayout());  
    cube = new Cube();  
    solveMaster = new SolveMaster(cube);  
    crossSolver = new CrossSolver(cube);  
    cornerSolver = new CornerSolver(cube);  
    edgeSolver = new EdgeSolver(cube);  
    orientationSolver = new OrientationSolver(cube);  
    permutationSolver = new PermutationSolver(cube);  
    tutorial = new Tutorial(cube);  
    statistics = new Statistics(solves);  
    selectionSolver = new MouseSelectionSolver(cube, crossSolver, cornerSolver, edgeSolver, orientationSolver,  
        permutationSolver);  
  
    addKeyListener(this);  
}  
  
/**  
 * This is used so that pressing tab transfers focus correctly in forms.  
 * This is needed because JTextAreas treat tab as a character rather than a  
 * traversal key.  
 *  
 * @param arg0  
 *      The key event triggered by the key press  
 */  
public static void transferFormFocus(KeyEvent arg0) {  
    Object source = arg0.getSource();  
  
    if (arg0.getKeyCode() == KeyEvent.VK_TAB) {  
        if (arg0.isShiftDown()) {  
            ((Component) source).transferFocusBackward();  
        } else {  
            ((Component) source).transferFocus();  
        }  
        arg0.consume();  
    }  
}
```

```
    }

}

/***
 * Returns an ImageIcon, or null if the path was invalid.
 *
 * @param path
 *         the path of the icon
 * @return the ImageIcon to be used
 */
public static ImageIcon createImageIcon(String path) {
    try {
        return new ImageIcon(path);
    } catch (Exception e) {
        System.err.println("Couldn't find file: " + path);
        return null;
    }
}

/***
 * Returns an Image, or null if the path was invalid.
 *
 * @param path
 *         the path of the icon
 * @return the Image to be used
 */
public static Image createImage(String path) {
    try {
        return ImageIO.read(new File(path));
    } catch (Exception e) {
        System.err.println("Couldn't find file: " + path);
        return null;
    }
}

/***
 * Resizes the widths of the columns in the specified table so that there is
 * maximum visibility in each column
 *
 * @param table
 *         the table whose columns need resized
 */
public static void resizeColumnWidths(JTable table) {
    final TableColumnModel columnModel = table.getColumnModel();
```

```
for (int column = 0; column < table.getColumnCount(); column++) {
    int width = 50; // Min width
    for (int row = 0; row < table.getRowCount(); row++) {
        TableCellRenderer renderer = table.getCellRenderer(row, column);
        Component comp = table.prepareRenderer(renderer, row, column);
        width = Math.max(comp.getPreferredSize().width, width);
    }
    columnModel.getColumn(column).setPreferredWidth(width);
}

/**
 * Returns the dataset to be used by {@link #timeGraph}
 *
 * @return the dataset of the past {@link #numTimesToGraph} times
 */
private static DefaultCategoryDataset getDatasetOfSelectedTimes() {
    // Stores the data contents of the graph
    DefaultCategoryDataset dataset = new DefaultCategoryDataset();
    // Stores the index of the first element in 'solves' to be graphed.
    int start = solves.size() - numTimesToGraph;
    // Stores the number of DNF times in the past 'numTimesToGraph' times.
    int numDNF = Statistics.getNumDNF(numTimesToGraph, solves);
    int end = solves.size();
    // Stores the numeric representation of the current time be examined.
    double current;
    // Accumulates the number of times graphed.
    int numDataValues = 0;
    // Stores the index of a valid time in case there is only one valid time
    // to be graphed.
    int indexOfValidTime = 0;

    /*
     * DNF times are not graphed, so this gets a start index that will allow
     * the maximum numbers of times (<= numTimesToGraph) to be graphed.
     */
    if (start > 0) {
        /*
         * The start index must be greater than or equal to zero, so exit
         * when 'start' will become -1. When numDNF = 0, then this block of
         * code has completed its purpose.
         */
        while ((start > 0) && (numDNF > 0)) {
            --start;
        }
    }
}
```

```
        if (solves.get(start).getNumericTime() != -1)
            --numDNF;
    }
}
/*
 * This will execute when the number of solves is less than the number
 * of times to graph, but if this is the case, then start must be reset
 * to 0. There will be fewer than numTimesToGraph times graphed.
 */
else
    start = 0;

for (int i = start; i < end; ++i) {
    current = solves.get(i).getNumericTime();

    if (current != -1) {
        dataset.addValue(solves.get(i).getNumericTime(), "times", "" + Integer.toString(i + 1));
        ++numDataValues;
        indexOfValidTime = i;
    }
}

/*
 * If the number of dataset values added is 1, then the graph will
 * consist of an invisible point. This if statement adds an identical
 * value to the graph so that instead of a point, it shows a straight
 * line.
 */
if (numDataValues == 1)
    dataset.addValue(solves.get(indexOfValidTime).getNumericTime(), "times", "");

return dataset;
}

/**
 * Determines whether or not the cube is in a valid state
 *
 * @param clearStickers
 *         if <b>true</b>, then when an invalid piece is found, its
 *         stickers will be cleared, i.e. turn grey; <br>
 *         if <b>false</b> then the stickers will remain as they are
 * @return <b>true</b> if the cube is in a valid state; <br>
 *         <b>false</b> otherwise
 */
```

```
public static boolean isValidCubeState(boolean clearStickers) {
    // Stores all valid corner sticker options.
    Color[][] validCornerStickers = Corner.getAllInitialStickers();
    // Stores all valid edge sticker options.
    Color[][] validEdgeStickers = Edge.getAllInitialStickers();
    // Stores all valid corners on a standard Rubik's cube.
    Corner[] validCorners = new Corner[8];
    // Stores all valid edges on a standard Rubik's cube.
    Edge[] validEdges = new Edge[12];
    boolean isValidCubeState = true;
    // Indicates whether the ith corner has already been found. This can be
    // used to check for duplicate pieces.
    boolean[] cornersFound = new boolean[8];
    // Indicates whether the ith edge has already been found. This can be
    // used to check for duplicate pieces.
    boolean[] edgesFound = new boolean[12];
    int validCubieIndex;
    String originalState = solveMaster.getStateString();

    for (int i = 0; i < 8; ++i) {
        validCorners[i] = new Corner(validCornerStickers[i]);
    }

    for (int i = 0; i < 12; ++i) {
        validEdges[i] = new Edge(validEdgeStickers[i]);
    }

    for (int i = 0; i < 8; ++i) { // These two for loops check if every
        // cubie has valid stickers.
        validCubieIndex = getValidCubieIndex(cube.getCorner(i), validCorners);

        /*
         * If valueCubieIndex = -1, then the current corner does not exist
         * on a standard cube, so set isValidCubeState to false. If
         * cornersFound[valueCubieIndex] is already true then this corner is
         * a duplicate, so set isValidCubeState to false.
         */
        if ((validCubieIndex == -1) || (cornersFound[validCubieIndex])) {
            isValidCubeState = false;
            if (clearStickers)
                cube.getCorner(i).setStickers(Color.LIGHT_GRAY, Color.LIGHT_GRAY, Color.LIGHT_GRAY);
        } else
            cornersFound[validCubieIndex] = true;
    }
}
```

```
for (int i = 0; i < 12; ++i) {
    validCubieIndex = getValidCubieIndex(cube.getEdge(i), validEdges);

    if ((validCubieIndex == -1) || (edgesFound[validCubieIndex])) {
        isValidCubeState = false;

        if (clearStickers)
            cube.getEdge(i).setStickers(Color.LIGHT_GRAY, Color.LIGHT_GRAY);
    } else
        edgesFound[validCubieIndex] = true;
}

if (!isValidCubeState)
    return false;

// Start solving cube
solveMaster.rotateToTop(Color.white);
assignOrientationsToCubies();

crossSolver.solveCross();
cornerSolver.solveFirstLayerCorners();
edgeSolver.solveMiddleLayerEdges();
orientationSolver.solveOrientation();
permutationSolver.solvePermutation();
// Stop solving cube

/*
 * This block checks if any pieces are unsolved after the solve masters
 * have generated a solution.
 */
for (int i = 0; i < 4; ++i) {
    if (!crossSolver.isPieceSolved(cube.getCorner(i))) {
        isValidCubeState = false;

        if (clearStickers)
            cube.getCorner(i).setStickers(Color.LIGHT_GRAY, Color.LIGHT_GRAY, Color.LIGHT_GRAY);
    }
    if (!crossSolver.isPieceSolved(cube.getEdge(i))) {
        isValidCubeState = false;

        if (clearStickers)
            cube.getEdge(i).setStickers(Color.LIGHT_GRAY, Color.LIGHT_GRAY);
    }
}
```

```
solveMaster.clearMoves();
SolveMaster.simplifyMoves(crossSolver.getCatalogMoves(), SolveMaster.CROSS);
SolveMaster.simplifyMoves(cornerSolver.getCatalogMoves(), SolveMaster.CORNER_EDGE);
SolveMaster.simplifyMoves(edgeSolver.getCatalogMoves(), SolveMaster.CORNER_EDGE);
SolveMaster.simplifyMoves(orientationSolver.getCatalogMoves(), SolveMaster.CANCELLATIONS);
SolveMaster.simplifyMoves(permutationSolver.getCatalogMoves(), SolveMaster.CANCELLATIONS);

solveMaster.applyStateString(originalState);

return isValidCubeState;
}

/**
 * Returns the index of <code>cubie</code> in <code>validCubies</code>
 *
 * @param cubie
 *          the cubie to find
 * @param validCubies
 *          the array in which <code>cubie</code> will be found
 * @return the index of <code>cubie</code> in <code>validCubies</code>. <br>
 *         If the cubie cannot be found, then <code>-1</code> will be
 *         returned
 */
private static int getValidCubieIndex(Cubie cubie, Cubie[] validCubies) {
    for (int i = 0; i < validCubies.length; ++i) {
        if (cubie.strictCompareTo(validCubies[i]) == 0)
            return i;
    }

    return -1;
}

/**
 * Assigns the corresponding orientation to each cubie on the cube. This
 * method is used to find the orientation of each piece after, e.g. painting
 * a custom state or loading a state from file
 */
public static void assignOrientationsToCubies() {
    // Stores the original colours of the centres on top and front.
    Color[] originalTopFront = { cube.getSlice(0).getCentre(), cube.getSlice(4).getCentre() };
    solveMaster.rotateToTopFront(Color.white, Color.green);

    // Stores the colour of the current centre at the top
```

```
Color topCentre = cube.getSlice(0).getCentre();
// Stores the colour of the current centre at the bottom
Color bottomCentre = cube.getSlice(1).getCentre();
// Stores the colour of the current centre at the right
Color rightCentre = cube.getSlice(2).getCentre();
// Stores the colour of the current centre at the left
Color leftCentre = cube.getSlice(3).getCentre();

// Stores the orientation of the current cubie.
int orientation;
// Stores the properties of the current corner being examined.
Corner currentCorner;
// Stores the properties of the current edge being examined.
Edge currentEdge;
// Stores the stickers on the current edge being examined.
Color[] edgeStickers;

for (int i = 0; i < 8; ++i) {
    currentCorner = cube.getCorner(i);

    /*
     * Finds the index of the top or bottom centre on the corner so that
     * the orientation can be determined.
     */
    if ((orientation = LinearSearch.linearSearchCornerOrientation(currentCorner.getStickers(), topCentre)) == -2)
        currentCorner.setOrientation(LinearSearch.linearSearchCornerOrientation(currentCorner.getStickers(),
            bottomCentre));
    else
        currentCorner.setOrientation(orientation);
}

for (int i = 0; i < 12; ++i) {
    currentEdge = cube.getEdge(i);
    edgeStickers = currentEdge.getStickers();

    if ((edgeStickers[0].equals(rightCentre)) || (edgeStickers[0].equals(leftCentre))
        || (edgeStickers[1].equals(topCentre)) || (edgeStickers[1].equals(bottomCentre)))
        currentEdge.setOrientation(1);
    else
        currentEdge.setOrientation(0);
}

solveMaster.rotateToTopFront(originalTopFront[0], originalTopFront[1]);
}
```

```
/**  
 * @see javax.swing.JComponent#addNotify()  
 */  
@Override  
public void addNotify() {  
    super.addNotify();  
    requestFocus();  
}  
  
/**  
 * @see java.awt.event.KeyListener#keyPressed(java.awt.event.KeyEvent)  
 */  
public void keyPressed(KeyEvent e) {  
}  
  
/**  
 * This method handles the release of spacebar or the solving of the cube  
 * after a key is released.  
 *  
 * @see java.awt.event.KeyListener#keyReleased(java.awt.event.KeyEvent)  
 */  
public void keyReleased(KeyEvent e) {  
    /*  
     * This method stops timers and records times, so if a tutorial is  
     * running or custom painting is in progress, then nothing should  
     * happen, so return.  
     */  
    if (tutorialIsRunning || customPaintingInProgress)  
        return;  
  
    if (e.getKeyChar() == ' ') {  
        /*  
         * This starts the timer when the inspection timer is running  
         */  
        if (timerHasPermissionToStart && (inspectionTimer != null)) {  
            inspectionTimer.stop();  
            inspectionTimer = null;  
            timerHasPermissionToStart = false;  
            timeToBeRecorded = true;  
            movesAllowed = true;  
            timeLabel.setForeground(Color.black);  
            elapsedTimingSeconds = 0.00F;  
            elapsedMinutes = 0;  
        }  
    }  
}
```

```
incTimer = new Timer((int) (timingDisplayInterval * 1000.0), new TimerListener());
incTimer.start();

int size = solves.size();
if (size > 0) {
    listHolder.setSelectedIndex(size - 1);
    listHolder.ensureIndexIsVisible(size - 1);
}
} else if ((inspectionTimer == null) && (realTimeSolutionTimer == null) && (incTimer == null)
    && (movesAllowed) && (!tutorialIsRunning)) {
    // Start timer with spacebar (no inspection)
    customTimerRunning = true;
    timerHasPermissionToStart = false;
    timeToBeRecorded = false;
    movesToBeRecorded = false;
    movesAllowed = true;
    timeLabel.setForeground(Color.black);
    elapsedTimingSeconds = 0.00F;
    elapsedMinutes = 0;
    incTimer = new Timer((int) (timingDisplayInterval * 1000.0), new TimerListener());
    incTimer.start();
} else if ((!timeToBeRecorded) && (incTimer != null) && (incTimer.isRunning())) {
    // Stop timer with spacebar
    customTimerRunning = false;
    incTimer.stop();
    incTimer = null;
    timerHasPermissionToStart = true;

    int previousSelectedIndex = listHolder.getSelectedIndex();
    SolveMaster.simplifyMoves(trackingMoves, SolveMaster.CANCELLATIONS);

    Solve currentTime = new Solve(timeLabel.getText(), "0", "");
    solves.add(currentTime);
    timeDisplayList.addElement(timeLabel.getText());

    /*
     * Makes sure the appropriate element in the list is selected.
     */
    if (previousSelectedIndex == solves.size() - 2)
        listHolder.setSelectedIndex(previousSelectedIndex + 1);

    listHolder.ensureIndexIsVisible(listHolder.getSelectedIndex());
    refreshTimeGraph(true);
```

```
        refreshStatistics();
    }
} else if (cubeSolved && timeToBeRecorded) {
    endSolve();
}
}

/**
 * @see java.awt.event.KeyListener#keyTyped(java.awt.event.KeyEvent)
 */
public void keyTyped(KeyEvent e) {
    if (e.isAltDown() || (e.getKeyChar() == ' '))
        return;

    // Stores the move in the correct notation, e.g. instead of "j", it
    // stores "U"
    String move = SolveMaster.getKeyToMove("") + e.getKeyChar();

    if (SolveMaster.isValidMove(move)) {
        if (movesToBeRecorded) {
            if (inspectionTimer != null) {
                if ("xyz".contains(move.substring(0, 1))) {
                    trackingMoves.add(move);
                }
            } else {
                trackingMoves.add(move);
            }
        }

        if (!movesAllowed) {
            if (realTimeSolutionTimer == null) {
                cube.rotate(move);
                repaint();
                return;
            }
        } else if (!customPaintingInProgress) {
            cube.performAbsoluteMoves(move);
        }
    }
}

if ((!tutorialIsRunning) && (!customPaintingInProgress) && (!customTimerRunning) && (incTimer != null)
    && (incTimer.isRunning()) && (isCubeSolved())) {
    // Stops timer when cube is solved
    incTimer.stop();
}
```

```
incTimer = null;
movesToBeRecorded = false;
timerHasPermissionToStart = false;
cubeSolved = true;
}

repaint();

if (tutorialIsRunning) { // Check if criteria is filled
    if (tutorial.criteriaFilled()) {
        // Stores the text to be shown in the dialog box
        String dialogText = "";

        SolveMaster.simplifyMoves(trackingMoves, SolveMaster.CANCELLATIONS);

        if (Tutorial.getNumMovesWithoutRotations(trackingMoves) <= tutorial.getOptimalSolutionLength()) {
            dialogText = "Well done!" + "\nWould you like to play again?";
        } else {
            dialogText = "You solved the problem, but you could have done it in fewer moves."
                + "\nWould you like to try again?";
        }
    }

    trackingMoves.clear();

    Object[] options = { "Yes", "No" };
    // Stores 0 if the user selects 'Yes'.
    int choice = JOptionPane.showOptionDialog(null, dialogText, "Congratulations",
        JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE, null, options, options[1]);

    if (choice == 0) // User wants to play again
        tutorialResetStateButton.doClick();
    else { // User wants to move on OR user closes dialog
        tutorialShowSolutionButton.doClick();
        movesAllowed = false;
        movesToBeRecorded = false;
    }
}
}

/***
 * Assigns the sticker colours to the appropriate elements of
 * <code>faceColors</code>
 */
}
```

```
private void assignFaceletPaintingColors() {
    /*
     * This specifies that the top face should be painted first, then the
     * front face, then the right face.
     */
    int[] sliceIndices = { 0, 4, 2 };
    // Stores the properties of the current slice be examined.
    Slice currentSlice;
    // Stores the index of the current corner being examined on the current
    // slice.
    int cornerIndex;
    // Stores the index of the current corner being examined on the current
    // slice.
    int edgeIndex;

    currentSlice = cube.getSlice(sliceIndices[0]);
    cornerIndex = 0;
    edgeIndex = 0;
    // Iterates over each row of the current face
    for (int i = 0; i < 3; ++i) {
        // Iterates over each column of the current face
        for (int j = 0; j < 3; ++j) {
            if ((i == 1) && (j == 1))
                faceletColors[0][i][j] = currentSlice.getCentre();
            else if ((i + j) % 2 == 0)
                faceletColors[0][i][j] = currentSlice.getCorner(cornerPaintOrder[cornerIndex++]).getStickers()[0];
            else
                faceletColors[0][i][j] = currentSlice.getEdge(edgePaintOrder[edgeIndex++]).getStickers()[0];
        }
    }

    currentSlice = cube.getSlice(sliceIndices[1]);
    cornerIndex = 0;
    edgeIndex = 0;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if ((i == 1) && (j == 1))
                faceletColors[1][i][j] = currentSlice.getCentre();
            else if ((i + j) % 2 == 0)
                /*
                 * i.e. if it is the FLU or FRD corner, then get the second
                 * sticker, and if it is the FUR or FDL corner, then get the
                 * third sticker
                */
        }
    }
}
```

```
        faceletColors[1][i][j] =
currentSlice.getCorner(cornerPaintOrder[cornerIndex]).getStickers() [(cornerPaintOrder[cornerIndex++] % 2 == 0) ? 1
: 2];
    else
        /*
         * i.e. if it is the FU or FD edge, then get the second
         * sticker, otherwise get the first sticker.
         */
        faceletColors[1][i][j] =
currentSlice.getEdge(edgePaintOrder[edgeIndex]).getStickers() [(edgePaintOrder[edgeIndex++] % 2 == 0) ? 1
: 0];
    }
}

currentSlice = cube.getSlice(sliceIndices[2]);
cornerIndex = 0;
edgeIndex = 0;
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
        if ((i == 1) && (j == 1))
            faceletColors[2][i][j] = currentSlice.getCentre();
        else if ((i + j) % 2 == 0)
            faceletColors[2][i][j] =
currentSlice.getCorner(cornerPaintOrder[cornerIndex]).getStickers() [(cornerPaintOrder[cornerIndex++] % 2 == 0) ? 1
: 2];
        else
            faceletColors[2][i][j] = currentSlice.getEdge(edgePaintOrder[edgeIndex++]).getStickers()[1];
    }
}
/***
 * @see javax.swing.JComponent#paintComponent(java.awt.Graphics)
 */
@Override
public void paintComponent(Graphics g) {
    int rectHeight, rectWidth, gridWidth, gridHeight;
    double x, y = 10;
    Graphics2D g2 = (Graphics2D) g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
    g2.clearRect(0, 0, getWidth(), getHeight());
    g2.setPaint(this.getBackground());
    g2.fillRect(0, 0, this.getWidth(), this.getHeight());
```

```
Dimension d = getSize();
gridWidth = d.width / 9;
gridHeight = d.height / 9;
rectWidth = gridWidth;
rectHeight = gridHeight;

assignFaceletPaintingColors();

// TOP FACE
g2.shear(-0.8, 0);
for (int i = 0; i < 3; ++i) {
    x = d.width / 3;
    for (int j = 0; j < 3; ++j) {
        g2.setStroke(stroke);
        g2.setPaint(Color.black);
        g2.draw(new Rectangle2D.Double(x, y, rectWidth, 0.8 * rectHeight));

        g2.setPaint(faceletColors[0][i][j]);

        g2.fill(new Rectangle2D.Double(x + strokeThickness - 1, y + strokeThickness - 1, rectWidth
            - strokeThickness + 0.5, (0.8 * rectHeight) - strokeThickness + 0.1));

        x += gridWidth;
    }
    y += 0.8 * gridHeight;
}

// FRONT FACE
g2.shear(0.8, 0);
for (int i = 0; i < 3; ++i) {
    x = 0.328 * (d.width / 3);
    for (int j = 0; j < 3; ++j) {
        g2.setStroke(stroke);
        g2.setPaint(Color.black);
        g2.draw(new Rectangle2D.Double(x, y, rectWidth, rectHeight));

        g2.setPaint(faceletColors[1][i][j]);

        g2.fill(new Rectangle2D.Double(x + strokeThickness - 1, y + strokeThickness - 1, rectWidth
            - strokeThickness + 0.5, rectHeight - strokeThickness + 0.1));

        x += gridWidth;
}
x = 250;
```

```
        y += gridHeight;
    }

    // RIGHT FACE
    g2.shear(0, -1.26);
    y = 501;
    for (int i = 0; i < 3; ++i) {
        x = 0.328 * (d.width / 3) + (3 * gridWidth);
        for (int j = 0; j < 3; ++j) {
            g2.setStroke(stroke);
            g2.setPaint(Color.black);
            g2.draw(new Rectangle2D.Double(x, y, (0.8 * rectWidth) - 10.5, rectHeight - 1));

            g2.setPaint(faceletColors[2][i][j]);

            g2.fill(new Rectangle2D.Double(x + strokeThickness - 1, y + strokeThickness - 1, (0.8 * rectWidth) - 11
                - strokeThickness + 0.3, rectHeight - 1 - strokeThickness + 0.2));
        }
        x += gridWidth - 24;
    }
    x = 250;
    y += gridHeight;
}

/**
 * Sets whether or not the 'Apply Random Scramble' menu item is enabled.
 *
 * @param state
 *      the resulting enabled/disabled state of the menu item
 */
public static void setRandomScrambleEnabled(boolean state) {
    MenuBar.setRandomScrambleEnabled(state);
}

/*
 * public static void printRuntimeStats() { Runtime runtime1 =
 * Runtime.getRuntime(); System.out.println("max memory: " +
 * runtime1.maxMemory() / (1024 * 1024));
 *
 * Runtime runtime2 = Runtime.getRuntime();
 * System.out.println("allocated memory: " + runtime2.totalMemory() / (1024 *
 * 1024));
 */
```

```
*  
* Runtime runtime3 = Runtime.getRuntime();  
* System.out.println("free memory: " + runtime3.freeMemory() / (1024 *  
* 1024)); }  
*/  
  
/**  
 * Records the specified list in {@link #solves} and appends the time to the  
 * display-list  
 *  
 * @param solve  
 *         the solve to be added  
 */  
public static void addSolveToList(Solve solve) {  
    solves.add(new Solve(solve.getStringTime(), solve.getPenalty(), solve.getComment(), solve.getScramble(), solve  
        .getSolution()));  
    copyAllTimesToDisplay();  
    listHolder.setSelectedIndex(solves.size() - 1);  
}  
  
/**  
 * Scrambles the cube (instantly) with the specified scramble and performs  
 * the solution in real-time as an animation at the speed defined in  
 * Preferences  
 *  
 * @param scramble  
 *         the scramble to be applied before the animation starts  
 * @param solution  
 *         the moves to be performed in real-time  
 */  
public static void performRealTimeSolving(String scramble, String solution) {  
    try {  
        if (realTimeSolutionTimer != null)  
            realTimeSolutionTimer.stop();  
  
        movesAllowed = false;  
  
        LinkedList<String> movesToPerform = new LinkedList<>();  
        LinkedList<String> movesCopy;  
        int size;  
        String current;  
  
        resetCube();  
        cube.performAbsoluteMoves(scramble);  
    }  
}
```

```
solveMaster.catalogMoves(solution);

movesCopy = solveMaster.getCatalogMoves();

size = movesCopy.size();
for (int i = 0; i < size; ++i) {
    current = movesCopy.get(i);
    if (current.substring(current.length() - 1).equals("2")) {
        movesToPerform.add(current.substring(0, current.length() - 1));
        movesToPerform.add(current.substring(0, current.length() - 1));
    } else
        movesToPerform.add(current);
}

realTimeSolutionTimer = null;
realTimeSolutionTimer = new Timer(preferencesPopUp.getRealTimeSolvingRate(), new RealTimeTimerListener(
    movesToPerform));
realTimeSolutionTimer.start();

// MenuBar.setRandomScrambleEnabled(false);
// timingDetailsStartNewSolveButton.setEnabled(false);

cubePanel.repaint();

solveMaster.clearMoves();
} catch (Exception e) {
    movesAllowed = true;
}
}

/**
 * This solves the cube and shows the moves as an animation (real-time) at
 * the speed specified in Preferences
 */
public static void performRealTimeSolving() {
    try {
        if (realTimeSolutionTimer != null)
            realTimeSolutionTimer.stop();

        movesAllowed = false;
        LinkedList<String> moves = new LinkedList<>();
        LinkedList<String> crossSolverMoves = crossSolver.getCatalogMoves();
        LinkedList<String> cornerSolverMoves = cornerSolver.getCatalogMoves();
```

```
LinkedList<String> edgeSolverMoves = edgeSolver.getCatalogMoves();
LinkedList<String> orientationSolverMoves = orientationSolver.getCatalogMoves();
LinkedList<String> permutationSolverMoves = permutationSolver.getCatalogMoves();
String current;

solveMaster.rotateToTopFront(Cubie.getWordToColor(crossSolverMoves.get(1)),
    Cubie.getWordToColor(crossSolverMoves.get(4)));
solveMaster.clearMoves();

int[] length = { crossSolver.getCatalogMoves().size(), cornerSolver.getCatalogMoves().size(),
    edgeSolver.getCatalogMoves().size(), orientationSolver.getCatalogMoves().size(),
    permutationSolver.getCatalogMoves().size() };

for (int i = 0; i < length[0]; ++i) {
    current = crossSolverMoves.get(i);
    if (SolveMaster.isValidMove(current)) { // We need this check
        // for the cross because
        // of the
        // "X on top and Y on front"
        // thing.
        if (current.substring(current.length() - 1).equals("2")) {
            moves.add(current.substring(0, current.length() - 1));
            moves.add(current.substring(0, current.length() - 1));
        } else
            moves.add(current);
    }
}
for (int i = 0; i < length[1]; ++i) {
    current = cornerSolverMoves.get(i);
    if (current.substring(current.length() - 1).equals("2")) {
        moves.add(current.substring(0, current.length() - 1));
        moves.add(current.substring(0, current.length() - 1));
    } else
        moves.add(current);
}
for (int i = 0; i < length[2]; ++i) {
    current = edgeSolverMoves.get(i);
    if (current.substring(current.length() - 1).equals("2")) {
        moves.add(current.substring(0, current.length() - 1));
        moves.add(current.substring(0, current.length() - 1));
    } else
        moves.add(current);
}
for (int i = 0; i < length[3]; ++i) {
```

```
        current = orientationSolverMoves.get(i);
        if (current.substring(current.length() - 1).equals("2")) {
            moves.add(current.substring(0, current.length() - 1));
            moves.add(current.substring(0, current.length() - 1));
        } else
            moves.add(current);
    }
    for (int i = 0; i < length[4]; ++i) {
        current = permutationSolverMoves.get(i);
        if (current.substring(current.length() - 1).equals("2")) {
            moves.add(current.substring(0, current.length() - 1));
            moves.add(current.substring(0, current.length() - 1));
        } else
            moves.add(current);
    }

    realTimeSolutionTimer = null;
    realTimeSolutionTimer = new Timer((int) (preferencesPopUp.getRealTimeSolvingRate()),
        new RealTimeTimerListener(moves));
    realTimeSolutionTimer.start();

    // MenuBar.setRandomScrambleEnabled(false);
    // timingDetailsStartNewSolveButton.setEnabled(false);

    crossSolverMoves.clear();
    cornerSolverMoves.clear();
    edgeSolverMoves.clear();
    orientationSolverMoves.clear();
    permutationSolverMoves.clear();
} catch (Exception e) {
    movesAllowed = true;
}

crossSolver.clearMoves();
cornerSolver.clearMoves();
edgeSolver.clearMoves();
orientationSolver.clearMoves();
permutationSolver.clearMoves();
}

/**
 * Cancels all animations, timers, etc., and labels reset to their default
 * values.
 */
```

```
private static void cancelSolve() {
    /*
     * boolean enabled = !customPaintingInProgress && !tutorialIsRunning;
     * MenuBar.setRandomScrambleEnabled(enabled);
     * MenuBar.clickToSolveItem.setEnabled(enabled);
     * MenuBar.paintCustomStateItem.setEnabled(enabled);
     * MenuBar.solveCubeItem.setEnabled(true);
     * timingDetailsStartNewSolveButton.setEnabled(true);
     */
    MenuBar.clickToSolveItem.setSelected(clickToSolve);
    timeToBeRecorded = false;
    timerHasPermissionToStart = false;
    movesAllowed = !customPaintingInProgress;

    trackingMoves.clear();
    clearAllSolverMoves();

    try {
        inspectionTimer.stop();
        inspectionTimer = null;
    } catch (Exception e) {
    }
    try {
        incTimer.stop();
        incTimer = null;
    } catch (Exception e) {
    }
    try {
        realTimeSolutionTimer.stop();
        realTimeSolutionTimer = null;
    } catch (Exception e) {
    }

    timeLabel.setForeground(Color.black);
    timeLabel.setText("0.00");
}

/**
 * Performs the operations required to end the solve in the appropriate way.
 * After the user completes a solve after clicking the 'Start New Solve'
 * button, this method will be invoked and the time will be recorded. But in
 * other situations, the time will not be recorded.
 */
private static void endSolve() {
```

```
/*
 * boolean enabled = !customPaintingInProgress && !tutorialIsRunning;
 * MenuBar.setRandomScrambleEnabled(enabled);
 * MenuBar.clickToSolveItem.setEnabled(enabled);
 * MenuBar.paintCustomStateItem.setEnabled(enabled);
 * MenuBar.solveCubeItem.setEnabled(true);
 * timingDetailsStartNewSolveButton.setEnabled(true);
 */
MenuBar.clickToSolveItem.setSelected(false);
timeToBeRecorded = false;

String currentTimeString = timeLabel.getText();

if (currentPenalty.equals("+2")) {
    currentTimeString = "" + (Double.parseDouble(currentTimeString) + 2);
} else if (currentPenalty.equals("DNF")) {
    currentTimeString = "DNF";
}

int previousSelectedIndex = listHolder.getSelectedIndex();
SolveMaster.simplifyMoves(trackingMoves, SolveMaster.CANCELLATIONS);

Solve currentTime = new Solve(currentTimeString, currentPenalty.replaceAll("[+]", ""), "", currentScramble,
    trackingMoves);
solves.add(currentTime);
String timeFormat = currentTime.getStringTime() + (currentPenalty.equals("+2") ? " (+2)" : "");
timeDisplayList.addElement(timeFormat);

if (previousSelectedIndex == solves.size() - 2)
    listHolder.setSelectedIndex(previousSelectedIndex + 1);

listHolder.ensureIndexIsVisible(listHolder.getSelectedIndex());

refreshTimeGraph(true);
refreshStatistics();

movesToBeRecorded = false;
trackingMoves.clear();
}

/**
 * @return the solution to the cube state in a formatted fashion
 */
public static String getFormattedCubeSolution() {
```

```
// String formattedString = "";  
  
// These are done in 'isValidCubeState(...)', so don't do them again  
/*  
 * crossSolver.solveCross(); cornerSolver.solveFirstLayerCorners();  
 * edgeSolver.solveMiddleLayerEdges();  
 * orientationSolver.solveOrientation();  
 * permutationSolver.solvePermutation();  
 */  
/*  
 * formattedString = ("-----Solution-----\n" + "Cross: \t"  
 * + crossSolver.getStringMoves() + "\nCorners: \t" +  
 * cornerSolver.getStringMoves() + "\nEdges: \t" +  
 * edgeSolver.getStringMoves() + "\nOrientation: \t" +  
 * orientationSolver.getStringMoves() + "\nPermutation: \t" +  
 * permutationSolver.getStringMoves() + "\n\n" +  
 * "-----Explanation-----\n" +  
 * cornerSolver.getSolutionExplanation() + "\n" +  
 * edgeSolver.getSolutionExplanation() + "\n" +  
 * orientationSolver.getSolutionExplanation() + "\n" +  
 * permutationSolver.getSolutionExplanation());  
 */  
  
return ("-----Solution-----\n"  
+ ((crossSolver.getStringMoves().trim().equals("")) ? "" : "Cross: \t" + crossSolver.getStringMoves())  
+ ((cornerSolver.getStringMoves().trim().equals("")) ? "" : "\nCorners: \t"  
+ cornerSolver.getStringMoves())  
+ ((edgeSolver.getStringMoves().trim().equals("")) ? "" : "\nEdges: \t" + edgeSolver.getStringMoves())  
+ ((orientationSolver.getStringMoves().trim().equals("")) ? "" : "\nOrientation: \t"  
+ orientationSolver.getStringMoves())  
+ ((permutationSolver.getStringMoves().trim().equals("")) ? "" : "\nPermutation: \t"  
+ permutationSolver.getStringMoves())  
+ "\n\n-----Explanation-----\n"  
+ ((cornerSolver.getSolutionExplanation().trim().equals("")) ? "" : "Corners:\n"  
+ cornerSolver.getSolutionExplanation() + "\n")  
+ ((edgeSolver.getSolutionExplanation().trim().equals("")) ? "" : "Edges:\n"  
+ edgeSolver.getSolutionExplanation() + "\n")  
+ ((orientationSolver.getSolutionExplanation().trim().equals("")) ? "" : "Orientation:\n"  
+ orientationSolver.getSolutionExplanation() + "\n") + ((permutationSolver  
.getSolutionExplanation().trim().equals("")) ? "" : "Permutation:\n"  
+ permutationSolver.getSolutionExplanation()));  
  
// return formattedString;  
}
```

```
/**  
 * Refreshes the list of times after a time is added, edited or deleted  
 */  
public static void refreshTimeList() {  
    // int selectedIndex = listHolder.getSelectedIndex();  
    Solve current;  
    timeDisplayList.clear();  
  
    for (int i = 0; i < solves.size(); ++i) {  
        current = solves.get(i);  
        if (current.getStringTime().equals("-1")) {  
            solves.remove(i);  
            --i; // so that i stays the same  
        } else  
            timeDisplayList.addElement(current.getStringTime()  
                + ((!current.getPenalty().equals("0") && (!current.getPenalty().equals("")))) ? String.format(  
                    " (%s)", current.getPenalty()) : "");  
    }  
    /*  
     * if (selectedIndex != -1) { String penalty =  
     * solves.get(selectedIndex).getPenalty(); if  
     * (solves.get(selectedIndex).getStringTime().equals("-1")) {  
     * solves.remove(selectedIndex); timeDisplayList.clear();  
     *  
     * for (int i = 0; i < solves.size(); ++i) {  
     * timeDisplayList.addElement(solves.get(i).getStringTime()); } } else  
     * timeDisplayList.set(selectedIndex,  
     * solves.get(selectedIndex).getStringTime() + ((!penalty.equals("0") &&  
     * (!penalty.equals("")))) ? String.format(" (%s)", penalty) : "");  
     */  
}  
  
/**  
 * Copies all times from {@link #solves} to {@link #timeDisplayList}  
 */  
public static void copyAllTimesToDisplay() {  
    int size = solves.size();  
    String penalty;  
    Solve current;  
    timeDisplayList.clear();  
  
    for (int i = 0; i < size; ++i) {  
        current = solves.get(i);  
    }  
}
```

```
penalty = current.getPenalty();
timeDisplayList.addElement(solves.get(i).getStringTime()
    + (!penalty.equals("0") ? String.format(" (%s)", penalty) : ""));
}

/***
 * Refreshes the statistics and resets the scrolling position to the top of
 * the text area
 */
public static void refreshStatistics() {
    timingDetailsTextArea.setText(statistics.getRecentFormattedStandardStatistics());
    timingDetailsTextArea.setCaretPosition(0);
}

/***
 * Refreshes the dataset being drawn in the Time Graph window.
 *
 * @param cubePanelFocus
 *         if <code>true</code>, then the cube panel should be focused
 *         after method executes; <br>
 *         otherwise the existing focus-window should keep focus
 */
public static void refreshTimeGraph(boolean cubePanelFocus) {
    timeGraph.setDataset(getDatasetOfSelectedTimes());
    timeGraph.draw();
    timeGraph.validate();

    if (cubePanelFocus)
        cubePanel.requestFocus();
}

/***
 * Initialises the components relating to the list of times at the
 * right-hand side of the main window.
 */
public static void initListPanel() {
    timeListPanel = new JPanel();
    timeDisplayList = new DefaultListModel<String>();

    listHolder = new JList<String>(timeDisplayList);
    listHolder.setFont(new Font("Arial", 0, 20));
    listHolder.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    listHolder.setLayoutOrientation(JList.VERTICAL);
```

```
listHolder.setSelectedIndex(0);
listHolder.addMouseListener(new MouseListener() {
    @Override
    public void mouseClicked(MouseEvent e) {
        if ((e.getClickCount() == 2) && (solves.size() > 0)) {
            timeListPopUp.setSolve(solves.get(listHolder.getSelectedIndex()));
            timeListPopUp.selectAllTimeText();
            timeListPopUp.setVisible(true);
        }
    }

    @Override
    public void mouseEntered(MouseEvent e) {
    }

    @Override
    public void mouseExited(MouseEvent e) {
    }

    @Override
    public void mousePressed(MouseEvent e) {
    }

    @Override
    public void mouseReleased(MouseEvent e) {
    }
});

listHolder.addKeyListener(new KeyListener() {
    @Override
    public void keyPressed(KeyEvent arg0) {
    }

    @Override
    public void keyReleased(KeyEvent arg0) {
    }

    @Override
    public void keyTyped(KeyEvent e) {
        int selectedIndex = listHolder.getSelectedIndex();

        if (selectedIndex == -1)
            return;
    }
});
```

```
        if (e.getKeyChar() == KeyEvent.VK_DELETE) {
            solves.get(selectedIndex).setStringTime("-1");
            refreshTimeList();
            refreshStatistics();
            copyAllTimesToDisplay();
            refreshTimeGraph(true);

            if (solves.size() > 0)
                listHolder.setSelectedIndex((selectedIndex < solves.size()) ? selectedIndex : selectedIndex - 1);

            listHolder.requestFocus();
        } else if (e.getKeyChar() == KeyEvent.VK_ENTER) {
            timeListPopUp.setSolve(solves.get(listHolder.getSelectedIndex()));
            timeListPopUp.selectAllTimeText();
            timeListPopUp.setVisible(true);
        }
    }
});
```

```
timeListScrollPane = new JScrollPane(listHolder);
timeListPanel.add(timeListScrollPane);

listHolder.setFocusable(true);
}

/**
 * @return <code>true</code> if the cube is solved; <br>
 *         <code>false</code> otherwise
 */
public static boolean isCubeSolved() {
    Edge[] edges = new Edge[12];
    Corner[] corners = new Corner[8];
    Color[][] edgeStickers = Edge.getAllInitialStickers();
    Color[][] cornerStickers = Corner.getAllInitialStickers();
    boolean isSolved = true;

    solveMaster.rotateToTop(Color.yellow);

    for (int i = 0; i < edges.length; ++i) {
        edges[i] = new Edge(edgeStickers[i]);

        if (!crossSolver.isPieceSolved(edges[i])) { // These used to be
            // 'pieceSolved(...)',
            // so if something goes
```

```
        // wrong, try changing
        // these back
    isSolved = false;
    break;
}
}

if (isSolved) {
    for (int i = 0; i < corners.length; ++i) {
        corners[i] = new Corner(cornerStickers[i]);

        if (!crossSolver.isPieceSolved(corners[i])) { // These used to
            // be
            // 'pieceSolved(...)',
            // so if
            // something
            // goes wrong,
            // try changing
            // these back
        isSolved = false;
        break;
    }
}
}

cube.performAbsoluteMoves(SolveMaster.getReverseStringMoves(solveMaster.getCatalogMoves()) );
solveMaster.clearMoves();

return isSolved;
}

/**
 * Requests focus for {@link #cubePanel}
 */
public static void requestCubePanelFocus() {
    cubePanel.requestFocus();
}

/*
 * private static void solveCube() { crossSolver.solveCross();
 * cornerSolver.solveFirstLayerCorners();
 * edgeSolver.solveMiddleLayerEdges(); orientationSolver.solveOrientation();
 * permutationSolver.solvePermutation();
 */
```

```
*  
* crossSolver.clearMoves(); cornerSolver.clearMoves();  
* edgeSolver.clearMoves(); orientationSolver.clearMoves();  
* permutationSolver.clearMoves(); }  
*/  
/**  
 * Resets the cube to the solve state with white on top and green on front  
 */  
private static void resetCube() {  
    solveMaster.rotateToTopFront(Color.white, Color.green);  
    solveMaster.clearMoves();  
    cube.resetCube();  
}  
  
/**  
 * Sets the size of the scramble text to the specified size  
 *  
 * @param size  
 *          the size for the text of the scramble  
 */  
public static void setScrambleTextSize(int size) {  
    scrambleLabel.setFont(new Font("Courier New", 0, size));  
}  
  
/**  
 * Scrambles the cube with the specified scramble and shows the scramble in  
 * {@link #scrambleLabel}  
 *  
 * @param scramble  
 *          the scramble to be applied  
 */  
public static void handleScramble(String scramble) {  
    if (!tutorialIsRunning && (incTimer == null) && (inspectionTimer == null) && !customPaintingInProgress  
        && (realTimeSolutionTimer == null)) {  
        currentScramble = scramble;  
        cube.performAbsoluteMoves(currentScramble);  
        scrambleLabel.setText("Scramble: " + currentScramble);  
  
        cubePanel.repaint();  
        // cubePanel.requestFocus();  
    } else  
        return;  
}
```

```
/**  
 * Clears all moves stored by the children of {@link #solveMaster}  
 */  
public static void clearAllSolverMoves() {  
    crossSolver.clearMoves();  
    cornerSolver.clearMoves();  
    edgeSolver.clearMoves();  
    orientationSolver.clearMoves();  
    permutationSolver.clearMoves();  
}  
  
/**  
 * @return a MouseListener for the cube panel so that the clicked piece can  
 *         be solved when in Click-to-Solve mode  
 */  
private static MouseListener getCubePanelMouseListener() {  
    MouseListener cubePanelMouseListener = new MouseListener() {  
        @Override  
        public void mouseClicked(MouseEvent arg0) {  
  
        }  
  
        public void mouseReleased(MouseEvent e) {  
            cubePanel.requestFocus();  
  
            if (clickToSolve) {  
                // Stores the index of the piece selected.  
                int index = MouseSelectionSolver.getIndexOfPieceOnScreen(e.getX(), e.getY());  
                // Stores an integer representing the choice made by the  
                // user in the question dialog.  
                int choice = -1;  
                // Stores the colours of the stickers of the cubie selected.  
                Color[] stickers;  
                String solution = "";  
  
                if (index < 0)  
                    return;  
  
                MenuBar.setSolvePieceSelected(false);  
                clickToSolve = false;  
  
                if (index >= 8)  
                    stickers = cube.getEdge(index - 8).getStickers();  
                else
```

```
stickers = cube.getCorner(index).getStickers();

colorSelection.setAlwaysOnTop(false);
choice = MouseSelectionSolver.getQuestionDialogResponse(String.format(
    "You have selected the %s-%s%s. Do you wish to continue?", 
    Cubie.getColorToWord(stickers[0]), Cubie.getColorToWord(stickers[1]),
    (stickers.length == 3) ? "-" + Cubie.getColorToWord(stickers[2]) + " Corner" : " Edge"));
colorSelection.setAlwaysOnTop(true);

if (choice == 0) {
    selectionSolver.solvePiece(index);
} else
    return;

solution = selectionSolver.getSolution();

timingDetailsTextArea.setText(String.format("Solving the %s-%s%s",
    Cubie.getColorToWord(stickers[0]), Cubie.getColorToWord(stickers[1]),
    (stickers.length == 3) ? "-" + Cubie.getColorToWord(stickers[2]) + " Corner" : " Edge"));

if (!solution.equals(MouseSelectionSolver.BLANK)) {
    timingDetailsTextArea.setText(solution);
    timingDetailsTextArea.setCaretPosition(0);
}

if (MenuBar.paintCustomStateItem.isSelected())
    MenuBar.paintCustomStateItem.doClick();

performRealTimeSolving();
cubePanel.repaint();
} else if (customPaintingInProgress && (realTimeSolutionTimer == null)) {
    // Stores the index of the facelet selected on screen.
    int index = MouseSelectionSolver.getIndexOfFaceletOnScreen(e.getX(), e.getY());
    // Stores the current colour which will be painted on the
    // selected sticker.
    Color currentCustomPaintingColor = colorSelection.getSelectedColor();

    if (index < 0)
        return;

    switch (index) {
    case 0:
        cube.getCorner(0).setSticker(0, currentCustomPaintingColor);
        break;
```

```
case 1:  
    cube.getEdge(0).setSticker(0, currentCustomPaintingColor);  
    break;  
case 2:  
    cube.getCorner(1).setSticker(0, currentCustomPaintingColor);  
    break;  
case 3:  
    cube.getEdge(3).setSticker(0, currentCustomPaintingColor);  
    break;  
case 5:  
    cube.getEdge(1).setSticker(0, currentCustomPaintingColor);  
    break;  
case 6:  
    cube.getCorner(3).setSticker(0, currentCustomPaintingColor);  
    break;  
case 7:  
    cube.getEdge(2).setSticker(0, currentCustomPaintingColor);  
    break;  
case 8:  
    cube.getCorner(2).setSticker(0, currentCustomPaintingColor);  
    break;  
  
case 9:  
    cube.getCorner(3).setSticker(1, currentCustomPaintingColor);  
    break;  
case 10:  
    cube.getEdge(2).setSticker(1, currentCustomPaintingColor);  
    break;  
case 11:  
    cube.getCorner(2).setSticker(2, currentCustomPaintingColor);  
    break;  
case 12:  
    cube.getEdge(7).setSticker(0, currentCustomPaintingColor);  
    break;  
case 14:  
    cube.getEdge(6).setSticker(0, currentCustomPaintingColor);  
    break;  
case 15:  
    cube.getCorner(6).setSticker(2, currentCustomPaintingColor);  
    break;  
case 16:  
    cube.getEdge(10).setSticker(1, currentCustomPaintingColor);  
    break;  
case 17:
```

```
        cube.getCorner(7).setSticker(1, currentCustomPaintingColor);
        break;

    case 18:
        cube.getCorner(2).setSticker(1, currentCustomPaintingColor);
        break;
    case 19:
        cube.getEdge(1).setSticker(1, currentCustomPaintingColor);
        break;
    case 20:
        cube.getCorner(1).setSticker(2, currentCustomPaintingColor);
        break;
    case 21:
        cube.getEdge(6).setSticker(1, currentCustomPaintingColor);
        break;
    case 23:
        cube.getEdge(5).setSticker(1, currentCustomPaintingColor);
        break;
    case 24:
        cube.getCorner(7).setSticker(2, currentCustomPaintingColor);
        break;
    case 25:
        cube.getEdge(11).setSticker(1, currentCustomPaintingColor);
        break;
    case 26:
        cube.getCorner(4).setSticker(1, currentCustomPaintingColor);
        break;
    }

    cubePanel.repaint();
}
}

@Override
public void mouseEntered(MouseEvent arg0) {

}

@Override
public void mouseExited(MouseEvent arg0) {

}

@Override
public void mousePressed(MouseEvent arg0) {
}
```

```
};

return cubePanelMouseListener;
}

/**
 * Sets up the main window
 */
public static void createAndShowGUI() {
    Color bg = null;
    int textSize = 17;

    timeGraph = new TimeGraph("Time Graph", "", "Solve No.", "Time / seconds");
    timeGraph.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    timeGraph.setMinimumSize(new Dimension(400, 400));
    timeGraph.setPreferredSize(new Dimension(600, 450));
    timeGraph.setLocation(new Point(510, 50));
    timeGraph.pack();
    timeGraph.addWindowListener(new PopUpWindowListener());

    timeListPopUp = new TimeListPopUp("Solve Editor");
    timeListPopUp.addWindowListener(new PopUpWindowListener());
    timeListPopUp.pack();

    preferencesPopUp = new Preferences();
    preferencesPopUp.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    preferencesPopUp.addWindowListener(new PopUpWindowListener());

    algorithmDatabasePopUp = new AlgorithmDatabasePopUp();
    algorithmDatabasePopUp.addWindowListener(new PopUpWindowListener());

    solveDatabasePopUp = new SolveDatabasePopUp();
    solveDatabasePopUp.addWindowListener(new PopUpWindowListener());

    competitionDatabasePopUp = new CompetitionDatabasePopUp();
    competitionDatabasePopUp.addWindowListener(new PopUpWindowListener());

    memberDatabasePopUp = new MemberDatabasePopUp();
    memberDatabasePopUp.addWindowListener(new PopUpWindowListener());

    colorSelection = new ColorSelection();
    colorSelection.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    colorSelection.setLocation(600, 150);
    colorSelection.pack();
```

```
colorSelection.addWindowListener(new WindowListener() {  
  
    @Override  
    public void windowActivated(WindowEvent e) {  
    }  
  
    @Override  
    public void windowClosed(WindowEvent e) {  
        MenuBar.paintCustomStateItem.doClick();  
    }  
  
    @Override  
    public void windowClosing(WindowEvent e) {  
    }  
  
    @Override  
    public void windowDeactivated(WindowEvent e) {  
    }  
  
    @Override  
    public void windowDeiconified(WindowEvent e) {  
    }  
  
    @Override  
    public void windowIconified(WindowEvent e) {  
    }  
  
    @Override  
    public void windowOpened(WindowEvent e) {  
    }  
  
});  
  
scramblePopUp = new ScramblePopUp();  
scramblePopUp.setLocation(600, 150);  
scramblePopUp.addWindowListener(new PopUpWindowListener());  
  
totalFrame = new JFrame("Kuubik");  
totalFrame.setIconImage(Main.createImage("res/images/RubikCubeBig.png"));  
totalFrame.setJMenuBar(new MenuBar().createMenuBar());  
// totalFrame.setContentPane(new MenuBar().createContentPane());  
  
cubePanel.setSize(600, 600);
```

```
cubePanel.setLocation(50, 0);
cubePanel.setBackground(bg);
cubePanel.addMouseListener(getCubePanelMouseListener());

totalPaintingPanel = new JPanel();
totalPaintingPanel.setVisible(true);
totalPaintingPanel.setLayout(null);
totalPaintingPanel.setSize(500, 500);
totalPaintingPanel.setBackground(bg);
totalFrame.getContentPane().add(totalPaintingPanel);

topComponentsPanel = new JPanel();
topComponentsPanel.setLayout(new GridLayout(2, 1, 0, -11)); // BoxLayout(topComponentsPanel,
                                                       // BoxLayout.PAGE_AXIS));
totalFrame.getContentPane().add(topComponentsPanel, BorderLayout.NORTH);

scrambleLabel = new JTextArea("Scramble:");
scrambleLabel.setEditable(false);
scrambleLabel.setLineWrap(true);
scrambleLabel.setWrapStyleWord(true);
scrambleLabel.setFont(new Font("Courier New", 0, preferencesPopUp.getScrambleTextSize()));
scrambleLabel.setBackground(bg);
topComponentsPanel.add(scrambleLabel);

timeLabel = new JLabel("0.00");
timeLabel.setFont(new Font("Arial", 0, 50));
// totalFrame.getContentPane().add(timeLabel, BorderLayout.NORTH);
topComponentsPanel.add(timeLabel);

initListPanel();
timeListScrollPane.setPreferredSize(new Dimension(150, 360));
totalFrame.getContentPane().add(timeListPanel, BorderLayout.EAST);

/********************* BOTTOM PANELS *****/
bottomPanel = new JPanel();
bottomPanel.setLayout(new BoxLayout(bottomPanel, BoxLayout.PAGE_AXIS));

tutorialComponentsPanel = new JPanel();
tutorialButtonsPanel = new JPanel();
tutorialTextArea = new JTextArea();
tutorialScrollPane = new JScrollPane(tutorialTextArea);
tutorialNextButton = new JButton();
tutorialBackButton = new JButton();
```

```
tutorialHintButton = new JButton();
tutorialResetStateButton = new JButton();
tutorialShowDescriptionButton = new JButton();
tutorialShowSolutionButton = new JButton();
tutorialComponentsPanel.setVisible(false);

tutorialComponentsPanel.setLayout(new BoxLayout(tutorialComponentsPanel, BoxLayout.PAGE_AXIS));
tutorialComponentsPanel.setPreferredSize(new Dimension(800, 190));
tutorialComponentsPanel.setBorder(BorderFactory.createEtchedBorder(EtchedBorder.LOWERED));

tutorialTextArea.setLineWrap(true);
tutorialTextArea.setWrapStyleWord(true);
tutorialTextArea.setEditable(false);
tutorialTextArea.setBorder(BorderFactory.createLineBorder(Color.black));
tutorialTextArea.setFont(new Font("Arial", Font.PLAIN, textSize));

tutorialScrollPane.setPreferredSize(new Dimension(700, 150));
tutorialScrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

tutorialNextButton.setSize(150, 20);
tutorialNextButton.setText("Next");
tutorialNextButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        if (tutorial.isLoaded()) {
            resetCube();
            solveMaster.rotateToTopFront(Color.white, Color.green);
            solveMaster.clearMoves();

            trackingMoves.clear();

            tutorial.loadNextSubTutorial();
            tutorialTextArea.setText(tutorial.getDescription());
            tutorialTextArea setCaretPosition(0);
            cube.performAbsoluteMoves(tutorial.getScramble());
            cubePanel.repaint();
            cubePanel.requestFocus();
            movesAllowed = tutorial.requiresUserAction();
            movesToBeRecorded = movesAllowed;
            tutorialBackButton.setEnabled(!tutorial.isFirstSubTutorial());
            tutorialNextButton.setEnabled(!tutorial.isLastSubTutorial());
            tutorialHintButton.setEnabled(tutorial.requiresUserAction());
            tutorialShowSolutionButton.setEnabled(tutorial.requiresUserAction());
        }
    }
})
```

```
        }

    });

tutorialBackButton.setSize(150, 20);
tutorialBackButton.setText("Back");
tutorialBackButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        if (tutorial.isLoaded()) {
            resetCube();
            solveMaster.rotateToTopFront(Color.white, Color.green);
            solveMaster.clearMoves();
            trackingMoves.clear();

            tutorial.loadPreviousSubTutorial();
            tutorialTextArea.setText(tutorial.getDescription());
            tutorialTextArea.setCaretPosition(0);
            cube.performAbsoluteMoves(tutorial.getScramble());
            cubePanel.repaint();
            cubePanel.requestFocus();
            movesAllowed = tutorial.requiresUserAction();
            tutorialBackButton.setEnabled(!tutorial.isFirstSubTutorial());
            tutorialNextButton.setEnabled(!tutorial.isLastSubTutorial());
            tutorialHintButton.setEnabled(tutorial.requiresUserAction());
            tutorialShowSolutionButton.setEnabled(tutorial.requiresUserAction());
        }
    }
});

tutorialShowDescriptionButton.setSize(150, 20);
tutorialShowDescriptionButton.setText("Description");
tutorialShowDescriptionButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        if (tutorial.isLoaded()) {
            tutorialTextArea.setText(tutorial.getDescription());
            tutorialTextArea.setCaretPosition(0);
            cubePanel.requestFocus();
        }
    }
});

tutorialHintButton.setSize(150, 20);
tutorialHintButton.setText("Hint");
```

```
tutorialHintButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        if (tutorial.isLoaded() && (tutorial.requiresUserAction())) {
            // solveCube();
            // resetCube();
            // solveMaster.rotateToTopFront(Color.white, Color.green);
            // solveMaster.clearMoves();

            tutorial.loadNextHint();
            tutorialTextArea.setText(tutorial.getHint());
            tutorialTextArea.setCaretPosition(0);
            cubePanel.requestFocus();
        }
    }
});

tutorialResetStateButton.setSize(150, 20);
tutorialResetStateButton.setText("Reset");
tutorialResetStateButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        if (tutorial.isLoaded() && (tutorial.requiresUserAction())) {
            // solveCube();
            resetCube();
            solveMaster.rotateToTopFront(Color.white, Color.green);
            solveMaster.clearMoves();

            cube.performAbsoluteMoves(tutorial.getScramble());
            cubePanel.requestFocus();
            cubePanel.repaint();
        }
    }
});

tutorialShowSolutionButton.setSize(150, 20);
tutorialShowSolutionButton.setText("Solution");
tutorialShowSolutionButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        if (tutorial.isLoaded() && (tutorial.requiresUserAction())) {
            resetCube();
            solveMaster.rotateToTopFront(Color.white, Color.green);
            solveMaster.clearMoves();
```

```
movesAllowed = false;

cube.performAbsoluteMoves(tutorial.getScramble());
cube.performAbsoluteMoves(tutorial.getOptimalSolutions()[0]);

String[] optimalSolutions = tutorial.getOptimalSolutions();
tutorialTextArea.setText("Solution: " + tutorial.getExplanation()
+ "\n\nOptimal solutions include:");

for (int i = 0; i < optimalSolutions.length; ++i)
    tutorialTextArea.setText(tutorialTextArea.getText() + "\n" + optimalSolutions[i]);

tutorialTextArea.setCaretPosition(0);

cubePanel.repaint();
cubePanel.requestFocus();
}
}

tutorialButtonsPanel.add(tutorialShowDescriptionButton);
tutorialButtonsPanel.add(tutorialHintButton);
tutorialButtonsPanel.add(tutorialResetStateButton);
tutorialButtonsPanel.add(tutorialShowSolutionButton);
tutorialButtonsPanel.add(tutorialBackButton);
tutorialButtonsPanel.add(tutorialNextButton);
tutorialComponentsPanel.add(tutorialScrollPane);
tutorialComponentsPanel.add(tutorialButtonsPanel);

// tutorialComponentsPanel.add(tutorialScrollPane); //Looks like these
// are redundant
// tutorialComponentsPanel.add(tutorialButtonsPanel);

timingDetailsComponentsPanel = new JPanel();
timingDetailsButtonPanel = new JPanel();
timingDetailsTextArea = new JTextArea();
timingDetailsScrollPane = new JScrollPane(timingDetailsTextArea);
timingDetailsResetSessionButton = new JButton();
timingDetailsStartNewSolveButton = new JButton();
resetCubeButton = new JButton("Reset Cube");
timingDetailsComponentsPanel.setVisible(true);

timingDetailsComponentsPanel.setLayout(new BoxLayout(timingDetailsComponentsPanel, BoxLayout.PAGE_AXIS));
```

```
timingDetailsComponentsPanel.setPreferredSize(new Dimension(800, 190));
timingDetailsComponentsPanel.setBorder(BorderFactory.createEtchedBorder(EtchedBorder.LOWERED));

timingDetailsTextArea.setLineWrap(true);
timingDetailsTextArea.setWrapStyleWord(true);
timingDetailsTextArea.setEditable(false);
timingDetailsTextArea.setBorder(BorderFactory.createLineBorder(Color.black));
timingDetailsTextArea.setFont(new Font("Arial", Font.PLAIN, textSize));

timingDetailsScrollPane.setPreferredSize(new Dimension(700, 150));
timingDetailsScrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

timingDetailsResetSessionButton.setSize(150, 20);
timingDetailsResetSessionButton.setText("Reset Times");
timingDetailsResetSessionButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        solves.clear();
        copyAllTimesToDisplay();
        refreshTimeGraph(true);
        refreshStatistics();
    }
});

timingDetailsStartNewSolveButton.setSize(150, 20);
timingDetailsStartNewSolveButton.setText("Start New Solve");
timingDetailsStartNewSolveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (((customPaintingInProgress) || (tutorialIsRunning))
            || ((inspectionTimer != null) && (inspectionTimer.isRunning())))
            || ((incTimer != null) && (incTimer.isRunning()))
            || ((realTimeSolutionTimer != null) && (realTimeSolutionTimer.isRunning())))
        {
            cubePanel.requestFocus();
            return;
        }

        if (!isValidCubeState(false)) {
            int choice = MouseSelectionSolver
                .getQuestionDialogResponse("The cube is not in a valid state, do you want to reset it?");

            if (choice == 0)
                resetCube();
            else
                return;
        }
    }
});
```

```
if (MenuBar.isUsingScramblesInList()) {
    try {
        currentScramble = scramblePopUp.getCurrentScramble();
    } catch (IndexOutOfBoundsException scrambleListIsEmpty) {
        JOptionPane.showMessageDialog(totalFrame, "No scrambles in scramble list", "Error",
            JOptionPane.ERROR_MESSAGE);
        return;
    }
} else {
    currentScramble = Scramble.generateScramble();
}

movesAllowed = false;
timerHasPermissionToStart = true;
movesToBeRecorded = true;
cubeSolved = false;

inspectionTimeRemaining = preferencesPopUp.getInspectionTime();
inspectionTimer = new Timer(1000, new InspectionTimerListener());
MenuBar.setSolvePieceSelected(false);
inspectionTimer.start();
currentPenalty = "0";

timeLabel.setForeground(Color.red);
timeLabel.setText(" " + inspectionTimeRemaining);

scrambleLabel.setText("Scramble: " + currentScramble);
resetCube();
solveMaster.rotateToTopFront(Color.white, Color.green);
solveMaster.clearMoves();
trackingMoves.clear();

cube.performAbsoluteMoves(currentScramble);
cubePanel.repaint();
cubePanel.requestFocus();
}
});
timingDetailsStartNewSolveButton.setMnemonic(KeyEvent.VK_1);
timingDetailsStartNewSolveButton.setToolTipText("Alt + 1");

resetCubeButton.setSize(150, 20);
resetCubeButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
```

```
cancelSolve();
resetCube();
refreshStatistics();
cubePanel.repaint();
cubePanel.requestFocus();
}
});

timingDetailsButtonPanel.add(timingDetailsResetSessionButton);
timingDetailsButtonPanel.add(timingDetailsStartNewSolveButton);
timingDetailsButtonPanel.add(resetCubeButton);

timingDetailsComponentsPanel.add(timingDetailsScrollPane);
timingDetailsComponentsPanel.add(timingDetailsButtonPanel);

// totalFrame.getContentPane().add(tutorialComponentsPanel,
// BorderLayout.SOUTH);
// totalFrame.getContentPane().add(timingDetailsPanel,
// BorderLayout.SOUTH);

bottomPanel.add(tutorialComponentsPanel);
bottomPanel.add(timingDetailsComponentsPanel);

totalFrame.add(bottomPanel, BorderLayout.SOUTH);

/********************* BOTTOM PANELS ********************/

totalPaintingPanel.add(cubePanel);
// Display the window.
totalFrame.pack();
totalFrame.setExtendedState(totalFrame.getExtendedState() | JFrame.MAXIMIZED_BOTH);
totalFrame.setVisible(true);
totalFrame.setResizable(true);
totalFrame.setMinimumSize(new Dimension(500, 500));
totalFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

refreshStatistics();

JOptionPane.showMessageDialog(totalFrame, "Make sure you backup your data regularly", "Tip",
JOptionPane.INFORMATION_MESSAGE);

}

/**
```

```
* Runs the program
*
* @param s
*        runtime argument
*/
public static void main(String s[]) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}

/*
 * JFrame f = new JFrame("ShapesDemo2D");
 *
 * final Component applet = new CubeDrawingTest(); applet.setSize(600,
 * 600);
 *
 * JPanel totalGUI = new JPanel(); totalGUI.setLayout(null);
 * totalGUI.addKeyListener(new KeyListener() {
 *
 * @Override public void keyPressed(KeyEvent arg0) {
 *
 * }
 *
 * @Override public void keyReleased(KeyEvent arg0) {
 * MyColors.changeColour(); applet.repaint(); }
 *
 * @Override public void keyTyped(KeyEvent arg0) {
 *
 * } });
 *
 * f.getContentPane().add("Center", totalGUI); totalGUI.add("Center",
 * applet);
 *
 *
 * JButton changeColourButton = new JButton("Shuffle");
 * changeColourButton.setSize(120, 30);
 * changeColourButton.setLocation(0,0);
 * changeColourButton.addActionListener(new ActionListener() {
 *
 * @Override public void actionPerformed(ActionEvent arg0) {
 * MyColors.changeColour(); applet.repaint(); }
 *
```

```
* }); totalGUI.add(changeColourButton);
*
* f.pack(); f.setExtendedState(f.getExtendedState() |
* JFrame.MAXIMIZED_BOTH); f.setVisible(true); f.setMinimumSize(new
* Dimension(700, 450));
* f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
*/
}

}
```

## Class Member

```
package jCube;

/**
 * @author Kelsey McKenna
 */
public class Member {
    /**
     * Stores the ID of the member
     */
    private int memberID;
    /**
     * Stores the forenames of the member
     */
    private String forenames;
    /**
     * Stores the surname of the member
     */
    private String surname;
    /**
     * Stores the gender of the member
     */
    private String gender;
    /**
     * Stores the date of birth of the member
     */
    private String dateOfBirth;
    /**
     * Stores an email address belonging to the member
     */
    private String email;
    /**
     * Stores the form class to which the member belongs
     */
    private String formClass;

    /**
     * Constructor - assigns values to the fields
     *
     * @param memberID
     *          the ID of the member
     * @param forenames
     *          the forenames of the member
     */
```

```
* @param surname
*      the surname of the member
* @param gender
*      the gender of the member ("male" or "female")
* @param dateOfBirth
*      the date of birth of the member (in form 'dd/MM/YYYY')
* @param email
*      the email address of the member
* @param formClass
*      the form class to which the member belongs
*/
public Member(int memberID, String forenames, String surname, String gender, String dateOfBirth, String email,
String formClass) {
super();
this.memberID = memberID;
this.forenames = forenames;
this.surname = surname;
this.gender = gender;
this.dateOfBirth = dateOfBirth;
this.email = email;
this.formClass = formClass;
}

/**
 * @return the ID of the member
 */
public int getMemberID() {
return memberID;
}

/**
 * Assigns the specified ID to the member
 *
 * @param memberID
 *      the new ID to be assigned to the member
 */
public void setMemberID(int memberID) {
this.memberID = memberID;
}

/**
 * @return the forenames of the member
 */
public String getForenames() {
```

```
        return forenames;
    }

    /**
     * Assigns the specified forenames to the member
     *
     * @param forenames
     *          the new forenames to be assigned to the member
     */
    public void setForenames(String forenames) {
        this.forenames = forenames;
    }

    /**
     * @return the surname of the member
     */
    public String getSurname() {
        return surname;
    }

    /**
     * Assigns the specified surname to the member
     *
     * @param surname
     *          the new surname to be assigned to the member
     */
    public void setSurname(String surname) {
        this.surname = surname;
    }

    /**
     * @return the gender of the member - <b>"male"</b> or <b>"female"</b>
     */
    public String getGender() {
        return gender;
    }

    /**
     * Assigns the specified gender to the member
     *
     * @param gender
     *          the new gender to be assigned to the member - <b>"male"</b> or
     *          <b>"female"</b>
     */
}
```

```
public void setGender(String gender) {
    this.gender = gender;
}

/**
 * @return the date of birth of the member
 */
public String getDateOfBirth() {
    return dateOfBirth;
}

/**
 * Assigns the specified date of birth to the member
 *
 * @param dateOfBirth
 *          the new date of birth for the member
 */
public void setDateOfBirth(String dateOfBirth) {
    this.dateOfBirth = dateOfBirth;
}

/**
 * @return the email address of the member
 */
public String getEmail() {
    return email;
}

/**
 * Assigns the specified email to the member
 *
 * @param email
 *          the new email address to be assigned to the member
 */
public void setEmail(String email) {
    this.email = email;
}

/**
 * @return the form class to which the member belongs
 */
public String getFormClass() {
    return formClass;
}
```

```
/**  
 * Assigns the specified form class to the member  
 *  
 * @param formClass  
 *         the new form class to be assigned to the member  
 */  
public void setFormClass(String formClass) {  
    this.formClass = formClass;  
}  
  
/**  
 * Determines whether the specified email is in a valid format or not. This  
 * does not verify that the address exists  
 *  
 * @param email  
 *         the email to be analysed  
 * @return <b>true</b> if the email address is valid; <br>  
 *         <b>false</b> otherwise  
 */  
public static boolean isValidEmail(String email) {  
    TextFile cFile = new TextFile();  
    String regex = "";  
    try {  
        cFile.setFilePath("res/RFC822.txt");  
        cFile.setIO(TextFile.READ);  
        regex = cFile.readLine();  
        cFile.close();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
  
    return email.matches(regex);  
}  
  
/**  
 * Determines whether the specified form class is valid or not. Valid form  
 * classes are of the form: <br>  
 * 08 | M <br>  
 * 09 | R <br>  
 * 10 | S <br>  
 * 11 | T <br>  
 * 12 | W <br>  
 * 13 | <br>
```

```
* 14 | <br>
* Leading zeros are ignored
*
* @param formClass
*         the form class to be analysed
* @return <b>true</b> if the form class is valid; <br>
*         <b>false</b> otherwise
*/
public static boolean isValidFormClass(String formClass) {
    return formClass.matches("0*(|8|9|10|11|12|13|14) [MRSTW]");
}
```

## Class MemberCompetition

```
package jCube;

/**
 * @author Kelsey McKenna
 */
public class MemberCompetition {

    /**
     * Stores the competition ID of the record
     */
    private int competitionID;
    /**
     * Stores the member ID of the record
     */
    private int memberID;
    /**
     * This stores the 5 times of the average of the record.
     */
    private String[] times;

    /**
     * Constructor - Assigns values to fields
     *
     * @param competitionID
     *         the ID of the competition
     * @param memberID
     *         the ID of the member
     * @param times
     *         an array of the 5 times recorded by the member in the
     *         competition
     */
    public MemberCompetition(int competitionID, int memberID, String[] times) {
        this.competitionID = competitionID;
        this.memberID = memberID;
        this.times = times;
    }

    /**
     * Constructor - Assigns values to fields
     *
     * @param competitionID
     *         the ID of the competition
     */
```

```
* @param memberID
*      the ID of the member
* @param time1
*      the first time of the average
* @param time2
*      the second time of the average
* @param time3
*      the third time of the average
* @param time4
*      the fourth time of the average
* @param time5
*      the fifth time of the average
*/
public MemberCompetition(int competitionID, int memberID, String time1, String time2, String time3, String time4,
    String time5) {
    this.competitionID = competitionID;
    this.memberID = memberID;

    times = new String[] { time1, time2, time3, time4, time5 };
}

/**
 * @return the ID of the competition
 */
public int getCompetitionID() {
    return competitionID;
}

/**
 * Assigns an ID to the competition
 *
 * @param competitionID
 *      the new ID of the competition
 */
public void setCompetitionID(int competitionID) {
    this.competitionID = competitionID;
}

/**
 * @return the ID of the member
 */
public int getMemberID() {
    return memberID;
}
```

```
/**  
 * Assigns an ID to the member  
 *  
 * @param memberID  
 *         the ID of the member  
 */  
public void setMemberID(int memberID) {  
    this.memberID = memberID;  
}  
  
/**  
 * @return an array of Strings representing the times of the average  
 */  
public String[] getTimes() {  
    return times;  
}  
  
/**  
 * Assigns the times of the average  
 *  
 * @param times  
 *         the times of the average  
 */  
public void setTimes(String[] times) {  
    this.times = times;  
}  
  
/**  
 * @return the calculated WCA average of the 5 times  
 */  
public double getAverage() {  
    return Statistics.getAverageOf(5, times);  
}  
  
/**  
 * @return the calculated WCA average of the 5 times to 2 decimal places  
 */  
public double get2DPAverage() {  
    return Double.valueOf(String.format("%.02f", getAverage()));  
}  
  
/**  
 * @return the 5 times of the average in a numerical representation
```

```
/*
public double[] getNumericTimeArray() {
    double[] timesToReturn = new double[5];

    for (int i = 0; i < 5; ++i)
        timesToReturn[i] = Solve.getFormattedStringToDouble(this.times[i]);

    return timesToReturn;
}

/**
 * This method compares <b>this</b> average to another average. An average
 * is better than another average if the 'average of 5' is faster, or the
 * fastest time of <b>this</b> is faster than the fastest time of
 * <b>other</b>. If the fastest times are the same, then the second fastest
 * times are compared in the same way. If all times are the same, then
 * <b>other</b> will be assumed to be the better average
 *
 * @param other
 *         the other MemberCompetition to which this MemberCompetition is
 *         compared
 * @return <b>true</b> if <b>this</b> is better than <b>other</b>; <br>
 *         <b>false</b> otherwise
 */
public boolean isBetterThan(MemberCompetition other) {
    /*
     * Stores the numerical representation of the average. For example, if
     * the times of this average were 12.00, 13.00, 14.00, 15.00, 16.00,
     * then 'thisAverage' would store 14.0
     */
    double thisAverage = get2DPAverage();
    double otherAverage = other.get2DPAverage();

    // If one of the average is DNF, then set it to infinity
    if (thisAverage == -1)
        thisAverage = 1e10;
    if (otherAverage == -1)
        otherAverage = 1e10;

    if (thisAverage < otherAverage)
        return true;
    else if (otherAverage < thisAverage)
        return false;
```

```
/*
 * Stores the times of the average in their numerical representation.
 * For example, if the times were {12.00, 1:10.50, 55.23, 2:00.95,
 * 1:46.58} then the list would store {12.0, 70.5, 55.23, 120.95,
 * 106.58}
 */
double[] list1 = this.getNumericTimeArray();
double[] list2 = other.getNumericTimeArray();

// If a time is DNF, then set it to infinity
for (int i = 0; i < 5; ++i) {
    if (list1[i] == -1)
        list1[i] = 1e10;
    if (list2[i] == -1)
        list2[i] = 1e10;
}

Sorter.quickSort(list1);
Sorter.quickSort(list2);

/*
 * i.e. if the fastest time of the first average is better than the
 * second average, then return true, or vice-versa return false. If the
 * fastest time of each is the same, the second times are compared etc.
 * until they are different.
 */
for (int i = 0; i < 5; ++i) {
    if (list1[i] < list2[i])
        return true;
    else if (list2[i] < list1[i])
        return false;
}

/*
 * If this point is reached, then the averages are exactly the same, so
 * just return false.
 */
return false;
}
```

## Class MemberCompetitionDatabaseConnection

```
package jCube;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * @author Kelsey McKenna
 */
public class MemberCompetitionDatabaseConnection {
    /**
     * The try/catch block is invoked if the table does not exist or the query
     * is invalid
     *
     * @param query
     *         the SQLite query to be performed on 'memberCompetition' table
     * @return an array of MemberCompetitions representing the result of the
     *         specified query
     * @throws SQLException
     *         if the query is invalid
     * @throws ClassNotFoundException
     *         if SQLite classes are missing
     */
    public static MemberCompetition[] executeQuery(String query) throws SQLException, ClassNotFoundException {
        MemberCompetition[] memberCompetitions = null;

        try {
            memberCompetitions = executeSafeQuery(query);
        } catch (SQLException e) {
            initTable();
            memberCompetitions = executeSafeQuery(query);
        }

        return memberCompetitions;
    }

    /**
     * @param query
     *         the SQLite query to be performed on 'memberCompetition' table
     * @return an array of MemberCompetitions representing the result of the
     */

}
```

```
*      specified query
* @throws ClassNotFoundException
*         if SQLite classes are missing
* @throws SQLException
*         if the table does not exist or the query is invalid
*/
private static MemberCompetition[] executeSafeQuery(String query) throws ClassNotFoundException, SQLException { // table
    // initialised
    // etc.

    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement;
    ResultSet rs;
    MemberCompetition[] memberCompetitions;
    int numRecords = 0;

    connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
    statement = connection.createStatement();
    statement.setQueryTimeout(30); // set timeout to 30 sec.

    rs = statement.executeQuery(query);

    while (rs.next())
        ++numRecords;

    rs.close();
    rs = statement.executeQuery(query);

    memberCompetitions = new MemberCompetition[numRecords];

    for (int i = 0; i < numRecords; ++i) {
        rs.next();
        memberCompetitions[i] = new MemberCompetition(rs.getInt("competitionID"), rs.getInt("memberID"),
            rs.getString("time1"), rs.getString("time2"), rs.getString("time3"), rs.getString("time4"),
            rs.getString("time5"));
    }

    try {
        if (connection != null)
            connection.close();
    } catch (SQLException e) {
    }

    return memberCompetitions;
}
```

```
}

/**
 * Executes the specified update on the 'memberCompetition' table
 *
 * @param update
 *          the update to be performed on the table
 * @throws ClassNotFoundException
 *          if SQLite classes are missing
 * @throws SQLException
 *          if the query is invalid
 */
public static void executeUpdate(String update) throws ClassNotFoundException, SQLException {
    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement = null;

    connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
    statement = connection.createStatement();
    statement.setQueryTimeout(30); // set timeout to 30 sec.

    statement.executeUpdate(update);

    try {
        if (connection != null)
            connection.close();
    } catch (SQLException e2) {
    }
}

/**
 * Initialises the table in the database. This method will be called if the
 * executeQuery(...) method cannot find the table in the database.
 *
 * @throws ClassNotFoundException
 *          if SQLite classes are missing
 */
private static void initTable() throws ClassNotFoundException {
    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement = null;

    try {
        connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
```

```
statement = connection.createStatement();
statement.setQueryTimeout(30); // set timeout to 30 sec.

statement.executeUpdate("DROP TABLE IF EXISTS memberCompetition");
statement.executeUpdate("CREATE TABLE memberCompetition(" + "competitionID INTEGER, "
    + "memberID INTEGER, " + "time1 TEXT, " + "time2 TEXT, " + "time3 TEXT, " + "time4 TEXT, "
    + "time5 TEXT, " + "FOREIGN KEY (competitionID) REFERENCES competition(competitionID), "
    + "FOREIGN KEY (memberID) REFERENCES member(memberID) " + ");");

} catch (SQLException e) {
    try {
        if (connection != null)
            connection.close();
    } catch (SQLException e2) {
    }
}
}

}
```

## Class MemberCompetitionDatabasePopUp

```
package jCube;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import java.sql.SQLException;

import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.DefaultTableModel;

/**
 * @author Kelsey McKenna
 */
public class MemberCompetitionDatabasePopUp extends JFrame implements KeyListener {
    /**
     * Default serialVersionUID
     */
    private static final long serialVersionUID = 1L;
    /**
     * This stores the initial width of the Member-Competition window.
     */
    private static final int WIDTH = 700;
```

```
/**  
 * This stores the padding of the elements in the window. The greater the  
 * padding, the further towards the centre of the window the elements will  
 * be.  
 */  
private final int pad = 10;  
/**  
 * This indicates the vertical spacing between the text boxes etc. in the  
 * window.  
 */  
private final int fieldYSpacing = 50;  
/**  
 * This indicates the vertical spacing between the buttons etc. in the  
 * window.  
 */  
private final int buttonYSpacing = 40;  
  
/**  
 * This variable keeps track of the current y position of the last element  
 * placed in the window.  
 */  
private int y = 0;  
/**  
 * Stores the font to be used in the text fields  
 */  
private Font fieldFont = new Font("Arial", 0, 25);  
/**  
 * If this variable is true, it indicates that a record is being edited,  
 * otherwise a record is being added.  
 */  
private boolean editing = false;  
  
/**  
 * Stores the competition ID of the selected competition.  
 */  
private int currentCompetitionID;  
/**  
 * After choosing to edit a record in the table, the index of this row in  
 * the table is stored in this variable.  
 */  
private int selectedMCIndex;  
/**
```

```
* This panel is used to store the table.  
*/  
private JPanel memberCompetitionListPanel;  
/**  
 * memberCompetitionTable is placed 'inside' this variable so that when the  
 * size of the table exceeds the size of the window, the user can scroll in  
 * order to view the rest of the table.  
 */  
private JScrollPane tableContainer;  
/**  
 * This is the table that is displayed in the window; it stores the contents  
 * of the table and the rendering features required to display the data.  
 */  
private final JTable memberCompetitionTable;  
/**  
 * This variable can be customised so that certain cells of the table are  
 * uneditable and the columns of the table can be given text. This variable  
 * is then set as the model of memberCompetitionTable.  
 */  
private final DefaultTableModel model;  
/**  
 * The buttons in the window are placed in this panel.  
 */  
private JPanel buttonPanel;  
/**  
 * Clicking this button opens the Member-Competition Form window.  
 */  
private JButton addRecordButton;  
/**  
 * Clicking this button opens the Member-Competition Form window if a row  
 * has been selected.  
 */  
private JButton editRecordButton;  
/**  
 * Clicking this button deletes the selected rows from the table.  
 */  
private JButton deleteRecordButton;  
/**  
 * This label is shown at the bottom left of the window and indicates the  
 * competition ID of the selected competition.  
 */  
private JLabel competitionIndicatorLabel;
```

```
/**  
 * This label is shown in the Member-Competition Form window with the text  
 * 'Member ID'.  
 */  
private JLabel memberIDLabel;  
/**  
 * This label is shown in the Member-Competition Form window with the text  
 * 'Time 1'.  
 */  
private JLabel time1Label;  
/**  
 * This label is shown in the Member-Competition Form window with the text  
 * 'Time 2'.  
 */  
private JLabel time2Label;  
/**  
 * This label is shown in the Member-Competition Form window with the text  
 * 'Time 3'.  
 */  
private JLabel time3Label;  
/**  
 * This label is shown in the Member-Competition Form window with the text  
 * 'Time 4'.  
 */  
private JLabel time4Label;  
/**  
 * This label is shown in the Member-Competition Form window with the text  
 * 'Time 5'.  
 */  
private JLabel time5Label;  
/**  
 * This label is shown in the Member-Competition Form window when the user  
 * enters invalid data.  
 */  
private JLabel errorMessageLabel;  
/**  
 * This drop-down list stores the member IDs in the Member-Competition Form  
 * window.  
 */  
private JComboBox<Integer> memberIDComboBox;  
/**  
 * This field is shown in the Member-Competition Form window and the user  
 * can enter a time into this field.  
 */
```

```
/*
private JTextField time1Field;
/**
 * This field is shown in the Member-Competition Form window and the user
 * can enter a time into this field.
*/
private JTextField time2Field;
/**
 * This field is shown in the Member-Competition Form window and the user
 * can enter a time into this field.
*/
private JTextField time3Field;
/**
 * This field is shown in the Member-Competition Form window and the user
 * can enter a time into this field.
*/
private JTextField time4Field;
/**
 * This field is shown in the Member-Competition Form window and the user
 * can enter a time into this field.
*/
private JTextField time5Field;

/**
 * This represents the Member-Competition Form window.
*/
private JFrame memberCompetitionInputForm;

/**
 * Clicking this button submits the data in the Member-Competition Form
 * window for validation.
*/
private JButton submitButton;

/**
 * This panel stores the elements of the Member-Competition Form window.
*/
private JPanel contentPane;

/**
 * Constructor - sets up the 'Member-Competition Table' window
*/
public MemberCompetitionDatabasePopUp() {
    super("Member-Competition");
```

```
setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
setLayout(new BorderLayout());
setPreferredSize(new Dimension(WIDTH, 400));
setVisible(false);
addWindowListener(new WindowListener() {

    @Override
    public void windowOpened(WindowEvent arg0) {
    }

    @Override
    public void windowIconified(WindowEvent arg0) {
    }

    @Override
    public void windowDeiconified(WindowEvent arg0) {
    }

    @Override
    public void windowDeactivated(WindowEvent arg0) {
    }

    @Override
    public void windowClosing(WindowEvent arg0) {
    }

    @Override
    public void windowClosed(WindowEvent arg0) {
        memberCompetitionInputForm.setVisible(false);
    }

    @Override
    public void windowActivated(WindowEvent arg0) {
    }
});
setUpMemberCompetitionInputForm();

memberCompetitionListPanel = new JPanel();
memberCompetitionListPanel.setOpaque(true);

final String[] columnNames = { "Name", "Member ID", "Rank", "Average", "Time 1", "Time 2", "Time 3", "Time 4",
    "Time 5" };
```

```
model = new DefaultTableModel() {
    private static final long serialVersionUID = 1L;

    public int getColumnCount() {
        return columnNames.length;
    }

    public String getColumnName(int col) {
        return columnNames[col];
    }

    public boolean isCellEditable(int row, int col) {
        return false;
    }
};

model.addTableModelListener(new TableModelListener() {
    @Override
    public void tableChanged(TableModelEvent e) {
    }
});

memberCompetitionTable = new JTable();
memberCompetitionTable.setModel(model);
memberCompetitionTable.setColumnSelectionAllowed(false);
memberCompetitionTable.setPreferredScrollableViewportSize(new Dimension(WIDTH, 70));
memberCompetitionTable.setFillsViewportHeight(true);
memberCompetitionTable.setAutoScrolls(true);
memberCompetitionTable.getTableHeader().setReorderingAllowed(false);
memberCompetitionTable.setFont(new Font("Arial", 0, 15));
memberCompetitionTable.setRowHeight(20);
memberCompetitionTable.addMouseListener(new MouseListener() {

    @Override
    public void mouseClicked(MouseEvent arg0) {
        if (arg0.getClickCount() == 2) {
            editRecordFunction();
        }
    }

    @Override
    public void mouseEntered(MouseEvent arg0) {
    }
})
```

```
@Override
public void mouseExited(MouseEvent arg0) {
}

@Override
public void mousePressed(MouseEvent arg0) {
}

@Override
public void mouseReleased(MouseEvent arg0) {
}

});

buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(1, 3));
buttonPanel.setPreferredSize(new Dimension(WIDTH, 40));
buttonPanel.setSize(WIDTH, 50);

addRecordButton = new JButton("Add");
addRecordButton.setFocusable(false);
addRecordButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        memberCompetitionTable.clearSelection();

        editing = false;
        resetMemberIDComboBoxItems();
        if (memberIDComboBox.getItemCount() == 0) {
            memberCompetitionInputForm.setVisible(false);

            JOptionPane.showMessageDialog(memberCompetitionListPanel, "No existing members remaining", "Error",
                JOptionPane.ERROR_MESSAGE);
        } else {
            time1Field.setText("");
            time2Field.setText("");
            time3Field.setText("");
            time4Field.setText("");
            time5Field.setText("");
            memberCompetitionInputForm.setVisible(true);
        }
    }
});

editRecordButton = new JButton("Edit");
```

```
editRecordButton.setFocusable(false);
editRecordButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        editRecordFunction();
    }
});

deleteRecordButton = new JButton("Delete");
deleteRecordButton.setFocusable(false);
deleteRecordButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        try {
            Object[] options = { "Yes", "No" };
            int choice = -1;

            int selectedIndex = memberCompetitionTable.getSelectedRow();

            if (selectedIndex == -1)
                return;

            choice = JOptionPane.showOptionDialog(null, "Are you sure you want to delete?", "Warning",
                JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[1]);

            if (choice == 0)
                deleteRow();
        } catch (ClassNotFoundException | SQLException e) {
            JOptionPane.showMessageDialog(memberCompetitionListPanel, "Unable to delete record from database",
                "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
});

competitionIndicatorLabel = new JLabel();
competitionIndicatorLabel.setHorizontalAlignment(JLabel.CENTER);
competitionIndicatorLabel.setFont(new Font("Arial", 0, 15));

buttonPanel.add(competitionIndicatorLabel);
buttonPanel.add(addRecordButton);
buttonPanel.add(editRecordButton);
buttonPanel.add(deleteRecordButton);

tableContainer = new JScrollPane(memberCompetitionTable);
tableContainer.setViewportView(memberCompetitionTable, null);
```

```
Main.resizeColumnWidths(memberCompetitionTable);
populateCellsWithDatabaseData();

add(tableContainer, null);
add(buttonPanel, BorderLayout.SOUTH);
pack();
}

/**
 * Performs the operations required to set up the 'Member-Competition Form'
 * window with the contents of the selected row in the table and indicates
 * that the row is being edited (and not added)
 */
private void editRecordFunction() {
    int selectedRow = memberCompetitionTable.getSelectedRow();

    if (selectedRow != -1) {
        editing = true;
        time1Field.setText("'" + memberCompetitionTable.getValueAt(selectedRow, 4));
        time2Field.setText("'" + memberCompetitionTable.getValueAt(selectedRow, 5));
        time3Field.setText("'" + memberCompetitionTable.getValueAt(selectedRow, 6));
        time4Field.setText("'" + memberCompetitionTable.getValueAt(selectedRow, 7));
        time5Field.setText("'" + memberCompetitionTable.getValueAt(selectedRow, 8));
        resetMemberIDComboBoxItems();
        if (!memberIDComboBoxContains(Integer.valueOf("'" + memberCompetitionTable.getValueAt(selectedRow, 1)))) {
            JOptionPane.showMessageDialog(memberCompetitionInputForm, "This member no longer exists", "Error",
                JOptionPane.ERROR_MESSAGE);
            editing = false;
            return;
        }
        memberIDComboBox.setSelectedItem(Integer.valueOf("'" + memberCompetitionTable.getValueAt(selectedRow, 1)));
        selectedMCIndex = selectedRow;
        memberCompetitionInputForm.setVisible(true);
    }
}

/**
 * Determines whether <b>memberIDComboBox</b> contains the specified item
 *
 * @param item
 *          the item to be found
 * @return <b>true</b> if the item is in memberIDComboBox; <br>
```

```
*      <b>false</b> otherwise
*/
private boolean memberIDComboBoxContains(Integer item) {
    int size = memberIDComboBox.getItemCount();

    for (int i = 0; i < size; ++i)
        if (memberIDComboBox.getItemAt(i).equals(item))
            return true;

    return false;
}

/**
 * Retrieves the data from the 'memberCompetition' table and adds it to the
 * table in the window
 */
public void populateCellsWithDatabaseData() {
    int rowCount = model.getRowCount();
    int rank = 1;
    String average;

    for (int i = 0; i < rowCount; ++i) { // Remove all rows from table
        model.removeRow(0);
    }

    MemberCompetition[] memberCompetitions = new MemberCompetition[0];

    try {
        memberCompetitions = MemberCompetitionDatabaseConnection.executeQuery("SELECT * "
            + "FROM memberCompetition;");
    } catch (SQLException e) {
        JOptionPane.showMessageDialog(memberCompetitionListPanel, "Could not load member-competitions", "Error",
            JOptionPane.ERROR_MESSAGE);
    } catch (ClassNotFoundException e) {
        JOptionPane.showMessageDialog(memberCompetitionListPanel, "Could not load member-competitions", "Error",
            JOptionPane.ERROR_MESSAGE);
    }

    if (memberCompetitions != null) {
        Sorter.sortByAverageThenTime(memberCompetitions, 0, memberCompetitions.length - 1);
        String[] times;
        Member[] currentMember;

        for (int i = 0; i < memberCompetitions.length; ++i) {
```

```
if (memberCompetitions[i].getCompetitionID() == currentCompetitionID) {
    try {
        currentMember = MemberDatabaseConnection.executeQuery("SELECT * FROM member WHERE memberID = "
            + memberCompetitions[i].getMemberID());

        if (currentMember.length == 0)
            throw new Exception();
    } catch (Exception e) {
        currentMember = new Member[1];
        currentMember[0] = new Member(0, "UNKNOWN", "", "", "", "", "", "");
    }

    times = memberCompetitions[i].getTimes();
    average = Solve.getSecondsToFormattedString(memberCompetitions[i].getAverage());

    model.addRow(new Object[] {
        "" + currentMember[0].getForenames() + " " + currentMember[0].getSurname(),
        "" + memberCompetitions[i].getMemberID(), "" + rank++,
        (average.equals("-1.00")) ? "DNF" : average + "", times[0], times[1], times[2], times[3],
        times[4] });
}
}

Main.resizeColumnWidths(memberCompetitionTable);
memberCompetitionInputForm.setVisible(false);
}

/**
 * Adds a row to the table and to the database
 *
 * @throws ClassNotFoundException
 *         if SQLite classes are missing
 * @throws SQLException
 *         if the table does not exist etc.
 */
private void addRow() throws ClassNotFoundException, SQLException {
    MemberCompetitionDatabaseConnection.executeUpdate(String.format(
        "INSERT INTO memberCompetition(competitionID, memberID, time1, time2, time3, time4, time5) "
        + "VALUES (%d, %d, \"\", \"\", \"\", \"\", %d, %d, %d)", currentCompetitionID,
        Integer.valueOf("" + memberIDComboBox.getSelectedItem())));
}

model.addRow(new Object[] { "", 0, "", "", "", "", 0, 0 });
```

```
    Main.resizeColumnWidths(memberCompetitionTable);
    memberCompetitionTable.setRowSelectionInterval(memberCompetitionTable.getRowCount() - 1,
        memberCompetitionTable.getRowCount() - 1);
}

/**
 * Deletes the selected rows from the table, and deletes the corresponding
 * records from the database
 *
 * @throws ClassNotFoundException
 *         if SQLite classes are missing
 * @throws SQLException
 *         if the table does not exist or the updates fail
 */
private void deleteRow() throws ClassNotFoundException, SQLException {
    int[] selectedIndices = memberCompetitionTable.getSelectedRows();
    // MemberCompetition[] membersCompetitions;

    if (selectedIndices.length == 0)
        return;

    for (int i = 0; i < selectedIndices.length; ++i) {
        MemberCompetitionDatabaseConnection.executeUpdate("DELETE FROM memberCompetition "
            + "WHERE competitionID = " + ("'" + currentCompetitionID) + " " + "AND memberID = "
            + ("'" + model.getValueAt(selectedIndices[i], 1)) + ";");
    }

    // for (int i = selectedIndices.length - 1; i >= 0; --i) {
    // model.removeRow(selectedIndices[i]);
    // }

    populateCellsWithDatabaseData();

    /*
     * int rowCount = memberCompetitionTable.getRowCount(); for (int i = 0;
     * i < rowCount; ++i) model.removeRow(0);
     *
     * membersCompetitions =
     * MemberCompetitionDatabaseConnection.executeQuery
     * ("SELECT * FROM memberCompetition");
     *
     * String[] times;
     *
     * for (int i = 0; i < membersCompetitions.length; ++i) { times =
     */
}
```

```
* membersCompetitions[i].getTimes(); .addRow(new Object[]{""
* membersCompetitions[i].getCompetitionID(),
* membersCompetitions[i].getMemberID(), times[0], times[1], times[2],
* times[3], times[4]}); }
*/
try {
    memberCompetitionTable.setRowSelectionInterval(selectedIndices[selectedIndices.length - 1],
        selectedIndices[selectedIndices.length - 1]);
} catch (Exception e) {
}
}

/**
 * Sets the competition ID to <b>competitionID</b> so that the correct
 * records are retrieved from the database
 *
 * @param competitionID
 *         the ID of the competition will be set to this value
 */
public void setCurrentCompetitionID(int competitionID) {
    this.currentCompetitionID = competitionID;
    competitionIndicatorLabel.setText("Competition ID = " + competitionID);
}

/**
 * Sets up the 'Member-Competition Form' window
 */
private void setUpMemberCompetitionInputForm() {
    memberCompetitionInputForm = new JFrame("Member-Competition Form");

    contentPane = new JPanel();
    contentPane.setLayout(null);

    memberCompetitionInputForm.setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
    memberCompetitionInputForm.setContentPane(contentPane);
    memberCompetitionInputForm.setSize(new Dimension(480, 397));
    memberCompetitionInputForm.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    memberCompetitionInputForm.setResizable(false);
    memberCompetitionInputForm.setLocation(600, 150);
    memberCompetitionInputForm.setVisible(false);

    memberCompetitionInputForm.addWindowListener(new WindowListener() {
        @Override
```

```
public void windowActivated(WindowEvent arg0) {  
}  
  
@Override  
public void windowClosed(WindowEvent arg0) {  
    time1Field.setText("");  
    time2Field.setText("");  
    time3Field.setText("");  
    time4Field.setText("");  
    time5Field.setText("");  
    errorMessageLabel.setVisible(false);  
}  
  
@Override  
public void windowClosing(WindowEvent arg0) {  
}  
  
@Override  
public void windowDeactivated(WindowEvent arg0) {  
}  
  
@Override  
public void windowDeiconified(WindowEvent arg0) {  
}  
  
@Override  
public void windowIconified(WindowEvent arg0) {  
}  
  
@Override  
public void windowOpened(WindowEvent arg0) {  
}  
  
});  
  
// *****Labels*****  
memberIDLabel = new JLabel("Member ID ");  
memberIDLabel.setSize(120, 40);  
memberIDLabel.setLocation(0 + pad, y + pad);  
  
y += fieldYSpacing;  
  
time1Label = new JLabel("Time 1");
```

```
time1Label.setSize(120, 40);
time1Label.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

time2Label = new JLabel("Time 2");
time2Label.setSize(120, 40);
time2Label.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

time3Label = new JLabel("Time 3");
time3Label.setSize(120, 40);
time3Label.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

time4Label = new JLabel("Time 4");
time4Label.setSize(120, 40);
time4Label.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

time5Label = new JLabel("Time 5");
time5Label.setSize(120, 40);
time5Label.setLocation(0 + pad, y + pad);

y += fieldYSpacing - 10;

errorMessageLabel = new JLabel("Error Message");
errorMessageLabel.setForeground(Color.RED);
errorMessageLabel.setSize(200, 40);
errorMessageLabel.setLocation(0 + pad, y + pad);
errorMessageLabel.setVisible(false);

// *****Text Fields*****

y = 0;

memberIDComboBox = new JComboBox<Integer>();
memberIDComboBox.setFont(fieldFont);
memberIDComboBox.setSize(350, 40);
memberIDComboBox.setLocation(100 + pad, y + pad);
memberIDComboBox.addKeyListener(this);
```

```
y += fieldYSpacing;

time1Field = new JTextField();
time1Field.setFont(fieldFont);
time1Field.setSize(350, 40);
time1Field.setLocation(100 + pad, y + pad);
time1Field.addKeyListener(this);

y += fieldYSpacing;

time2Field = new JTextField("");
time2Field.setFont(fieldFont);
time2Field.setSize(350, 40);
time2Field.setLocation(100 + pad, y + pad);
time2Field.addKeyListener(this);

y += fieldYSpacing;

time3Field = new JTextField("");
time3Field.setFont(fieldFont);
time3Field.setSize(350, 40);
time3Field.setLocation(100 + pad, y + pad);
time3Field.addKeyListener(this);

y += fieldYSpacing;

time4Field = new JTextField("");
time4Field.setFont(fieldFont);
time4Field.setSize(350, 40);
time4Field.setLocation(100 + pad, y + pad);
time4Field.addKeyListener(this);

y += fieldYSpacing;

time5Field = new JTextField("");
time5Field.setFont(fieldFont);
time5Field.setSize(350, 40);
time5Field.setLocation(100 + pad, y + pad);
time5Field.addKeyListener(this);

y += fieldYSpacing;

// *****Submission Button*****
```

```
y = 330;

submitButton = new JButton("Submit");
submitButton.setSize(480, 30);
submitButton.setLocation(0, y + pad);
submitButton.setFocusable(false);
submitButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        String[] times = { "" + time1Field.getText(), "" + time2Field.getText(), "" + time3Field.getText(),
                           "" + time4Field.getText(), "" + time5Field.getText() };

        for (int i = 0; i < 5; ++i) {
            if (!Solve.isValidTime(times[i])) {
                errorMessageLabel.setText("Some of your times are invalid");
                errorMessageLabel.setVisible(true);
                return;
            }
        }
        errorMessageLabel.setVisible(false);

        try {
            int row;
            if (!editing) {
                addRow();
                row = memberCompetitionTable.getRowCount() - 1;
            } else {
                // row = memberCompetitionTable.getSelectedRow();
                row = selectedMCIndex;
            }

            memberCompetitionTable.setValueAt("'" + memberIDComboBox.getSelectedItem(), row, 1);

            for (int i = 0; i < 5; ++i) {
                if (times[i].trim().equalsIgnoreCase("DNF"))
                    memberCompetitionTable.setValueAt("DNF", row, i + 4);
                else
                    memberCompetitionTable.setValueAt(Solve.getPaddedTime(Solve
                        .getSecondsToFormattedString(Solve.getFormattedString.ToDouble(times[i]))), row,
                        i + 4);
            }
        }
        editing = false;
    }
})
```

```
String update = String.format("UPDATE memberCompetition " + "SET time1 = \"%s\", "
    + "time2 = \"%s\", " + "time3 = \"%s\", " + "time4 = \"%s\", " + "time5 = \"%s\" "
    + "WHERE memberID = %d AND competitionID = %d", "" + model.getValueAt(row, 4),
    "" + model.getValueAt(row, 5), "" + model.getValueAt(row, 6),
    "" + model.getValueAt(row, 7), "" + model.getValueAt(row, 8),
    Integer.valueOf("") + model.getValueAt(row, 1)), currentCompetitionID);

MemberCompetitionDatabaseConnection.executeUpdate(update);
populateCellsWithDatabaseData();
Main.resizeColumnWidths(memberCompetitionTable);
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
}

errorMessageLabel.setVisible(false);
memberCompetitionInputForm.setVisible(false);
}
});

y += buttonYSpacing;

contentPane.add(memberIDLabel);
contentPane.add(time1Label);
contentPane.add(time2Label);
contentPane.add(time3Label);
contentPane.add(time4Label);
contentPane.add(time5Label);
contentPane.add(errorMessageLabel);

contentPane.add(time1Field);
contentPane.add(time2Field);
contentPane.add(memberIDComboBox);
contentPane.add(time3Field);
contentPane.add(time4Field);
contentPane.add(time5Field);

contentPane.add(submitButton);

}
/**
```

```
* This resets the items in the drop-down list in the 'Member-Competition
* Form' window so that it contains the correct items
*/
private void resetMemberIDComboBoxItems() {
    Integer[] items = new Integer[memberCompetitionTable.getRowCount()];
    int currentMemberID, selectedMemberID = -1, selectedRow;

    for (int i = 0; i < items.length; ++i) {
        items[i] = Integer.valueOf("'" + memberCompetitionTable.getValueAt(i, 1));
    }

    memberIDComboBox.removeAllItems();

    try {
        Member[] members = MemberDatabaseConnection.executeQuery("SELECT * FROM member ORDER BY memberID ASC");
        selectedRow = memberCompetitionTable.getSelectedRow();

        if (selectedRow > -1)
            selectedMemberID = Integer.valueOf("'" + memberCompetitionTable.getValueAt(selectedRow, 1));

        for (int i = 0; i < members.length; ++i) {
            currentMemberID = members[i].getMemberID();

            if (!LinearSearch.linearSearchContains(items, currentMemberID)
                || ((editing) && (currentMemberID == selectedMemberID)))
                memberIDComboBox.addItem(currentMemberID);
        }
    } catch (ClassNotFoundException | SQLException e1) {
        e1.printStackTrace();
    }
}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.KeyListener#keyPressed(java.awt.event.KeyEvent)
 */
@Override
public void keyPressed(KeyEvent arg0) {

}

/**
 * (non-Javadoc)
 *
```

```
* @see java.awt.event.KeyListener#keyReleased(java.awt.event.KeyEvent)
*/
@Override
public void keyReleased(KeyEvent arg0) {
}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.KeyListener#keyTyped(java.awt.event.KeyEvent)
 */
@Override
public void keyTyped(KeyEvent arg0) {
    if (arg0.getKeyChar() == KeyEvent.VK_ENTER) {
        submitButton.doClick();
    }
}
```

## Class MemberDatabaseConnection

```
package jCube;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * @author Kelsey McKenna
 */
public class MemberDatabaseConnection {

    /**
     * The try/catch block is invoked if the table does not exist or the query
     * is invalid
     *
     * @param query
     *         the SQLite query to be performed on 'member' table
     * @return an array of Members representing the result of the specified
     *         query
     * @throws SQLException
     *         if the query is invalid
     * @throws ClassNotFoundException
     *         if SQLite classes are missing
     */
    public static Member[] executeQuery(String query) throws SQLException, ClassNotFoundException {
        Member[] members = null;

        try {
            members = executeSafeQuery(query);
        } catch (SQLException e) {
            initTable();
            members = executeSafeQuery(query);
        }

        return members;
    }

    /**
     * This method will only be called once the table exists
     */
}
```

```
* @param query
*       the SQLite query to be performed on 'Member' table
* @return an array of Members representing the result of the specified
*       query
* @throws ClassNotFoundException
*       if SQLite classes are missing
* @throws SQLException
*       if the table does not exist or the query is invalid
*/
private static Member[] executeSafeQuery(String query) throws ClassNotFoundException, SQLException {
    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement;
    ResultSet rs;
    Member[] members;
    int numRecords = 0;

    connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
    statement = connection.createStatement();
    statement.setQueryTimeout(30); // set timeout to 30 sec.

    rs = statement.executeQuery(query);

    while (rs.next())
        ++numRecords;

    rs.close();
    rs = statement.executeQuery(query);

    members = new Member[numRecords];

    for (int i = 0; i < numRecords; ++i) {
        rs.next();
        members[i] = new Member(rs.getInt("memberID"), rs.getString("forenames"), rs.getString("surname"),
                               rs.getString("gender"), rs.getString("dateOfBirth"), rs.getString("email"),
                               rs.getString("formClass"));
    }

    try {
        if (connection != null)
            connection.close();
    } catch (SQLException e) {
    }
}
```

```
        return members;
    }

    /**
     * Executes the specified update on the 'member' table
     *
     * @param update
     *          the update to be performed on the table
     * @throws ClassNotFoundException
     *          if SQLite classes are missing
     * @throws SQLException
     *          if the query is invalid
     */
    public static void executeUpdate(String update) throws ClassNotFoundException, SQLException {
        Class.forName("org.sqlite.JDBC");
        Connection connection = null;
        Statement statement = null;

        connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
        statement = connection.createStatement();
        statement.setQueryTimeout(30); // set timeout to 30 sec.

        statement.executeUpdate(update);

        try {
            if (connection != null)
                connection.close();
        } catch (SQLException e2) {
        }
    }

    /**
     * Initialises the table in the database. This method will be called if the
     * executeQuery(...) method cannot find the table in the database.
     *
     * @throws ClassNotFoundException
     *          if SQLite classes are missing
     */
    private static void initTable() throws ClassNotFoundException {
        Class.forName("org.sqlite.JDBC");
        Connection connection = null;
        Statement statement = null;

        try {
```

```
connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
statement = connection.createStatement();
statement.setQueryTimeout(30); // set timeout to 30 sec.

statement.executeUpdate("DROP TABLE IF EXISTS member");
statement.executeUpdate("CREATE TABLE member(" + "memberID INTEGER PRIMARY KEY AUTOINCREMENT,"
+ "forenames TEXT," + "surname TEXT," + "gender TEXT," + "dateOfBirth TEXT," + "email TEXT, "
+ "formClass TEXT" + ");");

} catch (SQLException e) {
    try {
        if (connection != null)
            connection.close();
    } catch (SQLException e2) {
    }
}
}

/***
 * Resets the IDs in the 'member' table so that they are continuous, e.g. if
 * the IDs were 1, 2, 5, 6, 7, 12, 13 then they would be reset to 1, 2, 3,
 * 4, 5, 6, 7
 *
 * @throws ClassNotFoundException
 *         if SQLite classes are missing
 * @throws SQLException
 *         if the table does not exist etc.
 */
public static void resetIDs() throws ClassNotFoundException, SQLException {
    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement = null;
    ResultSet rs;

    try {
        connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
        statement = connection.createStatement();
        statement.setQueryTimeout(30); // set timeout to 30 sec.

        rs = statement.executeQuery("SELECT * FROM member;");

        statement = connection.createStatement();
        statement.executeUpdate("CREATE TABLE memberCopy(" + "memberID INTEGER PRIMARY KEY AUTOINCREMENT,"
+ "forenames TEXT," + "surname TEXT," + "gender TEXT," + "dateOfBirth TEXT," + "email TEXT, "
```

```
+ "formClass TEXT" + ");");  
  
while (rs.next()) {  
    statement.executeUpdate(String.format(  
        "INSERT INTO memberCopy(forenames, surname, gender, dateOfBirth, email, formClass) "  
        + "VALUES (\"%s\", \"%s\", \"%s\", \"%s\", \"%s\", \"%s\");",  
        rs.getString("forenames"), rs.getString("surname"), rs.getString("gender"),  
        rs.getString("dateOfBirth"), rs.getString("email"), rs.getString("formClass")));  
}  
  
statement.executeUpdate("DROP TABLE member");  
statement.executeUpdate("ALTER TABLE memberCopy RENAME TO member");  
  
/*  
 * Set up other tables etc.  
 */  
} catch (SQLException e) {  
    if (connection != null)  
        connection.close();  
  
    throw new SQLException();  
}  
}  
}
```

## Class MemberDatabasePopUp

```
package jCube;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.sql.SQLException;
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.DefaultTableModel;

/**
 * @author Kelsey McKenna
 */
public class MemberDatabasePopUp extends JFrame implements KeyListener {
    /**
     * Default serialVersionUID
     */
    private static final long serialVersionUID = 1L;
    /**
     * This stores the initial width of the window.
     */
    private static final int WIDTH = 800;
```

```
/**  
 * This stores the padding of the elements in the window. The greater the  
 * padding, the further towards the centre of the window the elements will  
 * be.  
 */  
private final int pad = 10;  
/**  
 * This indicates the vertical spacing between the text boxes etc. in the  
 * window.  
 */  
private final int fieldYSpacing = 50;  
/**  
 * This indicates the vertical spacing between the buttons etc. in the  
 * window.  
 */  
private final int buttonYSpacing = 40;  
  
/**  
 * This variable keeps track of the current y position of the last element  
 * placed in the window.  
 */  
private int y = 0;  
/**  
 * Stores the font to be used in the text fields  
 */  
private Font fieldFont = new Font("Arial", 0, 25);  
/**  
 * If this variable is true, it indicates that a record is being edited,  
 * otherwise a record is being added.  
 */  
private boolean editing = false;  
  
/**  
 * After choosing to edit a record in the table, the index of this row in  
 * the table is stored in this variable.  
 */  
private int selectedRowIndex;  
  
/**  
 * This panel is used to store the table.  
 */  
private JPanel memberListPanel;  
/**  
 * memberTable is placed 'inside' this variable so that when the size of the
```

```
* table exceeds the size of the window, the user can scroll in order to
* view the rest of the table.
*/
private JScrollPane tableContainer;
/**
 * This is the table that is displayed in the window; it stores the contents
 * of the table and the rendering features required to display the data.
*/
private final JTable memberTable;
/**
 * This variable can be customised so that certain cells of the table are
 * uneditable and the columns of the table can be given text. This variable
 * is then set as the model of memberTable.
*/
private final DefaultTableModel model;
/**
 * The buttons in the window are placed in this panel.
*/
private JPanel buttonPanel;

/**
 * Clicking this button opens the 'Member Form' window.
*/
private JButton addMemberButton;
/**
 * Clicking this button opens the 'Member Form' window if a row has been
 * selected.
*/
private JButton editMemberButton;
/**
 * Clicking this button deletes the selected rows from the table.
*/
private JButton deleteMemberButton;

/**
 * This label is shown in the window with the text 'Forenames'.
*/
private JLabel forenamesLabel;
/**
 * This label is shown in the window with the text 'Surname'.
*/
private JLabel surnameLabel;
/**
 * This label is shown in the window with the text 'Gender'.
*/
```

```
/*
private JLabel genderLabel;
/**
 * This label is shown in the window with the text 'Date of Birth'.
 */
private JLabel dateOfBirthLabel;
/**
 * This label is shown in the window with the text 'Email'.
 */
private JLabel emailLabel;
/**
 * This label is shown in the window with the text 'Form Class'.
 */
private JLabel formClassLabel;

/**
 * This field is shown in the window and the user can enter the forenames
 * into this field.
 */
private JTextField forenamesField;
/**
 * This field is shown in the window and the user can enter the surname into
 * this field.
 */
private JTextField surnameField;
/**
 * This drop-down list stores the items 'Male' and 'Female' * window.
 */
private JComboBox<String> genderField;
/**
 * This field is shown in the window and the user can enter the date of
 * birth into this field.
 */
private JTextField dateOfBirthField;
/**
 * This field is shown in the window and the user can enter the email into
 * this field.
 */
private JTextField emailField;
/**
 * This field is shown in the window and the user can enter the form class
 * into this field.
 */
private JTextField formClassField;
```

```
/**  
 * This represents the 'Member Form' window.  
 */  
private JFrame memberInputForm;  
  
/**  
 * Clicking this button submits the data in the 'Member Form' window for  
 * validation.  
 */  
private JButton submitButton;  
  
/**  
 * This panel stores the elements of the 'Member Form' window.  
 */  
private JPanel contentPane;  
  
/**  
 * Constructor - sets up the 'Member Table' window  
 */  
public MemberDatabasePopUp() {  
    super("Member Table");  
  
    setIconImage(Main.createImage("res/images/RubikCubeBig.png"));  
    setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);  
    setLayout(new BorderLayout());  
    setPreferredSize(new Dimension(WIDTH, 400));  
    setVisible(false);  
    setUpMemberInputForm();  
  
    memberListPanel = new JPanel();  
    memberListPanel.setOpaque(true);  
  
    final String[] columnNames = { "ID", "Forenames", "Surname", "Gender", "Date of Birth", "Email", "Form Class" };  
  
    model = new DefaultTableModel() {  
        private static final long serialVersionUID = 1L;  
  
        public int getColumnCount() {  
            return columnNames.length;  
        }  
  
        public String getColumnName(int col) {  
            return columnNames[col];  
        }  
    };  
}
```

```
}

public boolean isCellEditable(int row, int col) {
    return false;
}
};

model.addTableModelListener(new TableModelListener() {
    @Override
    public void tableChanged(TableModelEvent e) {
    }
});

memberTable = new JTable();
memberTable.setModel(model);
memberTable.setColumnSelectionAllowed(false);
memberTable.setPreferredScrollableViewportSize(new Dimension(WIDTH, 70));
memberTable.setFillsViewportHeight(true);
memberTable.setAutoScrolls(true);
memberTable.getTableHeader().setReorderingAllowed(false);
memberTable.setFont(new Font("Arial", 0, 15));
memberTable.setRowHeight(20);
memberTable.addMouseListener(new MouseListener() {

    @Override
    public void mouseClicked(MouseEvent arg0) {
        if (arg0.getClickCount() == 2) {
            editMemberButtonFunction();
        }
    }

    @Override
    public void mouseEntered(MouseEvent arg0) {
    }

    @Override
    public void mouseExited(MouseEvent arg0) {
    }

    @Override
    public void mousePressed(MouseEvent arg0) {
    }

    @Override
    public void mouseReleased(MouseEvent arg0) {
```

```
        }

    });

buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(1, 3));
buttonPanel.setPreferredSize(new Dimension(WIDTH, 40));
buttonPanel.setSize(WIDTH, 50);

addMemberButton = new JButton("Add Member");
addMemberButton.setFocusable(false);
addMemberButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        memberTable.clearSelection();
        editing = false;
        forenamesField.setText("");
        surnameField.setText("");
        dateOfBirthField.setText("");
        emailField.setText("");
        formClassField.setText("");
        memberInputForm.setVisible(true);
    }
});
editMemberButton = new JButton("Edit Member");
editMemberButton.setFocusable(false);
editMemberButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        editMemberButtonFunction();
    }
});
deleteMemberButton = new JButton("Delete Member");
deleteMemberButton.setFocusable(false);
deleteMemberButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        try {
            Object[] options = { "Yes", "No" };
            int choice = -1;

            int selectedIndex = memberTable.getSelectedRow();

            if (selectedIndex == -1)
                return;
        }
    }
});
```

```
choice = JOptionPane.showOptionDialog(null, "Are you sure you want to delete?", "Warning",
JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[1]);

if (choice == 0)
    deleteRow();
} catch (ClassNotFoundException | SQLException e) {
    JOptionPane.showMessageDialog(memberListPanel, "Unable to delete record from database", "Error",
    JOptionPane.ERROR_MESSAGE);
}
};

buttonPanel.add(addMemberButton);
buttonPanel.add(editMemberButton);
buttonPanel.add(deleteMemberButton);

tableContainer = new JScrollPane(memberTable);
tableContainer.setViewportView(memberTable, null);

Main.resizeColumnWidths(memberTable);
populateCellsWithDatabaseData();

add(tableContainer, null);
add(buttonPanel, BorderLayout.SOUTH);
pack();
}

/**
 * Performs the operations required to set up the 'Member Form' window with
 * the contents of the selected row in the table and indicates that the row
 * is being edited (and not added)
 */
private void editMemberButtonFunction() {
    int selectedRow = memberTable.getSelectedRow();

    if (selectedRow != -1) {
        forenamesField.setText("") + memberTable.getValueAt(selectedRow, 1));
        surnameField.setText("") + memberTable.getValueAt(selectedRow, 2));
        genderField.setSelectedItem("") + memberTable.getValueAt(selectedRow, 3));
        dateOfBirthField.setText("") + memberTable.getValueAt(selectedRow, 4));
        emailField.setText("") + memberTable.getValueAt(selectedRow, 5));
        formClassField.setText("") + memberTable.getValueAt(selectedRow, 6));
        memberInputForm.setVisible(true);
    }
}
```

```
        editing = true;
        selectedRowIndex = selectedRow;
    }
}

/**
 * Retrieves the data from the 'member' table and adds it to the table in
 * the window
 */
private void populateCellsWithDatabaseData() {
    int rowCount = model.getRowCount();

    for (int i = 0; i < rowCount; ++i) { // Remove all rows from table
        model.removeRow(0);
    }

    Member[] members = new Member[0];

    try {
        members = MemberDatabaseConnection.executeQuery("SELECT * " + "FROM member;");
    } catch (SQLException e) {
        JOptionPane
            .showMessageDialog(memberListPanel, "Could not load members", "Error", JOptionPane.ERROR_MESSAGE);
    } catch (ClassNotFoundException e) {
        JOptionPane
            .showMessageDialog(memberListPanel, "Could not load members", "Error", JOptionPane.ERROR_MESSAGE);
    }

    if (members != null) {
        for (int i = 0; i < members.length; ++i) {
            model.addRow(new Object[] { "" + members[i].getMemberID(), members[i].getForenames(),
                members[i].getSurname(), members[i].getGender(), members[i].getDateOfBirth(),
                members[i].getEmail(), members[i].getFormClass() });
        }
    }
}

Main.resizeColumnWidths(memberTable);
}

/**
 * Adds a row to the table and to the database
 *
 * @throws ClassNotFoundException
 *         if SQLite classes are missing

```

```
* @throws SQLException
*           if the table does not exist etc.
*/
private void addRow() throws ClassNotFoundException, SQLException {
    MemberDatabaseConnection
        .executeUpdate("INSERT INTO member(forenames, surname, gender, dateOfBirth, email, formClass) VALUES
(\"\", \"\", \"\", \"\", \"\", \"\")");
    Member[] member = MemberDatabaseConnection.executeQuery("SELECT * FROM member ORDER BY memberID DESC LIMIT 1");
    model.addRow(new Object[] { "" + (member[0].getMemberID()), "", "", "", "", "" });

    Main.resizeColumnWidths(memberTable);
    memberTable.setRowSelectionInterval(memberTable.getRowCount() - 1, memberTable.getRowCount() - 1);
}

/**
 * Deletes the selected rows from the table, and deletes the corresponding
 * records from the database
 *
 * @throws ClassNotFoundException
 *           if SQLite classes are missing
 * @throws SQLException
 *           if the table does not exist or the updates fail
 */
private void deleteRow() throws ClassNotFoundException, SQLException {
    int[] selectedIndices = memberTable.getSelectedRows();
    Member[] members;

    if (selectedIndices.length == 0)
        return;

    for (int i = 0; i < selectedIndices.length; ++i) {
        MemberDatabaseConnection.executeUpdate("DELETE FROM member WHERE memberID = "
            + ("\" + model.getValueAt(selectedIndices[i], 0)) + ";");
    }

    int rowCount = memberTable.getRowCount();
    for (int i = 0; i < rowCount; ++i)
        model.removeRow(0);

    members = MemberDatabaseConnection.executeQuery("SELECT * FROM member");

    for (int i = 0; i < members.length; ++i)
        model.addRow(new Object[] { members[i].getMemberID(), members[i].getForenames(), members[i].getSurname(),
            members[i].getGender(), members[i].getDateOfBirth(), members[i].getEmail(),
```

```
        members[i].getFormClass() });

    try {
        memberTable.setRowSelectionInterval(selectedIndices[selectedIndices.length - 1],
            selectedIndices[selectedIndices.length - 1]);
    } catch (Exception e) {
    }
}

/**
 * Returns a value indicating whether the argument is a valid date for a
 * date of birth
 *
 * @param dateString
 *         the date to be analysed
 * @return true if the argument is valid and is in the format dd/MM/yyyy;
 *         false otherwise
 */
private static boolean isValidDate(String dateString) {
    try {
        Date dateEntered = new SimpleDateFormat("dd/MM/yyyy").parse(dateString);

        if (!dateString.matches("((\\d*){2,2}\\d{4,4})"))
            return false;

        Date currentDate = new Date();
        int day, month, year;

        day = Integer.valueOf(dateString.substring(0, dateString.indexOf("/")));
        dateString = dateString.substring(dateString.indexOf("/") + 1);

        month = Integer.valueOf(dateString.substring(0, dateString.indexOf("/")));
        year = Integer.valueOf(dateString.substring(dateString.indexOf("/") + 1));

        if ((month > 12) || (getNumDaysInMonth(month, year) < day) || (dateEntered.after(currentDate)))
            return false;
        else
            return true;
    } catch (Exception e) {
        return false;
    }
}

/**
```

```
* @param month
*       the month of the date in question
* @param year
*       the year of the date in question
* @return the number of days in the month
*/
private static int getNumDaysInMonth(int month, int year) {
    switch (month) {
        case 2:
            if ((year % 4 == 0) && ((year % 100 == 0) ? (year % 400 == 0) : true)) {
                return 29;
            } else {
                return 28;
            }
        case 4:
        case 6:
        case 7:
        case 11:
            return 30;
        default:
            return 31;
    }
}

/**
 * Sets up the 'Member Form' window
 */
private void setUpMemberInputForm() {
    memberInputForm = new JFrame("Member Form");

    contentPane = new JPanel();
    contentPane.setLayout(null);

    memberInputForm.setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
    memberInputForm.setContentPane(contentPane);
    memberInputForm.setPreferredSize(new Dimension(480, 440));
    memberInputForm.setSize(new Dimension(480, 380));
    memberInputForm.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    memberInputForm.setResizable(false);
    memberInputForm.setLocation(600, 150);
    memberInputForm.setVisible(false);

    // *****Labels*****
    forenamesLabel = new JLabel("Forenames ");
}
```

```
forenamesLabel.setSize(120, 40);
forenamesLabel.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

surnameLabel = new JLabel("Surname ");
surnameLabel.setSize(120, 40);
surnameLabel.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

genderLabel = new JLabel("Gender ");
genderLabel.setSize(120, 40);
genderLabel.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

dateOfBirthLabel = new JLabel("Date of Birth ");
dateOfBirthLabel.setSize(120, 40);
dateOfBirthLabel.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

emailLabel = new JLabel("Email ");
emailLabel.setSize(120, 40);
emailLabel.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

formClassLabel = new JLabel("Form Class ");
formClassLabel.setSize(120, 40);
formClassLabel.setLocation(0 + pad, y + pad);

// *****Text Fields*****
y = 0;

forenamesField = new JTextField();
forenamesField.setFont(fieldFont);
forenamesField.setSize(350, 40);
forenamesField.setLocation(100 + pad, y + pad);
forenamesField.addKeyListener(this);

y += fieldYSpacing;
```

```
surnameField = new JTextField("");
surnameField.setFont(fieldFont);
surnameField.setSize(350, 40);
surnameField.setLocation(100 + pad, y + pad);
surnameField.addKeyListener(this);

y += fieldYSpacing;

genderField = new JComboBox<String>();
genderField.addItem("Male");
genderField.addItem("Female");
genderField.setFont(fieldFont);
genderField.setSize(350, 40);
genderField.setLocation(100 + pad, y + pad);
genderField.addKeyListener(this);

y += fieldYSpacing;

dateOfBirthField = new JTextField("");
dateOfBirthField.setFont(fieldFont);
dateOfBirthField.setSize(350, 40);
dateOfBirthField.setLocation(100 + pad, y + pad);
dateOfBirthField.addKeyListener(this);

y += fieldYSpacing;

emailField = new JTextField("");
emailField.setFont(fieldFont);
emailField.setSize(350, 40);
emailField.setLocation(100 + pad, y + pad);
emailField.addKeyListener(this);

y += fieldYSpacing;

formClassField = new JTextField();
formClassField.setFont(fieldFont);
formClassField.setSize(350, 40);
formClassField.setLocation(100 + pad, y + pad);
formClassField.addKeyListener(this);

y += fieldYSpacing;

// *****Submission Button*****
```

```
y = 310;

submitButton = new JButton("Submit");
submitButton.setSize(480, 30);
submitButton.setLocation(0, y + pad);
submitButton.setFocusable(false);
submitButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        int row, memberID;
        String dateOfBirth = dateOfBirthField.getText().trim();
        String email = emailField.getText().trim();
        String formClass = formClassField.getText().trim();
        String forenames = forenamesField.getText().trim();
        String surname = surnameField.getText().trim();
        String gender = (genderField.getSelectedItem() + "").trim();

        if ((dateOfBirth.equals("")) || (!isValidDate(dateOfBirth))) {
            JOptionPane.showMessageDialog(memberInputForm, "Invalid date of birth", "Error",
                JOptionPane.ERROR_MESSAGE);
            return;
        } else if ((email.equals("")) || (!Member.isValidEmail(email))) {
            JOptionPane.showMessageDialog(memberInputForm, "Invalid email", "Error", JOptionPane.ERROR_MESSAGE);
            return;
        } else if ((formClass.equals("")) || (!Member.isValidFormClass(formClass))) {
            JOptionPane.showMessageDialog(memberInputForm, "Invalid form class", "Error",
                JOptionPane.ERROR_MESSAGE);
            return;
        } else if ((forenames.equals(""))){
            JOptionPane.showMessageDialog(memberInputForm, "Empty forenames", "Error",
                JOptionPane.ERROR_MESSAGE);
            return;
        } else if ((surname.equals(""))){
            JOptionPane.showMessageDialog(memberInputForm, "Empty surname", "Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        try {
            if (!editing) {
                addRow();
                row = memberTable.getRowCount() - 1;
            } else {
                // row = memberTable.getSelectedRow();
                row = selectedRowIndex;
            }
        }
    }
})
```

```
        }

        memberID = Integer.valueOf("") + memberTable.getValueAt(row, 0));

        model.setValueAt(forenames + "", row, 1);
        model.setValueAt(surname + "", row, 2);
        model.setValueAt(gender + "", row, 3);
        model.setValueAt(dateOfBirth + "", row, 4);
        model.setValueAt(email + "", row, 5);
        model.setValueAt(formClass + "", row, 6);

        memberInputForm.setVisible(false);
        forenamesField.requestFocus();
        editing = false;

        MemberDatabaseConnection.executeUpdate(String.format("UPDATE member " + "SET forenames = \"%s\", "
            + "surname = \"%s\", " + "gender = \"%s\", " + "dateOfBirth = \"%s\", "
            + "email = \"%s\", " + "formClass = \"%s\" " + "WHERE memberID = %d", "" + forenames, ""
            + surname, "" + gender, "" + dateOfBirth, "" + email, "" + formClass, memberID));
        Main.resizeColumnWidths(memberTable);
    } catch (ClassNotFoundException | SQLException e) {
        e.printStackTrace();
    }
}

y += buttonYSpacing;

contentPane.add(forenamesLabel);
contentPane.add(surnameLabel);
contentPane.add(genderLabel);
contentPane.add(dateOfBirthLabel);
contentPane.add(emailLabel);
contentPane.add(formClassLabel);

contentPane.add(forenamesField);
contentPane.add(surnameField);
contentPane.add(genderField);
contentPane.add(dateOfBirthField);
contentPane.add(emailField);
contentPane.add(formClassField);

contentPane.add(submitButton);
```

```
}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.KeyListener#keyPressed(java.awt.event.KeyEvent)
 */
@Override
public void keyPressed(KeyEvent arg0) {

}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.KeyListener#keyReleased(java.awt.event.KeyEvent)
 */
@Override
public void keyReleased(KeyEvent arg0) {

}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.KeyListener#keyTyped(java.awt.event.KeyEvent)
 */
@Override
public void keyTyped(KeyEvent arg0) {
    if (arg0.getKeyChar() == KeyEvent.VK_ENTER) {
        submitButton.doClick();
    }
}

}
```

## Class MouseSelectionSolver

```
package jCube;

import static java.lang.Math.tan;
import static java.lang.Math.toRadians;

import java.awt.Color;
import java.util.Arrays;

import javax.swing.JOptionPane;

/**
 * @author Kelsey McKenna
 */
public class MouseSelectionSolver extends SolveMaster {

    /**
     * This is so that an identifier in the code instead of the String "BLANK"
     */
    public static final String BLANK = "BLANK";

    /**
     * This variable allows certain methods to be accessed to perform general
     * operations on the cube.
     */
    private SolveMaster solveMaster;
    /**
     * This variable allows a solution for the cross to be generated.
     */
    private CrossSolver crossSolver;
    /**
     * This variable allows a solution for the first-layer corners to be
     * generated.
     */
    private CornerSolver cornerSolver;
    /**
     * This variable allows a solution for the middle-layer edges to be
     * generated.
     */
    private EdgeSolver edgeSolver;
    /**
     * This variable allows a solution for the orientation of the last layer to
     * be generated.
     */
}
```

```
/*
private OrientationSolver orientationSolver;
/**
 * This variable allows a solution for the permutation of the last layer to
 * be generated.
 */
private PermutationSolver permutationSolver;

/**
 * This variable is used in calculations for determining the indices of
 * pieces on the screen.
 */
private static double xFactor = 190. / 156;
/**
 * This variable is used in calculations for determining the indices of
 * pieces on the screen.
 */
private static double uAngle = tan(toRadians(33));
/**
 * This variable is used in calculations for determining the indices of
 * pieces on the screen.
 */
private static double rAngle = tan(toRadians(55));

/**
 * This accumulates the solutions generated by crossSolver, edgeSolver etc.
 */
private String solution = BLANK;

/**
 * Constructor - assigns values to fields
 *
 * @param cube
 *          the Cube to be used
 * @param a
 *          the CrossSolver to be used
 * @param b
 *          the CornerSolver to be used
 * @param c
 *          the EdgeSolver to be used
 * @param d
 *          the OrientationSolver to be used
 * @param e
 *          the PermutationSolver to be used
```

```
/*
public MouseSelectionSolver(Cube cube, CrossSolver a, CornerSolver b, EdgeSolver c, OrientationSolver d,
    PermutationSolver e) {
    super(cube);
    solveMaster = new SolveMaster(cube);
    this.crossSolver = a;
    this.cornerSolver = b;
    this.edgeSolver = c;
    this.orientationSolver = d;
    this.permutationSolver = e;
}

/**
 * @return the generated solution
 */
public String getSolution() {
    return this.solution;
}

/**
 * Solves the piece at the specified index, and records the solution and
 * explanation simultaneously.
 *
 * @param index
 *          the index of the piece to be solved
 */
public void solvePiece(int index) {
    // int choice = -1;
    Edge edge;
    Corner corner;

    Main.clearAllSolverMoves();

    if (index < 0)
        return;
    else if (index >= 8) { // an Edge
        index = index - 8;
        edge = new Edge(NSArray.copyOf(cube.getEdge(index).getStickers(), 2));
        solveMaster.rotateToTopFront(Color.green, Color.white);
        // edge.setOrientation(cube.getEdge(solveMaster.getIndexOf(edge)).getOrientation());
        edge.setOrientation(LinearSearch.linearSearch(edge.getStickers(), Color.white));
        if (edge.getOrientation() == -1)
            edge.setOrientation(LinearSearch.linearSearch(edge.getStickers(), Color.yellow));
    }
}
```

```
if (CrossSolver.isCrossEdge(edge)) {
    crossSolver.solveCross();
} else if (EdgeSolver.isMLEdge(edge)) {
    crossSolver.solveCross();
    cornerSolver.solveFirstLayerCorners();
    edgeSolver.solveEdge(getIndexOf(edge));
} else {
    crossSolver.solveCross();
    cornerSolver.solveFirstLayerCorners();
    edgeSolver.solveMiddleLayerEdges();
    orientationSolver.solveOrientation();
    permutationSolver.solvePermutation();
}
} else { // Corner
    corner = new Corner(Arrays.copyOf(cube.getCorner(index).getStickers(), 3));
    solveMaster.rotateToTopFront(Color.white, Color.green);
    corner.setOrientation(LinearSearch.linearSearchCornerOrientation(corner.getStickers(), Color.white));
    if (corner.getOrientation() == -2)
        corner.setOrientation(LinearSearch.linearSearchCornerOrientation(corner.getStickers(), Color.yellow));

    if (CornerSolver.isFLCorner(corner)) {
        crossSolver.solveCross();
        cornerSolver.solveCorner(getIndexOf(corner));
    } else {
        crossSolver.solveCross();
        cornerSolver.solveFirstLayerCorners();
        edgeSolver.solveMiddleLayerEdges();
        orientationSolver.solveOrientation();
        permutationSolver.solvePermutation();
    }
}

simplifyMoves(crossSolver.getCatalogMoves(), CROSS);
simplifyMoves(cornerSolver.getCatalogMoves(), CORNER_EDGE);
simplifyMoves(edgeSolver.getCatalogMoves(), CORNER_EDGE);
simplifyMoves(orientationSolver.getCatalogMoves(), CANCELLATIONS);
simplifyMoves(permutationSolver.getCatalogMoves(), CANCELLATIONS);

/*
 * this.solution = ("-----Solution-----\n" +
 * ((crossSolver.getStringMoves().trim().equals("")) ? "" : "Cross: \t"
 * + crossSolver.getStringMoves() +
 * ((cornerSolver.getStringMoves().trim().equals("")) ? "" :
 * "\nCorners: \t" + cornerSolver.getStringMoves()) +
 * ((edgeSolver.getStringMoves().trim().equals("")) ? "" :
 * "\nEdges: \t" + edgeSolver.getStringMoves()) +
 * ((orientationSolver.getStringMoves().trim().equals("")) ? "" :
 * "\nOrientation: \t" + orientationSolver.getStringMoves()) +
 * ((permutationSolver.getStringMoves().trim().equals("")) ? "" :
 * "\nPermutation: \t" + permutationSolver.getStringMoves()));
 */
```

```
* ((edgeSolver.getStringMoves().trim().equals("")) ? "" : "\nEdges: \t"
* + edgeSolver.getStringMoves() +
* ((orientationSolver.getStringMoves().trim().equals("")) ? "" :
* "\nOrientation: \t" + orientationSolver.getStringMoves() +
* ((permutationSolver.getStringMoves().trim().equals("")) ? "" :
* "\nPermutation: \t" + permutationSolver.getStringMoves() +
* "\n\n-----Explanation-----\n" +
* ((cornerSolver.getSolutionExplanation().trim().equals("")) ? "" :
* "Corners:\n" + cornerSolver.getSolutionExplanation() + "\n" +
* ((edgeSolver.getSolutionExplanation().trim().equals("")) ? "" :
* "Edges:\n" + edgeSolver.getSolutionExplanation() + "\n" +
* ((orientationSolver.getSolutionExplanation().trim().equals("")) ? "" :
* : "Orientation:\n" + orientationSolver.getSolutionExplanation() +
* "\n" +
* ((permutationSolver.getSolutionExplanation().trim().equals("")) ? ""
* : "Permutation:\n" + permutationSolver.getSolutionExplanation()));
*/
solution = Main.getFormattedCubeSolution();

if (!solution.contains("U") && !solution.contains("R") && !solution.contains("F") && !solution.contains("D")
    && !solution.contains("L") && !solution.contains("B") && !solution.contains("M"))
    solution = BLANK;

cube.performAbsoluteMoves(SolveMaster.getReverseStringMoves(permutationSolver.getCatalogMoves()));
cube.performAbsoluteMoves(SolveMaster.getReverseStringMoves(orientationSolver.getCatalogMoves()));
cube.performAbsoluteMoves(SolveMaster.getReverseStringMoves(edgeSolver.getCatalogMoves()));
cube.performAbsoluteMoves(SolveMaster.getReverseStringMoves(cornerSolver.getCatalogMoves()));
cube.performAbsoluteMoves(SolveMaster.getReverseStringMoves(crossSolver.getCatalogMoves()));

/*
 * crossSolver.clearMoves(); cornerSolver.clearMoves();
 * edgeSolver.clearMoves(); orientationSolver.clearMoves();
 * permutationSolver.clearMoves();
 */
}

/**
 * Returns the index of the piece on screen with the coordinates (x, y).
 *
 * @param x
 *      the x coordinate of the piece on screen
 * @param y
 *      the y coordinate of the piece on screen
```

```
* @return Corners: 0, 1, 2, ..., 7 <br>
*         Edges: 8, 9, 10, ..., 19 <br>
*         <b>-2</b> if the selection is invalid
*/
public static int getIndexOfPieceOnScreen(int x, int y) {
    double cartY = 166 - y;
    double cartX = x - 266;
    double uExtension = cartY * uAngle * xFactor;
    double rExtension = cartX * rAngle / xFactor;
    int row, col;

    // U-Slice
    if ((cartY >= 0) && (cartY <= 156) && (x >= 70 + uExtension) && (x <= uExtension + 260)) {
        if (cartY >= 106)
            row = 0;
        else if (cartY >= 56) {
            row = 1;
        } else
            row = 2;

        if (x >= 203 + uExtension)
            col = 2;
        else if (x >= 137 + uExtension)
            col = 1;
        else
            col = 0;

        switch (row) {
        case 0:
            switch (col) {
            case 0:
                return 0;
            case 1:
                return 8;
            case 2:
                return 1;
            }
        case 1:
            switch (col) {
            case 0:
                return 11;
            case 1:
                return -1; // Centre
            case 2:
```

```
        return 9;
    }
    case 2:
        switch (col) {
            case 0:
                return 3;
            case 1:
                return 10;
            case 2:
                return 2;
        }
    }
// F-slice
else if ((y >= 169) && (y <= 363) && (x >= 67) && (x <= 260)) {
    if (y <= 231)
        row = 0;
    else if (y <= 298)
        row = 1;
    else
        row = 2;

    if (x <= 128)
        col = 0;
    else if (x <= 194)
        col = 1;
    else
        col = 2;

    switch (row) {
        case 0:
            switch (col) {
                case 0:
                    return 3;
                case 1:
                    return 10;
                case 2:
                    return 2;
            }
        case 1:
            switch (col) {
                case 0:
                    return 15;
                case 1:
```

```
        return -1; // Centre
    case 2:
        return 14;
    }
case 2:
    switch (col) {
    case 0:
        return 6;
    case 1:
        return 18;
    case 2:
        return 7;
    }
}
// R-slice
else if ((x >= 266) && (x <= 387) && (y <= 359 - rExtension) && (y >= 169 - rExtension)) {
    if (x >= 350)
        col = 2;
    else if (x >= 308)
        col = 1;
    else
        col = 0;

    if (y >= 300 - rExtension)
        row = 2;
    else if (y >= 234 - rExtension)
        row = 1;
    else
        row = 0;

    switch (row) {
    case 0:
        switch (col) {
        case 0:
            return 2;
        case 1:
            return 9;
        case 2:
            return 1;
        }
    case 1:
        switch (col) {
        case 0:
```

```
        return 14;
    case 1:
        return -1; // Centre
    case 2:
        return 13;
    }
    case 2:
        switch (col) {
    case 0:
        return 7;
    case 1:
        return 19;
    case 2:
        return 4;
    }
}
}

return -2; // Not a valid selection
}

/**
 * Returns the index of the facelet/sticker on screen with the coordinates
 * (x, y).
 *
 * @param x
 *      the x coordinate of the facelet/sticker on screen
 * @param y
 *      the y coordinate of the facelet/sticker on screen
 * @return index of facelet/sticker on screen <br>
 *         <b>-2</b> if the selection is invalid
 */
public static int getIndexOfFaceletOnScreen(int x, int y) {
    double cartY = 166 - y;
    double cartX = x - 266;
    double uExtension = cartY * uAngle * xFactor;
    double rExtension = cartX * rAngle / xFactor;
    int row, col;

    // U-slice
    if ((cartY >= 0) && (cartY <= 156) && (x >= 70 + uExtension) && (x <= uExtension + 260)) {
        if (cartY >= 106)
            row = 0;
        else if (cartY >= 56) {
```

```
    row = 1;
} else
    row = 2;

if (x >= 203 + uExtension)
    col = 2;
else if (x >= 137 + uExtension)
    col = 1;
else
    col = 0;

switch (row) {
case 0:
    switch (col) {
    case 0:
        return 0;
    case 1:
        return 1;
    case 2:
        return 2;
    }
case 1:
    switch (col) {
    case 0:
        return 3;
    case 1:
        return -1; // Centre
    case 2:
        return 5;
    }
case 2:
    switch (col) {
    case 0:
        return 6;
    case 1:
        return 7;
    case 2:
        return 8;
    }
}
// F-slice
else if ((y >= 169) && (y <= 363) && (x >= 67) && (x <= 260)) {
    if (y <= 231)
```

```
    row = 0;
else if (y <= 298)
    row = 1;
else
    row = 2;

if (x <= 128)
    col = 0;
else if (x <= 194)
    col = 1;
else
    col = 2;

switch (row) {
case 0:
    switch (col) {
    case 0:
        return 9;
    case 1:
        return 10;
    case 2:
        return 11;
    }
case 1:
    switch (col) {
    case 0:
        return 12;
    case 1:
        return -1; // Centre
    case 2:
        return 14;
    }
case 2:
    switch (col) {
    case 0:
        return 15;
    case 1:
        return 16;
    case 2:
        return 17;
    }
}
// R-slice
```

```
else if ((x >= 266) && (x <= 387) && (y <= 359 - rExtension) && (y >= 169 - rExtension)) {
    if (x >= 350)
        col = 2;
    else if (x >= 308)
        col = 1;
    else
        col = 0;

    if (y >= 300 - rExtension)
        row = 2;
    else if (y >= 234 - rExtension)
        row = 1;
    else
        row = 0;

    switch (row) {
        case 0:
            switch (col) {
                case 0:
                    return 18;
                case 1:
                    return 19;
                case 2:
                    return 20;
            }
        case 1:
            switch (col) {
                case 0:
                    return 21;
                case 1:
                    return -1; // Centre
                case 2:
                    return 23;
            }
        case 2:
            switch (col) {
                case 0:
                    return 24;
                case 1:
                    return 25;
                case 2:
                    return 26;
            }
    }
}
```

```
}

    return -2; // Not a valid selection
}

/**
 * Shows a question dialog and gets a decision from a user
 *
 * @param message
 *         the text to be shown in the dialog
 * @return <b>0</b> if 'Yes' is selected; <br>
 *         <b>1</b> if 'No' is selected; <br>
 *         <b>-1</b> if nothing is selected
 */
public static int getQuestionDialogResponse(String message) {
    Object[] options = { "Yes", "No" };
    int choice = -1;

    choice = JOptionPane.showOptionDialog(null, message, "Warning", JOptionPane.YES_NO_OPTION,
        JOptionPane.WARNING_MESSAGE, null, options, options[0]);

    return choice;
}
```

## Class OrientationSolver

```
package jCube;

import java.awt.Color;

/**
 * @author Kelsey McKenna
 */
public class OrientationSolver extends SolveMaster {

    /**
     * Constructor - assigns a value to the parent class's cube field
     *
     * @param cube
     *         the cube for which the solution will be generated
     */
    public OrientationSolver(Cube cube) {
        super(cube);
    }

    /**
     * Performs and records the moves required to solve the orientation of
     * <b>cube</b>
     */
    public void solveOrientation() {
        rotateToTop(Color.yellow);
        solveEdgeOrientation();
        solveCornerOrientation();
    }

    /**
     * Performs and records the moves required to solve the edge orientation of
     * <b>cube</b>
     */
    private void solveEdgeOrientation() {
        int numEdgesOriented = getNumOriented();
        String moves = "";

        if (numEdgesOriented < 4) {
            if (numEdgesOriented == 0) {
                moves = "F R U R' U' F' U2 F U R U' R' F'";
                solutionExplanation += "\nThere are no edges oriented, so orient two edges using F R U R' U' F' ";
                solutionExplanation += "then orient the remaining edges using U2 F U R U' R' F' \n";
            }
        }
    }
}
```

```
// cube.performAbsoluteMoves("M U R U R' U' M2 U R U' R w'");  
} else {  
    // while ((isEdgeOriented(2)) || (!isEdgeOriented(3)))  
    while ((cube.getEdge(2).getOrientation() == 0) || (cube.getEdge(3).getOrientation() == 1)) {  
        cube.performAbsoluteMoves("U");  
        catalogMoves("U");  
    }  
  
    // if (!isEdgeOriented(0))  
    if (cube.getEdge(0).getOrientation() == 1) {  
        moves = "F R U R' U' F'";  
        solutionExplanation += "Here we have the \"bar\" formation, so perform F R U R' U' F' \n";  
    } else {  
        moves = "F U R U' R' F'";  
        solutionExplanation += "Here we have the \"r\" formation, so perform F U R U' R' F' \n";  
    }  
    cube.performAbsoluteMoves(moves);  
    catalogMoves(moves);  
}  
}  
  
/**  
 * Performs the moves required to solve the corner orientation of  
 * <b>cube</b>  
 */  
private void solveCornerOrientation() {  
    // Stores the orientation of the current corner being examined.  
    int orientation = 0;  
    // Stores the number of trailing U moves performed.  
    int numTrailing;  
    // Stores the moves to be performed after inspection of the corner.  
    String moves;  
  
    for (int i = 0; i < 4; ++i) {  
        moves = "";  
        orientation = cube.getCorner(2).getOrientation();  
        if (orientation == 1) {  
            moves = "R' D' R D R' D' R D";  
            solutionExplanation += String.format(  
                "Bring the %s-%s-%s corner to the URF position then twist it anti-clockwise using %s%n",  
                Cubie.getColorToWord(cube.getCorner(2).getStickers()[0]),  
                Cubie.getColorToWord(cube.getCorner(2).getStickers()[1]),  
                Cubie.getColorToWord(cube.getCorner(2).getStickers()[2]), "R' D' R D R' D' R D");  
        }  
    }  
}
```

```
        } else if (orientation == -1) {
            moves = ("D' R' D R D' R' D R");
            solutionExplanation += String.format(
                "Bring the %s-%s-%s corner to the URF position then twist it clockwise using %s%n",
                Cubie.getColorToWord(cube.getCorner(2).getStickers()[0]),
                Cubie.getColorToWord(cube.getCorner(2).getStickers()[1]),
                Cubie.getColorToWord(cube.getCorner(2).getStickers()[2]), "D' R' D R D' R' D R");
        }

        cube.performAbsoluteMoves(moves);
        cube.performAbsoluteMoves("U");
        catalogMoves(moves + " U");
    }

    /*
     * Undo unnecessary U moves
     */
    numTrailing = getNumTrailingU();
    for (int i = 0; i < numTrailing; ++i) {
        cube.performAbsoluteMoves("U'");
    }
}

/***
 * @return the number of edges in the top layer are oriented correctly
 */
private int getNumOriented() {
    int numOriented = 0;

    for (int i = 0; i < 4; ++i) {
        // if (cube.getEdge(i).getStickers()[0].equals(Color.yellow))
        if (cube.getEdge(i).getOrientation() == 0)
            ++numOriented;
    }

    return numOriented;
}

/***
 * Determines whether the edge at the specified index is oriented correctly
 *
 * @param edgeIndex
 *          the index of the Edge to be analysed
 */
```

```
* @return <b>true</b> if the edge at <b>index</b> is oriented correctly; <br>
*           <b>false</b> otherwise
*/
private boolean isEdgeOriented(int edgeIndex) {
    return (cube.getEdge(edgeIndex).getStickers() [0].equals(Color.yellow));
}

/**
 * Determines whether the corner at the specified index is oriented
 * correctly
 *
 * @param cornerIndex
 *         the index of the Corner to be analysed
 * @return <b>true</b> if the Corner at <b>index</b> is oriented correctly; <br>
 *           <b>false</b> otherwise
 */
private boolean isCornerOriented(int cornerIndex) {
    return (cube.getCorner(cornerIndex).getStickers() [0].equals(Color.yellow));
}

/**
 * @return <b>true</b> if all pieces in the top layer are oriented
 *         correctly; <br>
 *           <b>false</b> otherwise
 */
public boolean isOrientationSolved() {
    for (int i = 0; i < 4; ++i) {
        if (!isEdgeOriented(i) || !isCornerOriented(i))
            return false;
    }

    return true;
}

/**
 * @return <b>true</b> if all Edges are oriented correctly; <br>
 *           <b>false</b> otherwise
 */
public boolean isEdgeOrientationSolved() {
    Color[] stickers;

    for (int i = 0; i < 4; ++i) {
        stickers = Edge.getInitialStickers(i + 8);
```

```
    if (!isEdgeOriented(getIndexOf(new Edge(stickers))))
        return false;
    }

    return true;
}

/**
 * @return <b>true</b> if all Corners are oriented correctly; <br>
 *         <b>false</b> otherwise
 */
public boolean isCornerOrientationSolved() {
    Color[] stickers;

    for (int i = 0; i < 4; ++i) {
        stickers = Corner.getInitialStickers(i + 4);

        if (cube.getCorner(getIndexOf(new Corner(stickers))).getOrientation() != 0)
            return false;
    }

    return true;
}
}
```

## Class PermutationSolver

```
package jCube;

import java.awt.Color;

/**
 * @author Kelsey McKenna
 */
public class PermutationSolver extends SolveMaster {

    /**
     * This stores Edge objects representing the Yellow-Green, Yellow-Orange,
     * Yellow-Blue, Yellow-Red edges.
     */
    private Edge[] topEdges = new Edge[4];

    /**
     * Constructor - assigns a value to the cube field in the parent class and
     * assigns Edges to the topEdges array
     *
     * @param cube
     *         the Cube for which a solution will be generated
     */
    public PermutationSolver(Cube cube) {
        super(cube);

        for (int i = 0; i < 4; ++i) {
            topEdges[i] = new Edge(Edge.getInitialStickers(i + 8));
        }
    }

    /**
     * Performs and records the moves required to solve the permutation of
     * <b>cube</b>
     */
    public void solvePermutation() {
        rotateToTop(Color.yellow);
        solveCornerPermutation();
        solveEdgePermutation();
        performAUF();
    }

    /**

```

```
* After solving the relative permutation of the pieces, this method
* performs the move "U" until the cube is solved.
*/
private void performAUF() {
    while (!cube.getEdge(0).getSecondaryColor().equals(cube.getSlice(5).getCentre())) {
        cube.performAbsoluteMoves("U");
        catalogMoves("U");
    }
}

/**
 * Performs and records the moves required to solve the corner permutation
 * of <b>cube</b>
 */
private void solveCornerPermutation() {
    int count = 0;
    if (isCornerPermutationSolved()) {
        return;
    }

    while ((count < 4) && (!headlightsAtBack())) {
        ++count;
        cube.performAbsoluteMoves("U");
        catalogMoves("U");
    }

    if (count < 4) { // headlights found
        solutionExplanation += String.format(
            "Bring the %s headlights to the back then perform L' U R' D2 R U' R' D2 R L",
            Cubie.getColorToWord(cube.getCorner(0).getStickers()[2]));
        cube.performAbsoluteMoves("L' U R' D2 R U' R' D2 R L");
        catalogMoves("L' U R' D2 R U' R' D2 R L");
    } else {
        /*
         * if (!cube.getCorner(2).getStickers()[2].equals(cube.getEdge(1).
         * getStickers()[1])) { cube.performAbsoluteMoves("U");
         * catalogMoves("U"); }
         */
        solutionExplanation += "There are no headlights, so we must perform x' R U' R' D R U R' D' R U R' D R U' R' D' x";
        cube.performAbsoluteMoves("x' R U' R' D R U R' D' R U R' D R U' R' D' x");
        catalogMoves("x' R U' R' D R U R' D' R U R' D R U' R' D' x");
    }
}

solutionExplanation += "\n";
```

```
}

/**
 * Performs and records the moves required to solve the edge permutation of
 * <b>cube</b>
 */
private void solveEdgePermutation() {
    // Stores the number of U moves performed so that no more than 3 are
    // performed during the initial inspection.
    int count = 0;
    // Stores the moves to be performed after inspection of the state.
    String moves = "";

    if (isEdgePermutationSolved()) {
        return;
    }

    // Get solid block on back
    while ((count < 4) && (!cube.getEdge(0).getSecondaryColor().equals(cube.getCorner(0).getStickers()[2]))) {
        ++count;
        cube.performAbsoluteMoves("U");
        catalogMoves("U");
    }

    if (count < 4) { // if block on back
        solutionExplanation += String.format("Bring the %s block to the back and then cycle the pieces ",
            Cubie.getColorToWord(cube.getEdge(0).getSecondaryColor()));

        if (edgesAreOpposite(cube.getEdge(1), cube.getEdge(2))) {
            solutionExplanation += String.format("anti-clockwise using R U' R U R U R U' R' U' R2");
            moves = ("R U' R U R U R U' R' U' R2");
        } else {
            solutionExplanation += String.format("clockwise using R2 U R U R' U' R' U' R' U R'");
            moves = ("R2 U R U R' U' R' U' R' U R'");
        }
    } else if ((cube.getEdge(0).getSecondaryColor().equals(cube.getCorner(3).getStickers()[1])) && (cube.getEdge(1).getSecondaryColor().equals(cube.getCorner(3).getStickers()[2]))) {
        solutionExplanation += "All edges needed swapped vertically or horizontally, so we must perform M2 U M2 U2 M2 U M2";
        moves = ("M2 U M2 U2 M2 U M2");
    } else {
        solutionExplanation += "The edges need swapped in a z formation, so ";
        if (!cube.getEdge(2).getSecondaryColor().equals(cube.getCorner(1).getStickers()[2])) {
            moves = "U ";
            solutionExplanation += "set them up by performing U then ";
        }
    }
}
```

```
        }

        solutionExplanation += "perform M2 U M2 U M' U2 M2 U2 M'";
        moves += "M2 U M2 U M' U2 M2 U2 M'";
    }

    cube.performAbsoluteMoves(moves);
    catalogMoves(moves);
}

/**
 * @return <b>true</b> if the permutation of the Edges of the last layer is
 *         solved; <br>
 *         <b>false</b> otherwise
 */
public boolean isEdgePermutationSolved() {
    for (int i = 0; i < 4; ++i) {
        if (!cube.getEdge(0).getSecondaryColor().equals(cube.getCorner(0).getStickers()[2]))
            return false;

        cube.performAbsoluteMoves("U");
        catalogMoves("U");
    }

    return true;
}

/**
 * @param edgeOne
 *         this Edge is compared to the second Edge
 * @param edgeTwo
 *         this Edge is compared to the first Edge
 * @return <b>true</b> if the edges are opposite each other and are on the
 *         same face on a solved cube; <br>
 *         <b>false</b> otherwise
 */
private boolean edgesAreOpposite(Edge edgeOne, Edge edgeTwo) {
    Color colorOne, colorTwo;

    colorOne = edgeOne.getSecondaryColor();
    colorTwo = edgeTwo.getSecondaryColor();

    int one = -1, two = -2;
```

```
for (int i = 0; i < 4; ++i) {
    if (topEdges[i].getSecondaryColor().equals(colorOne))
        one = i;
    else if (topEdges[i].getSecondaryColor().equals(colorTwo))
        two = i;
}

return ((one + two) % 2 == 0);
}

/**
 * @return <b>true</b> if the permutation of the Corners of the last layer
 *         is solved; <br>
 *         <b>false</b> otherwise
 */
public boolean isCornerPermutationSolved() {
    for (int i = 0; i < 4; ++i) {
        if (!headlightsAtBack())
            return false;

        cube.performAbsoluteMoves("U");
        catalogMoves("U");
    }

    return true;
}

/**
 * @return <b>true</b> if there are 'headlights' at the back face of the
 *         cube, i.e. two matching Corner stickers; <br>
 *         <b>false</b> otherwise
 */
private boolean headlightsAtBack() {
    if (cube.getCorner(0).getStickers()[2].equals(cube.getCorner(1).getStickers()[1]))
        return true;

    return false;
}

/**
 * @return <b>true</b> if the permutation of all pieces in the top layer is
 *         solved; <br>
 *         <b>false</b> otherwise
*/
```

```
public boolean permutationSolved() {  
    return (isEdgePermutationSolved() && isCornerPermutationSolved());  
}  
}
```

## Class Preferences

```
package jCube;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swingBoxLayout;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JTextField;

/**
 * @author Kelsey McKenna
 */
public class Preferences extends JFrame implements ActionListener {
    /**
     * Default serialVersionUID
     */
    private static final long serialVersionUID = 1L;

    /**
     * This indicates the vertical spacing between the elements in the window.
     */
    private static final int ySpacing = 40;
    /**
     * This stores the padding of the elements in the window. The greater the
     * padding, the further towards the centre of the window the elements will
     * be.
     */
    private static final int pad = 5;
    /**
     * This stores the width of the window.
     */
}
```

```
private static final int FRAME_WIDTH = 400;
/**
 * This stores (FRAME_WIDTH/2) and is used to identify the middle
 * x-coordinate of the window.
 */
private static final int MIDDLE_X = FRAME_WIDTH / 2;
/**
 * This stores the width of each label and text field added to the window.
 */
private static final int COMPONENT_WIDTH = MIDDLE_X - pad - 5;
/**
 * This stores the height of each label and text field added to the window.
 */
private static final int COMPONENT_HEIGHT = 40;
/**
 * This stores the default real-time solving speed.
 */
private static final int REAL_TIME_SOLVING_SPEED = 250;
/**
 * This stores the default inspection time.
 */
private static final int INSPECTION_TIME = 15;
/**
 * This stores the default scramble text size.
 */
private static final int SCRAMBLE_TEXT_SIZE = 27;

/**
 * This stores all components of the window other than buttons.
 */
private JPanel fieldPanel;
/**
 * This stores the buttons in the window.
 */
private JPanel buttonPanel;

/**
 * This label is shown in the window with the text 'Real-time solving speed
 * (ms)'.
 */
private JLabel realTimeSolvingRateLabel;
/**
 * This label is shown in the window with the text 'Inspection time
 * (seconds)'.
```

```
/*
private JLabel inspectionTimeLabel;
/**
 * This label is shown in the window with the text 'Show Click-to-Solve
 * warning'.
*/
private JLabel solvePieceWarningLabel;
/**
 * This label is shown in the window with the text 'Scramble text size'.
*/
private JLabel scrambleTextSizeLabel;

/**
 * This field is shown in the window and the user can enter the speed into
 * this field.
*/
private JTextField realTimeSolvingSpeedField;
/**
 * This field is shown in the window and the user can enter the inspection
 * time into this field.
*/
private JTextField inspectionTimeField;
/**
 * This field is shown in the window and the user can enter the scramble
 * text size this field.
*/
private JTextField scrambleTextSizeField;

/**
 * This button group stores the solvePieceWarningYesItem and the
 * solvePieceWarningNoItem radio buttons. Using the button group, only one
 * of the two radio buttons can be selected, not both.
*/
private static ButtonGroup solvePieceWarningButtonGroup;
/**
 * By selecting this radio button, a warning/information message will be
 * shown when 'Click to Solve' mode is enabled. If the
 * solvePieceWarningNoItem is selected, then no message will be shown.
*/
private static JRadioButton solvePieceWarningYesItem;
/**
 * By selecting this radio button, the warning/information message will not
 * be shown when 'Click to Solve' mode is enabled. If the
 * solvePieceWarningYesItem is selected, then the message will be shown.
```

```
/*
private static JRadioButton solvePieceWarningNoItem;

/**
 * Clicking this button submits the data in the window for validation. If
 * the data is valid, then the window Preferences window will close.
 */
private JButton saveAndCloseButton;
/**
 * Clicking this button discards any unsaved changes and closes the window.
 */
private JButton cancelButton;
/**
 * Clicking this button restores the default preferences and updates the
 * fields accordingly.
 */
private JButton restoreDefaultsButton;

/**
 * This stores the current real-time solving speed saved in the preferences
 * file.
 */
private static int realTimeSolvingSpeed = REAL_TIME_SOLVING_SPEED;
/**
 * This stores the current inspection time saved in the preferences file.
 */
private static int inspectionTime = INSPECTION_TIME;
/**
 * This stores the current scramble text size saved in the preferences file.
 */
private static int scrambleTextSize = SCRAMBLE_TEXT_SIZE;

/**
 * Constructor - sets up the 'Preferences' window
 */
public Preferences() {
    super("Preferences");

    fieldPanel = new JPanel();
    fieldPanel.setLayout(null);

    setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
    getContentPane().setLayout(new BoxLayout(getContentPane(), BoxLayout.PAGE_AXIS));
    // setPreferredSize(new Dimension(FRAME_WIDTH, FRAME_HEIGHT));
}
```

```
setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
setResizable(false);
 setLocation(600, 150);
setVisible(false);
addWindowListener(new WindowListener() {

    @Override
    public void windowActivated(WindowEvent arg0) {
    }

    @Override
    public void windowClosed(WindowEvent arg0) {
        populateFieldsWithValues();
    }

    @Override
    public void windowClosing(WindowEvent arg0) {
    }

    @Override
    public void windowDeactivated(WindowEvent arg0) {
    }

    @Override
    public void windowDeiconified(WindowEvent arg0) {
    }

    @Override
    public void windowIconified(WindowEvent arg0) {
    }

    @Override
    public void windowOpened(WindowEvent arg0) {
    }

});

int y = 0;

fieldPanel = new JPanel();
fieldPanel.setLayout(null);

***** FIELDS AND LABELS *****
```

```
realTimeSolvingRateLabel = new JLabel("Real-time solving speed (ms)");
realTimeSolvingRateLabel.setSize(COMPONENT_WIDTH, COMPONENT_HEIGHT);
realTimeSolvingRateLabel.setLocation(0 + pad, y + pad);

realTimeSolvingSpeedField = new JTextField();
realTimeSolvingSpeedField.setSize(COMPONENT_WIDTH, 40);
realTimeSolvingSpeedField.setLocation(MIDDLE_X, y + pad);
realTimeSolvingSpeedField.addActionListener(this);
realTimeSolvingSpeedField.setToolTipText("0 < x < 10000");

y += ySpacing;

inspectionTimeLabel = new JLabel("Inspection time (seconds)");
inspectionTimeLabel.setSize(COMPONENT_WIDTH, COMPONENT_HEIGHT);
inspectionTimeLabel.setLocation(0 + pad, y + pad);

inspectionTimeField = new JTextField();
inspectionTimeField.setSize(COMPONENT_WIDTH, COMPONENT_HEIGHT);
inspectionTimeField.setLocation(MIDDLE_X, y + pad);
inspectionTimeField.addActionListener(this);
inspectionTimeField.setToolTipText("0 < x < 100");

y += ySpacing;

solvePieceWarningLabel = new JLabel();
solvePieceWarningLabel.setText("Show Click-to-Solve warning");
solvePieceWarningLabel.setSize(COMPONENT_WIDTH, COMPONENT_HEIGHT);
solvePieceWarningLabel.setLocation(0 + pad, y + pad);

solvePieceWarningButtonGroup = new ButtonGroup();

solvePieceWarningYesItem = new JRadioButton("Yes");
solvePieceWarningYesItem.setEnabled(true);
solvePieceWarningYesItem.setFocusable(false);
solvePieceWarningYesItem.setSize(COMPONENT_WIDTH / 2, COMPONENT_HEIGHT);
solvePieceWarningYesItem.setLocation(MIDDLE_X, y + pad);

solvePieceWarningNoItem = new JRadioButton("No");
solvePieceWarningNoItem.setFocusable(false);
solvePieceWarningNoItem.setSize(COMPONENT_WIDTH / 2, COMPONENT_HEIGHT);
solvePieceWarningNoItem.setLocation(MIDDLE_X + COMPONENT_WIDTH / 2, y + pad);

solvePieceWarningButtonGroup.add(solvePieceWarningYesItem);
solvePieceWarningButtonGroup.add(solvePieceWarningNoItem);
```

```
y += ySpacing;

scrambleTextSizeLabel = new JLabel();
scrambleTextSizeLabel.setText("Scramble text size");
scrambleTextSizeLabel.setSize(COMPONENT_WIDTH, COMPONENT_HEIGHT);
scrambleTextSizeLabel.setLocation(0 + pad, y + pad);

scrambleTextSizeField = new JTextField(scrambleTextSize + "");
scrambleTextSizeField.setSize(COMPONENT_WIDTH, COMPONENT_HEIGHT);
scrambleTextSizeField.setLocation(MIDDLE_X, y + pad);
scrambleTextSizeField.addActionListener(this);
scrambleTextSizeField.setToolTipText("0 < x < 100");

y += ySpacing;

fieldPanel.add(realTimeSolvingRateLabel);
fieldPanel.add(realTimeSolvingSpeedField);
fieldPanel.add(inspectionTimeLabel);
fieldPanel.add(inspectionTimeField);
fieldPanel.add(solvePieceWarningLabel);
fieldPanel.add(solvePieceWarningYesItem);
fieldPanel.add(solvePieceWarningNoItem);
fieldPanel.add(scrambleTextSizeLabel);
fieldPanel.add(scrambleTextSizeField);

fieldPanel.setPreferredSize(new Dimension(FRAME_WIDTH, y + 30));
/**************** END FIELDS AND LABELS *******/

/**************** BUTTON PANE *****/
buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout());

buttonPanel.setSize(0, 30);

saveAndCloseButton = new JButton("Save and Close");
saveAndCloseButton.setFocusable(false);
saveAndCloseButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        boolean valid = true;

        try {
            realTimeSolvingSpeed = (int) Double.parseDouble(realTimeSolvingSpeedField.getText().trim());
        } catch (NumberFormatException ex) {
            valid = false;
        }

        if (!valid) {
            JOptionPane.showMessageDialog(buttonPanel, "Please enter a valid integer value for the solving speed.");
        }
    }
});
```

```
        if ((realTimeSolvingSpeed <= 0) || (realTimeSolvingSpeed >= 10000)) {
            throw new Exception();
        }

        realTimeSolvingSpeed = Math.abs(realTimeSolvingSpeed);
        realTimeSolvingSpeedField.setBackground(Color.white);
    } catch (Exception exc) {
        realTimeSolvingSpeedField.setBackground(new Color(255, 150, 150));
        valid = false;
    }

    try {
        inspectionTime = (int) Double.parseDouble(inspectionTimeField.getText().trim());

        if ((inspectionTime <= 0) || (inspectionTime >= 100)) {
            throw new Exception();
        }

        inspectionTimeField.setBackground(Color.white);
    } catch (Exception exc) {
        inspectionTimeField.setBackground(new Color(255, 150, 150));
        valid = false;
    }

    try {
        scrambleTextSize = (int) Double.parseDouble(scrambleTextSizeField.getText().trim());

        if ((scrambleTextSize <= 0) || (scrambleTextSize >= 100))
            throw new Exception();

        scrambleTextSizeField.setBackground(Color.white);
    } catch (Exception exc) {
        scrambleTextSizeField.setBackground(new Color(255, 150, 150));
        valid = false;
    }

    if (valid) {
        setVisible(false);
        savePreferences();
        populateFieldsWithValues();
        Main.setScrambleTextSize(scrambleTextSize);
    }
}
});
```

```
cancelButton = new JButton("Cancel");
cancelButton.setFocusable(false);
cancelButton.addActionListener(new ActionListener() {

    @Override
    public void actionPerformed(ActionEvent arg0) {
        setVisible(false);
        populateFieldsWithValues();
    }
});

restoreDefaultsButton = new JButton("Restore Defaults");
restoreDefaultsButton.setFocusable(false);
restoreDefaultsButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        restoreDefaults();
    }
});

buttonPanel.add(saveAndCloseButton);
buttonPanel.add(cancelButton);
buttonPanel.add(restoreDefaultsButton);

/**************** END BUTTON PANE *****/

populateFieldsWithValues();
getContentPane().add(fieldPanel);
getContentPane().add(buttonPanel);

pack();
}

/**
 * @return the real-time solving speed in milliseconds
 */
public int getRealTimeSolvingRate() {
    return realTimeSolvingSpeed;
}

/**
 * @return the time allowed to inspect the cube before a solve starts
 *         (seconds)

```

```
/*
 * @return <b>true</b> if the solvePieceWarningYesItem is selected; <br>
 *         <b>false</b> otherwise
 */
public boolean solvePieceWarningEnabled() {
    return solvePieceWarningYesItem.isSelected();
}

/**
 * @return the scramble text size as saved in the 'Preferences' window
 */
public int getScrambleTextSize() {
    return scrambleTextSize;
}

/**
 * This method is called when the enter key is pressed when typing data into
 * one of the text fields.
 *
 * @see java.awt.event.ActionListener#actionPerformed(java.awt.event.ActionEvent)
 */
@Override
public void actionPerformed(ActionEvent arg0) {
    saveAndCloseButton.doClick();
}

/**
 * Retrieves the data from preferences.txt and inserts it into the text
 * fields
 */
public void populateFieldsWithValues() {
    TextFile currentFile = new TextFile();
    try {
        currentFile.setFilePath("settings/preferences.txt");
        currentFile.setIO(TextFile.READ);

        String[] data = currentFile.readAllLines();
    }
}
```

```
realTimeSolvingSpeedField.setText(data[0].trim());
inspectionTimeField.setText(data[1].trim());
solvePieceWarningYesItem.setSelected((data[2].trim()).equalsIgnoreCase("Yes")) ? true : false);
solvePieceWarningNoItem.setSelected(!solvePieceWarningYesItem.isSelected());
scrambleTextSizeField.setText(data[3].trim());

try {
    realTimeSolvingSpeed = Integer.valueOf(data[0].trim());
    inspectionTime = Integer.valueOf(data[1].trim());
    scrambleTextSize = Integer.valueOf(data[3].trim());
} catch (Exception e) {

}

realTimeSolvingSpeedField.setBackground(Color.white);
inspectionTimeField.setBackground(Color.white);
scrambleTextSizeField.setBackground(Color.white);
} catch (Exception e) {
    JOptionPane.showMessageDialog(this, "Could not open preferences file. Restoring defaults.", "Error",
        JOptionPane.ERROR_MESSAGE);
    restoreDefaults();
} finally {
    currentFile.close();
}
}

/**
 * Restores and saves the system's preferences to its default settings
 */
private void restoreDefaults() {
    solvePieceWarningYesItem.setSelected(true);
    solvePieceWarningNoItem.setSelected(false);

    realTimeSolvingSpeedField.setText(String.format("%d", REAL_TIME_SOLVING_SPEED));
    realTimeSolvingSpeed = REAL_TIME_SOLVING_SPEED;
    realTimeSolvingSpeedField.setBackground(Color.white);

    inspectionTimeField.setText(String.format("%d", INSPECTION_TIME));
    inspectionTime = INSPECTION_TIME;
    inspectionTimeField.setBackground(Color.white);

    scrambleTextSize = SCRAMBLE_TEXT_SIZE;
    scrambleTextSizeField.setText("" + SCRAMBLE_TEXT_SIZE);
    scrambleTextSizeField.setBackground(Color.white);
```

```
solvePieceWarningEnabled();
Main.setScrambleTextSize(scrambleTextSize);
savePreferences();
}

/**
 * Writes the preferences to preferences.txt
 */
private void savePreferences() {
    TextFile currentFile = new TextFile();
    try {
        currentFile.setFilePath("settings/preferences.txt");
        currentFile.setIO(TextFile.WRITE);

        currentFile.writeLine(realTimeSolvingSpeed);
        currentFile.writeLine(inspectionTime);
        currentFile.writeLine((solvePieceWarningYesItem.isSelected()) ? "Yes" : "No");
        currentFile.writeLine(scrambleTextSize);
    } catch (Exception e) {
        JOptionPane.showMessageDialog(this, "Could not save preferences to file", "Error",
            JOptionPane.ERROR_MESSAGE);
    } finally {
        currentFile.close();
    }
}
}
```

## Class Scramble

```
package jCube;

/**
 * @author Kelsey McKenna
 */
public class Scramble {

    /**
     * This array stores all possible types of moves and their corresponding
     * rotation directions.
     */
    private static final String[] moves = { "RX", "LX", "UY", "DY", "FZ", "BZ" };
    /**
     * This stores the three different directions that can be attributed to a
     * move.
     */
    private static final String[] directions = { "", "'", "2" };

    /**
     * @return a randomly generated 25-move scramble
     */
    public static String generateScramble() {
        String s = "";

        String penultimateFace = " ";
        String lFace = " ";

        for (int i = 0; i < 25; ++i) {
            String newFace = getRandomFace(penultimateFace, lFace);
            s += newFace.charAt(0) + getRandomDirection() + " ";
            penultimateFace = lFace;
            lFace = newFace;
        }

        return s;
    }

    /**
     * @return a random element from the directions array
     */
    private static String getRandomDirection() {
        return directions[(int) (Math.random() * 3)];
    }
}
```

```
}

/**
 * @param penultimate
 *         the penultimate move in the scramble
 * @param last
 *         the last move in the scramble
 * @return a random face whose direction of motion is in the same axis
 *         compared with the penultimate and last moves. For example, R L F
 *         would be an acceptable series of moves because the directions of
 *         motion of these moves is x x z. U D U' would not be acceptable
 *         because the directions of motion of these moves is y y y, which
 *         are all the same so a new move must be generated.
 */
private static String getRandomFace(String penultimate, String last) {
    String toReturn = moves[(int) (Math.random() * moves.length)];

    if (last.equals(toReturn) || isSameAxis(penultimate, last, toReturn))
        return getRandomFace(penultimate, last);

    return toReturn;
}

/**
 * @param a
 *         the first move with its associated rotation direction
 * @param b
 *         the second move with its associated rotation direction
 * @param c
 *         the third move with its associated rotation direction
 * @return <b>true</b> if the three moves are in the same axis; <br>
 *         <b>false</b> otherwise
 */
private static boolean isSameAxis(String a, String b, String c) {
    return a.charAt(1) == b.charAt(1) && b.charAt(1) == c.charAt(1);
}

}
```

## Class ScramblePopUp

```
package jCube;

import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;

import javax.swingBoxLayout;
import javax.swing.DefaultListModel;
import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;

/**
 * @author Kelsey McKenna
 */
public class ScramblePopUp extends JFrame {
    /**
     * Default serialVersionUID
     */
    private static final long serialVersionUID = 1L;

    /**
     * The list of scrambles is stored in this panel.
     */
    private JPanel scrambleListPanel;
    /**
     * This stores the contents (scrambles) in the list.
     */
    private DefaultListModel<String> scrambleList;
    /**

```

```
* This is the list that is added to scrambleListPanel so that the contents
* of the list can be displayed.
*/
private JList<String> listHolder;
/**
 * listHolder is placed 'inside' this variable so that once the size of the
 * list exceeds the size of the window, the user can scroll to view the rest
 * of the list.
*/
private JScrollPane listScrollPane;

/**
 * The buttons of the window are stored on this panel.
*/
private JPanel buttonPanel;

/**
 * Clicking this button opens an input dialog into which the user can enter
 * a new scramble.
*/
private JButton addScrambleButton;
/**
 * Clicking this button opens an input dialog displaying the existing
 * scramble, which the user can edit.
*/
private JButton editScrambleButton;
/**
 * Clicking this button deletes the selected scrambles from the list
*/
private JButton deleteScrambleButton;
/**
 * Clicking this button opens a save dialog in which the user can choose a
 * location to save the selected scrambles
*/
private JButton saveScrambleToFileButton;
/**
 * Clicking this button opens a load dialog in which the user can select the
 * text file which contains the scrambles they want to load
*/
private JButton loadScrambleFromFileButton;
/**
 * Clicking this button applies hte selected scramble to the cube in the
 * main window and displays the scramble at the top of the main window
*/
```

```
private JButton applyScrambleButton;

/**
 * This stores the index of the next scramble to be used if the 'Use
 * Scrambles in List' option is selected in the main window.
 */
private int scrambleIndex = 0;

/**
 * This variable is used to select a location to save or load scrambles.
 */
private JFileChooser fileChooser = new JFileChooser(System.getProperty("user.home") + "/Desktop") {
    private static final long serialVersionUID = 1L;

    @Override
    public void approveSelection() {
        // String filePath = getSelectedFile() + ".txt";
        // filePath = filePath.substring(0, filePath.indexOf(".txt")) +
        // ".txt";
        setSelectedFile(new File((( "" + getSelectedFile().replace("\\\\.", "") ) + ".txt"));
        File f = new File(getSelectedFile().toString());

        if ((f.exists()) && (getDialogType() == SAVE_DIALOG)) {
            int result = JOptionPane.showConfirmDialog(this, String.format(
                "The file %s already exists. Do you want to overwrite?", getSelectedFile().toString()),
                "Existing File", JOptionPane.YES_NO_OPTION);
            switch (result) {
                case JOptionPane.YES_OPTION:
                    super.approveSelection();
                    return;
                case JOptionPane.NO_OPTION:
                    return;
                case JOptionPane.CLOSED_OPTION:
                    return;
            }
        }
        super.approveSelection();
    }
};

/**
 * Constructor - sets up the "Scramble List" window
 */
public ScramblePopUp() {
```

```
super("Scramble List");

initListPanel();

setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
setLayout(new BoxLayout(getContentPane(), BoxLayout.PAGE_AXIS));
setVisible(false);
setPreferredSize(new Dimension(500, 285));
setResizable(false);
setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
pack();

buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(2, 3));

addScrambleButton = new JButton("Add Scramble");
addScrambleButton.setFocusable(false);
addScrambleButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String scramble = JOptionPane.showInputDialog(null, "Enter Scramble", "R U R' U'");
        if (scramble != null && (!scramble.trim().equals(""))) {
            scrambleList.addElement(" " + scramble.trim());
            listHolder.ensureIndexIsVisible(scrambleList.getSize() - 1);
            listHolder.setSelectedIndex(scrambleList.getSize() - 1);
        }
    }
});

editScrambleButton = new JButton("Edit Scramble");
editScrambleButton.setFocusable(false);
editScrambleButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int selectedIndex = listHolder.getSelectedIndex();
        if (selectedIndex == -1)
            return;
        else {
            String scramble = scrambleList.get(selectedIndex);
            scramble = JOptionPane.showInputDialog(null, "Enter New Scramble", scramble.trim());
            if (scramble != null && (!scramble.trim().equals("")))
                scrambleList.set(selectedIndex, " " + scramble.trim());
        }
    }
});
```

```
        }
    });
});

deleteScrambleButton = new JButton("Delete Scramble");
deleteScrambleButton.setFocusable(false);
deleteScrambleButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        int selectedIndex = listHolder.getSelectedIndex();

        if (selectedIndex == -1)
            return;
        else {
            int[] selectedIndices = listHolder.getSelectedIndices();

            for (int i = selectedIndices.length - 1; i >= 0; --i) {
                scrambleList.remove(selectedIndices[i]);
                listHolder.clearSelection();
            }

            if (scrambleList.size() > 0)
                listHolder.setSelectedIndex((selectedIndex < scrambleList.size()) ? selectedIndex
                    : selectedIndex - 1);
        }
    }
});
}

applyScrambleButton = new JButton("Apply Scramble");
applyScrambleButton.setFocusable(false);
applyScrambleButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int selectedIndex = listHolder.getSelectedIndex();

        if (selectedIndex == -1)
            return;
        else
            Main.handleScramble(scrambleList.get(selectedIndex));
    }
});
}

saveScrambleToFileButton = new JButton("Save Selected to File");
saveScrambleToFileButton.setFocusable(false);
saveScrambleToFileButton.addActionListener(new ActionListener() {
```

```
public void actionPerformed(ActionEvent e) {
    int selectedIndex = listHolder.getSelectedIndex();

    if (selectedIndex == -1)
        return;
    else {
        int[] selectedIndices = listHolder.getSelectedIndices();

        if (fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
            TextFile currentFile = new TextFile();

            try {
                currentFile.setFilePath(fileChooser.getSelectedFile().toString());
                currentFile.setIO(TextFile.WRITE);

                for (int i = 0; i < selectedIndices.length; ++i) {
                    currentFile.writeLine(scrambleList.get(selectedIndices[i]).trim());
                }
            } catch (Exception exc) {
                JOptionPane.showMessageDialog(scrambleListPanel, "Could not save to file", "Error",
                    JOptionPane.ERROR_MESSAGE);
            } finally {
                currentFile.close();
            }
        }
    }
};

loadScrambleFromFileButton = new JButton("Load Scrambles");
loadScrambleFromFileButton.setFocusable(false);
loadScrambleFromFileButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            if (fileChooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
                String current;
                int numLines;

                TextFile currentFile = new TextFile();
                currentFile.setFilePath(fileChooser.getSelectedFile().toString());
                currentFile.setIO(TextFile.READ);
                numLines = currentFile.getNumLines();

                for (int i = 0; i < numLines; ++i) {
```

```
        current = currentFile.readLine().trim();

        if (!current.equals("")) {
            scrambleList.addElement(" " + current);
        }
    }

    currentFile.close();
}
} catch (Exception exc) {
    JOptionPane.showMessageDialog(scrambleListPanel, "Error Reading File", "Error",
        JOptionPane.ERROR_MESSAGE);
}
}

buttonPanel.add(addScrambleButton);
buttonPanel.add(editScrambleButton);
buttonPanel.add(deleteScrambleButton);
buttonPanel.add(applyScrambleButton);
buttonPanel.add(saveScrambleToFileButton);
buttonPanel.add(loadScrambleFromFileButton);

getContentPane().add(buttonPanel);
}

/**
 * Initialises the components associated with the list of scrambles in the
 * window. Fonts, selection modes, and contents are declared here.
 */
private void initListPanel() {
    scrambleListPanel = new JPanel();
    scrambleList = new DefaultListModel<String>();

    listHolder = new JList<String>(scrambleList);
    listHolder.setFont(new Font("Arial", 0, 15));
    listHolder.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
    listHolder.setLayoutOrientation(JList.VERTICAL);
    listHolder.setSelectedIndex(0);
    listHolder.addMouseListener(new MouseListener() {
        public void mouseClicked(MouseEvent e) {
            if ((e.getClickCount() == 2) && (scrambleList.size() > 0)) {
                int selectedIndex = listHolder.getSelectedIndex();
                ...
            }
        }
    });
}
```

```
        if (selectedIndex == -1)
            return;
        else {
            String scramble = scrambleList.get(selectedIndex);
            scramble = JOptionPane.showInputDialog(null, "Enter New Scramble", scramble.trim());

            if (scramble != null && (!scramble.trim().equals("")))
                scrambleList.set(selectedIndex, " " + scramble.trim());
        }
    }

@Override
public void mouseEntered(MouseEvent e) {

}

@Override
public void mouseExited(MouseEvent e) {

}

@Override
public void mousePressed(MouseEvent e) {

}

@Override
public void mouseReleased(MouseEvent e) {
}
);

listHolder.addKeyListener(new KeyListener() {
    @Override
    public void keyPressed(KeyEvent arg0) {

    }

    @Override
    public void keyReleased(KeyEvent arg0) {

    }

    @Override
    public void keyTyped(KeyEvent e) {
        int selectedIndex = listHolder.getSelectedIndex();

        if (selectedIndex == -1)
```

```
        return;

        if (e.getKeyChar() == KeyEvent.VK_DELETE) {
            int[] selectedIndices = listHolder.getSelectedIndices();

            for (int i = 0; i < selectedIndices.length; ++i) {
                scrambleList.remove(selectedIndex);
                listHolder.clearSelection();
            }

            if (scrambleList.size() > 0)
                listHolder.setSelectedIndex((selectedIndex < scrambleList.size()) ? selectedIndex
                    : selectedIndex - 1);
            } else if (e.getKeyChar() == KeyEvent.VK_ENTER) {
                String scramble = scrambleList.get(selectedIndex);
                scramble = JOptionPane.showInputDialog(null, "Enter New Scramble", scramble.trim());

                if (scramble != null && (!scramble.trim().equals("")))
                    scrambleList.set(selectedIndex, " " + scramble.trim());
            }
        }
    });
}

listScrollPane = new JScrollPane(listHolder);
listScrollPane.setPreferredSize(new Dimension(500, 200));
scrambleListPanel.add(listScrollPane);

listHolder.setFocusable(true);

getContentPane().add(scrambleListPanel);
}

/**
 * @return the next scramble in the list
 * @throws IndexOutOfBoundsException
 *          if there are no elements in the list
 */
public String getCurrentScramble() throws IndexOutOfBoundsException {
    if (scrambleIndex >= scrambleList.size())
        scrambleIndex = 0;

    String currentScramble = scrambleList.get(scrambleIndex);
    scrambleIndex = (scrambleIndex + 1) % scrambleList.size();
```

```
    return currentScramble;
}
}
```

## Class Slice

```
package jCube;

import java.awt.Color;
import java.util.Arrays;

/*
 * 4 corners and 4 edges per slice
 * Corners are indexed like this:
 *      0 1
 *      2 3
 * Edges are indexed like this
 *      0
 *      3  1
 *      2
 */
/***
 * @author Kelsey McKenna
 */
public class Slice {
    /**
     * This variable stores the number '4' and is used for readability in code.
     */
    private static final int NUM_EDGES = 4; // Readability
    /**
     * This variable stores the number '4' and is used for readability in code.
     */
    private static final int NUM_CORNERS = 4; // Readability

    /**
     * This array stores the Edges in the slice.
     */
    private Edge[] edges = new Edge[NUM_EDGES];
    /**
     * This arrays stores Corners in the slice.
     */
    private Corner[] corners = new Corner[NUM_CORNERS];
    /**
     * This array stores the colour of the centre of the slice.
     */
    private Color centre;
```

```
/**  
 * This array stores the indices of the cubies in relation to the other  
 * pieces of the cubie. The first four elements are corner indices; the last  
 * four are edge indices.  
 */  
private int[] cubieIndices;  
  
/**  
 * Constructor - assigns cubie indices to pieces  
 *  
 * @param cubieIndices  
 *          the first four elements should be corner indices and the next  
 *          four should be edges indices.  
 */  
public Slice(int[] cubieIndices) {  
    this.cubieIndices = cubieIndices;  
  
    for (int i = 0; i < 4; ++i) {  
        edges[i] = new Edge();  
        corners[i] = new Corner();  
        corners[i].setCubieIndex(cubieIndices[i]);  
        edges[i].setCubieIndex(cubieIndices[i + 4]);  
    }  
}  
  
/**  
 * Constructor - assigns Edges and Corners to the slice  
 *  
 * @param edges  
 *          the Edges of the slice  
 * @param corners  
 *          the Corners of the slice  
 */  
public Slice(Edge[] edges, Corner[] corners) {  
    this.edges = edges;  
    this.setCorners(corners);  
}  
  
/**  
 * Sets the cubie indices of the pieces  
 *  
 * @param cubieIndices  
 *          the cubie indices to be assigned to the pieces. The first four  
 *          elements should be for the Corners and the next four should be
```

```
*           for the Edges.  
*/  
public void setCubieIndices(int[] cubieIndices) {  
    for (int i = 0; i < 4; ++i) {  
        corners[i].setCubieIndex(cubieIndices[i]);  
        edges[i].setCubieIndex(cubieIndices[i + 4]);  
    }  
}  
  
/**  
 * Sets the Edges of the slice to the specified Edges  
 *  
 * @param edges  
 *          the Edges to be assigned to the slice  
 */  
public void setEdges(Edge[] edges) {  
    this.edges = edges;  
}  
  
/**  
 * Sets the Corners of the slice to the specified Corners  
 *  
 * @param corners  
 *          the Corners to be assigned to the slice  
 */  
public void setCorners(Corner[] corners) {  
    this.corners = corners;  
}  
  
/**  
 * Sets the Edge at the specified index to the specified Edge  
 *  
 * @param index  
 *          the index of the Edge to be changed  
 * @param edge  
 *          the Edge to be assigned to the specified index  
 */  
public void setEdge(int index, Edge edge) {  
    edges[index] = edge;  
}  
  
/**  
 * Sets the Corner at the specified index to the specified Corner  
 */
```

```
* @param index
*          the index of the Corner to be changed
* @param corner
*          the Corner to be assigned to the specified index
*/
public void setCorner(int index, Corner corner) {
    corners[index] = corner;
}

/**
 * Sets the centre to the specified Color
 *
 * @param centre
 *          the Color to be set as the centre
 */
public void setCentre(Color centre) {
    this.centre = centre;
}

/**
 * @return an array containing the Edges of the slice
 */
public Edge[] getEdges() {
    return edges;
}

/**
 * @return an array containing the Corners of the slice
 */
public Corner[] get Corners() {
    return corners;
}

/**
 * @param index
 *          the index of the Edge to be returned
 * @return the <b>index</b>th Edge
 */
public Edge getEdge(int index) {
    return edges[index];
}

/**
 * @param index
```

```
*           the index of the Corner to be returned
* @return the <b>index</b>th Corner
*/
public Corner getCorner(int index) {
    return corners[index];
}

/**
 * @return the centre Color
 */
public Color getCentre() {
    return centre;
}

/**
 * @return an array containing the indices for all cubies
 */
public int[] getCubieIndices() {
    return this.cubieIndices;
}

/**
 * Performs a move on the slice in the specified direction <br>
 * 1 - Clockwise 2 - 180 degree -1 - Anticlockwise
 *
 * This method handles an argument of '2'
 *
 * @param direction
 *          the direction in which to move the slice
 */
public void performMove(int direction) {
    int tempDirection = (direction == 2) ? 1 : direction;

    for (int i = 0; i < (int) Math.abs(direction); ++i)
        pMove(tempDirection);
}

/**
 * Performs a move on the slice in the specified direction <br>
 * 1 - Clockwise -1 - Anticlockwise
 *
 * @param direction
 *          the direction in which to move the slice
 */
```

```
private void pMove(int direction) {
    // Stores a copy of the stickers of the last edge in the swap-cycle.
    Color[] tempEdgeStickers = Arrays.copyOf(edges[0].getStickers(), 2);
    // Stores a copy of the stickers of the last corner in the swap-cycle.
    Color[] tempCornerStickers = Arrays.copyOf(corners[0].getStickers(), 3);
    // Stores a copy of the orientation of the last edge in the swap-cycle.
    int tempEOrientation = edges[0].getOrientation();
    // Stores a copy of the orientation of the last corner in the
    // swap-cycle.
    int tempCOrientation = corners[0].getOrientation();
    // Stores index of the next cubie in the swap cycle.
    int nextIndex = 0, end = (4 + direction) % 4;

    // Cycles around the stickers in the specified direction.
    for (int i = 0; i != end; i = (i - direction + 4) % 4) {
        nextIndex = (i - direction + 4) % 4;
        edges[i].setStickers(edges[nextIndex].getStickers());
        edges[i].setOrientation(edges[nextIndex].getOrientation());
        corners[i].setStickers(corners[nextIndex].getStickers());
        corners[i].setOrientation(corners[nextIndex].getOrientation());
    }

    // Completes the cycle
    edges[end].setStickers(tempEdgeStickers);
    edges[end].setOrientation(tempEOrientation);
    corners[end].setStickers(tempCornerStickers);
    corners[end].setOrientation(tempCOrientation);
}

/**
 * Flips all edges so that the stickers and saved orientation changes for
 * each edge
 */
public void flipAllEdges() {
    for (int i = 0; i < 4; ++i) {
        edges[i].flip();
    }
}

/**
 * Flips the stickers and orientation of the Edge at the specified index
 *
 * @param index
 *         the index of the Edge to be flipped
```

```
/*
public void flipEdge(int index) {
    this.getEdge(index).flip();
}

/**
 * This method should be called after performing a move so that the corners
 * are twisted correctly.
 *
 * @param direction
 *         the direction of the move that was performed.
 */
public void twistAllCorners(int direction) {
    for (int i = 0; i < 4; ++i) {
        if (i % 2 == 0)
            corners[i].twist(-1);
        else
            corners[i].twist(1);
    }
}

/**
 * Twists (rotates stickers and changes orientation) the Corner at the
 * specified index in the specified direction
 *
 * @param index
 *         the index of the Corner to be twisted
 * @param direction
 *         the direction in which to rotate the Corner
 */
public void twistCorner(int index, int direction) {
    corners[index].twist(direction);
}

/**
 * Swaps the Edges at the specified indices
 *
 * @param i
 *         the index of the first Edge to be swapped.
 * @param j
 *         the index of the second Edge to be swapped.
 */
public void swapEdges(int i, int j) {
    Edge temp = getEdge(i);
```

```
        setEdge(i, getEdge(j));
        setEdge(j, temp);
    }

    /**
     * Swaps the Corners at the specified indices
     *
     * @param i
     *          the index of the first Corner to be swapped.
     * @param j
     *          the index of the second Corner to be swapped.
     */
    public void swapCorners(int i, int j) {
        Corner temp = getCorner(i);
        setCorner(i, getCorner(j));
        setCorner(j, temp);
    }

    /**
     * @param cubie
     *          the cubie to be searched for
     * @return <b>true</b> if the slice contains the specified Cubie; <br>
     *         <b>false</b> otherwise
     */
    public boolean contains(Cubie cubie) {
        for (int i = 0; i < 4; ++i) {
            if (edges[i].compareTo(cubie) == 0)
                return true;
            else if (corners[i].compareTo(cubie) == 0)
                return true;
        }
        return false;
    }
}
```

## Class Solve

```
package jCube;

import java.text.SimpleDateFormat;
import java.util.LinkedList;

/**
 * @author Kelsey McKenna
 */
public class Solve {

    /**
     * This stores the time of the solve.
     */
    private String time = "";
    /**
     * This stores the comment for the solve.
     */
    private String comment = "";
    /**
     * This stores the penalty for the solve.
     */
    private String penalty = "";
    /**
     * This stores the scramble for the solve.
     */
    private String scramble = "";
    /**
     * This stores the solution for the solve.
     */
    private String solution = "";
    /**
     * This stores the move count for the solve.
     */
    private int moveCount = 0;

    /**
     * Constructor - assigns values to time, penalty, and comment fields
     *
     * @param time
     *          the time of the solve
     * @param penalty
     *          the penalty of the solve
    
```

```
* @param comment
*      the comment of the solve
*/
public Solve(String time, String penalty, String comment) {
    this.time = time;
    this.penalty = penalty;
    this.comment = comment;
}

/**
 * Constructor - assigns values to time, penalty, comment, scramble, and
 * solution fields
 *
 * @param time
 *      the time of the solve
 * @param penalty
 *      the penalty of the solve
 * @param comment
 *      the comment of the solve
 * @param scramble
 *      the scramble of the solve
 * @param solution
 *      the solution of the solve
 */
public Solve(String time, String penalty, String comment, String scramble, String solution) {
    this.time = time;
    this.penalty = penalty;
    this.comment = comment;
    this.scramble = scramble;
    this.solution = solution;
    this.moveCount = getMoveCount(solution);
}

/**
 * Constructor - assigns values to time, penalty, comment, scramble, and
 * solution fields
 *
 * @param time
 *      the time of the solve
 * @param penalty
 *      the penalty of the solve
 * @param comment
 *      the comment of the solve
 * @param scramble
```

```
*           the scramble of the solve
 * @param solution
 *           the solution of the solve
 */
public Solve(String time, String penalty, String comment, String scramble, LinkedList<String> solution) {
    int size = solution.size();

    this.time = time;
    this.penalty = penalty;
    this.comment = comment;
    this.scramble = scramble;
    this.moveCount = size;

    for (int i = 0; i < size; ++i)
        this.solution += solution.get(i) + " ";
}

/**
 * Sets the time of the solve.
 *
 * @param time
 *           the time of the solve in string format and formatted
 *           representation, e.g. the input should be 1:02.00 rather than
 *           62.0
 */
public void setStringTime(String time) {
    this.time = time;
}

/**
 * Sets the time of the solve
 *
 * @param time
 *           the time of the solve as a real number
 */
public void setNumericTime(double time) {
    this.time = Double.toString(time);
}

/**
 * Sets the penalty of the solve
 *
 * @param penalty
 *           the penalty of the solve
 */
```

```
/*
public void setPenalty(String penalty) {
    this.penalty = penalty;
}

/**
 * Sets the comment for the solve
 *
 * @param comment
 *         the comment for the solve
 */
public void setComment(String comment) {
    this.comment = comment;
}

/**
 * @param scramble
 *         the scramble for the solve
 */
public void setScramble(String scramble) {
    this.scramble = scramble;
}

/**
 * @param solution
 *         the solution for the solve
 */
public void setSolution(String solution) {
    this.solution = solution;
}

/**
 * @param moveCount
 *         the move count for the solve
 */
public void setMoveCount(int moveCount) {
    this.moveCount = moveCount;
}

/**
 * @return the time in a string representation
 */
public String getStringTime() {
    return time;
}
```

```
}

/**
 * @return a numeric representation of the solve's time
 */
public double getNumericTime() {
    return getFormattedStringToDouble(time);
}

/**
 * @return the penalty for the solve
 */
public String getPenalty() {
    return this.penalty;
}

/**
 * @return the comment for the solve
 */
public String getComment() {
    return comment;
}

/**
 * @return the scramble for the solve
 */
public String getScramble() {
    return scramble;
}

/**
 * @return the solution for the solve
 */
public String getSolution() {
    return solution;
}

/**
 * @return the move count for the solve
 */
public int getMoveCount() {
    return moveCount;
}
```

```
/**  
 * Determines whether the specified time is in a valid. All valid formats  
 * are: MM:SS.sss <br>  
 * MM:SS. <br>  
 * MM:Ssss <br>  
 * MM:S. <br>  
 * M:SS.sss <br>  
 * M:SS. <br>  
 * M:Ssss <br>  
 * M:S. <br>  
 * SS.sss <br>  
 * SS. <br>  
 * Ssss <br>  
 * S. <br>  
 * DNF <br>  
 *  
 * @param time  
 *         the time to be analysed  
 * @return <b>true</b> if the specified time is valid; <br>  
 *         <b>false</b> otherwise  
 */  
public static boolean isValidTime(String time) {  
    if (time == null)  
        return false;  
  
    if (time.equals("DNF"))  
        return true;  
    else if (!time.matches("(\\d{1,2}:)?\\d{1,2}\\.\\d*"))  
        return false;  
    // Times cannot be greater than or equal to one hour  
    else if (getFormattedStringToDouble(time) > 3599.59)  
        return false;  
    else  
        return true;  
}  
  
/**  
 * Pads a time-string with leading and trailing zeros where appropriate. The  
 * argument must be in the form X:M.ss or similar so that it can be padded  
 * correctly.  
 *  
 * @param time  
 *         the time-string to be padded
```

```
* @return the padded time-string
*/
public static String getPaddedTime(String time) {
    // Stores the resulting padded time-string to be returned.
    String paddedTime = "";
    // Stores the index of the ':' character in the 'time'.
    int indexOfColon = 0;
    // Stores the index of the '.' character in the 'time'.
    int indexOfPeriod = 0;
    // Iterates over each character in 'time'.
    int j = 0;

    if ((indexOfColon = time.indexOf(":")) != -1) {
        // Finds the index of the first non-zero character
        while (time.substring(j, j + 1).equals("0"))
            ++j;

        if (j < indexOfColon)
            paddedTime += time.substring(j, indexOfColon + 1);
    }

    indexOfPeriod = time.indexOf(".");

    if (indexOfColon != -1) {
        /*
         * Adds leading zeros as required after the colon
         */
        for (int i = 0; i < 3 - indexOfPeriod + indexOfColon; ++i)
            paddedTime += "0";
    }
}

paddedTime += time.substring(indexOfColon + 1);

/*
 * Adds trailing zeros so that there are two digits after decimal point
 */
int end = 3 - time.length() + indexOfPeriod;
for (int i = 0; i < end; ++i)
    paddedTime += "0";

return paddedTime;
}
```

```
/**  
 * Returns the numerical value represented by the formatted time-string  
 *  
 * @param time  
 *         the formatted time-string from which the numerical time is to  
 *         be calculated  
 * @return the numerical value represented by <b>time</b>  
 */  
public static double getFormattedStringToDouble(String time) {  
    if (time.equalsIgnoreCase("DNF"))  
        return -1;  
  
    // Stores the index of the ':' character in 'time'.  
    int index = 0;  
    // Stores the resulting numerical representation of 'time'.  
    double result = 0;  
  
    /*  
     * If the formatted string contains a minute component, then multiply  
     * this part by 60 to get the number of seconds.  
     */  
    if ((index = time.indexOf(":")) != -1)  
        result += 60 * Integer.valueOf("0" + time.substring(0, index));  
  
    result += Double.parseDouble("0" + time.substring(index + 1));  
  
    return result;  
}  
  
/**  
 * Returns a formatted time-string representing the same number of seconds  
 * as the argument.  
 *  
 * @param seconds  
 *         the number of seconds of the time  
 * @return a formatted time-string representting the same number of seconds  
 *         as the argument.  
 */  
public static String getSecondsToFormattedString(double seconds) {  
    String formattedString = "";  
  
    formattedString = Integer.toString((int) seconds / 60);  
  
    seconds = seconds % 60;
```

```
        formattedString += ":" + String.format("%.02f", seconds);

    return getPaddedTime(formattedString);
}

/**
 * @param moves
 *          the sequence of moves whose length is to be calculated.
 * @return the number of moves represented in <b>moves</b>
 */
public static int getMoveCount(String moves) {
    int length = moves.trim().length();
    int num = (length == 0) ? 0 : 1;

    for (int i = 0; i < length; ++i) {
        if (moves.substring(i, i + 1).equals(" "))
            ++num;
    }

    return num;
}

/**
 * @param dateString
 *          the date to be analysed
 * @return <b>true</b> if the date is valid and in the form yyyy-MM-dd
 *         HH:mm:ss <br>
 *         <b>false</b> otherwise
 */
public static boolean isValidDate(String dateString) {
    try {
        new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse(dateString);
        if (!dateString.matches("\d{4}-\d{1,2}-\d{1,2} \d{1,2}:\d{1,2}:\d{1,2}"))
            return false;
        else
            return isValidDateCheck(dateString.substring(0, dateString.indexOf(" ")))
                && isValidTimeCheck(dateString.substring(dateString.indexOf(" ") + 1));
    } catch (Exception e) {
        return false;
    }
}

/**
 * @param dateString
```

```
*      the date to be analysed
* @return <b>true</b> if the date is in the form yyyy-MM-dd and is valid; <br>
*      <b>false</b> otherwise
*/
private static boolean isValidDateCheck(String dateString) {
    try {
        // new SimpleDateFormat("yyyy-MM-dd").parse(dateString);

        int day, month, year;

        year = Integer.valueOf(dateString.substring(0, dateString.indexOf("-")));
        dateString = dateString.substring(dateString.indexOf("-") + 1);

        month = Integer.valueOf(dateString.substring(0, dateString.indexOf("-")));
        day = Integer.valueOf(dateString.substring(dateString.indexOf("-") + 1));

        if ((month > 12) || (getNumDaysInMonth(month, year) < day))
            return false;
        else
            return true;
    } catch (Exception e) {
        return false;
    }
}

/**
 * @param month
 *      the month whose number of days is to be found
 * @param year
 *      the year of the date
 * @return the number of days in the month
 */
private static int getNumDaysInMonth(int month, int year) {
    switch (month) {
    case 2:
        if ((year % 4 == 0) && ((year % 100 == 0) ? (year % 400 == 0) : true)) {
            return 29;
        } else {
            return 28;
        }
    case 4:
    case 6:
    case 7:
    case 11:
```

```
        return 30;
    default:
        return 31;
    }
}

/**
 * @param timeString
 *         the time to be analysed
 * @return <b>true</b> if the time is valid; <br>
 *         <b>false</b> otherwise
 */
public static boolean isValidTimeCheck(String timeString) {
    String one, two, three;

    one = timeString.substring(0, timeString.indexOf(":"));
    timeString = timeString.substring(timeString.indexOf(":") + 1);

    two = timeString.substring(0, timeString.indexOf(":"));
    timeString = timeString.substring(timeString.indexOf(":") + 1);

    three = timeString;

    return (Integer.valueOf(one) < 24) && (Integer.valueOf(two) < 60) && (Integer.valueOf(three) < 60);
}
```

## Class SolveDatabaseConnection

```
package jCube;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * @author Kelsey McKenna
 */
public class SolveDatabaseConnection {
    /**
     * The try/catch block is invoked if the table does not exist or the query
     * is invalid
     *
     * @param query
     *      the SQLite query to be performed on 'solve' table
     * @return an array of Solves representing the result of the specified query
     * @throws SQLException
     *      if the query is invalid
     * @throws ClassNotFoundException
     *      if SQLite classes are missing
     */
    public static SolveDBType[] executeQuery(String query) throws SQLException, ClassNotFoundException {
        SolveDBType[] solves = null;

        try {
            solves = executeSafeQuery(query);
        } catch (SQLException e) {
            System.err.print(e.getMessage());
            try {
                initTable();
                solves = executeSafeQuery(query);
            } catch (SQLException exc) {
                System.err.print(exc.getMessage());
            }
        }

        throw new SQLException();
    }
}
```

```
    }

    return solves;
}

/**
 * This method will only be called once the table exists
 *
 * @param query
 *          the SQLite query to be performed on 'solve' table
 * @return an array of Solves representing the result of the specified query
 * @throws ClassNotFoundException
 *          if SQLite classes are missing
 * @throws SQLException
 *          if the table does not exist or the query is invalid
 */
private static SolveDBType[] executeSafeQuery(String query) throws ClassNotFoundException, SQLException { // table
    // initialised
    // etc.

    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement;
    ResultSet rs;
    SolveDBType[] solves;
    int numRecords = 0;

    connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
    statement = connection.createStatement();
    statement.setQueryTimeout(30); // set timeout to 30 sec.

    rs = statement.executeQuery(query);

    while (rs.next())
        ++numRecords;

    rs.close();
    rs = statement.executeQuery(query);

    solves = new SolveDBType[numRecords];

    for (int i = 0; i < numRecords; ++i) {
        rs.next();

        solves[i] = new SolveDBType(rs.getInt("solveID"), rs.getString("solveTime"), rs.getString("penalty"),

```

```
        rs.getString("comment"), rs.getString("scramble"), rs.getString("solution"),
        rs.getString("dateAdded"));
    }

    try {
        if (connection != null)
            connection.close();
    } catch (SQLException e) {
        System.err.print(e.getMessage());
    }

    return solves;
}

/**
 * Executes the specified update on the 'solve' table
 *
 * @param update
 *          the update to be performed on the table
 * @throws ClassNotFoundException
 *          if SQLite classes are missing
 * @throws SQLException
 *          if the query is invalid
 */
public static void executeUpdate(String update) throws ClassNotFoundException, SQLException {
    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement = null;

    connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
    statement = connection.createStatement();
    statement.setQueryTimeout(30); // set timeout to 30 sec.

    statement.executeUpdate(update);

    try {
        if (connection != null)
            connection.close();
    } catch (SQLException e2) {
        System.err.println(e2.getMessage());
    }
}

/**
```

```
* Initialises the table in the database. This method will be called if the
* executeQuery(...) method cannot find the table in the database.
*
* @throws ClassNotFoundException
*         if SQLite classes are missing
*/
private static void initTable() throws ClassNotFoundException {
    Class.forName("org.sqlite.JDBC");
    Connection connection = null;
    Statement statement = null;

    try {
        connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
        statement = connection.createStatement();
        statement.setQueryTimeout(30); // set timeout to 30 sec.
        statement.executeUpdate("DROP TABLE IF EXISTS solve");
        statement.executeUpdate("CREATE TABLE solve(" + "solveID INTEGER PRIMARY KEY AUTOINCREMENT,"
            + "solveTime TEXT," + "penalty TEXT," + "comment TEXT," + "scramble TEXT," + "solution TEXT,"
            + "dateAdded TEXT" + ")");
    } catch (SQLException e) {
        try {
            if (connection != null)
                connection.close();
        } catch (SQLException e2) {
            System.err.println(e.getMessage());
        }
    }
}

/*
 * private static void resetIDs() throws ClassNotFoundException {
 * Class.forName("org.sqlite.JDBC"); Connection connection = null; Statement
 * statement = null; ResultSet rs;
 *
 * try { connection =
 * DriverManager.getConnection("jdbc:sqlite:res/cube.db"); statement =
 * connection.createStatement(); statement.setQueryTimeout(30); // set
 * timeout to 30 sec.
 *
 * rs = statement.executeQuery("SELECT * FROM solve;");
 *
 * statement = connection.createStatement();
 * statement.executeUpdate("DROP TABLE IF EXISTS solveCopy");
 *
```

```
* statement.executeUpdate( "CREATE TABLE solveCopy(" +
* "solveID INTEGER PRIMARY KEY AUTOINCREMENT," + "solveTime TEXT NOT NULL,"
* + "penalty TEXT," + "comment TEXT," + "scramble TEXT," + "solution TEXT,"
* + "dateAdded TEXT" + ");" );
*
*
* while (rs.next()) { String solveTime = rs.getString("solveTime"); String
* penalty = rs.getString("penalty"); String comment =
* rs.getString("comment"), scramble = rs.getString("scramble"); String
* solution = rs.getString("solution"), dateAdded =
* rs.getString("dateAdded");
*
* statement.executeUpdate(String.format(
* "INSERT INTO solveCopy(solveTime, penalty, comment, scramble, solution, dateAdded) "
* + "VALUES (\"%s\", \"%s\", \"%s\", \"%s\", \"%s\", \"%s\");", solveTime,
* penalty, comment, scramble, solution, dateAdded ) );
*
* statement.executeUpdate("DROP TABLE solve");
* statement.executeUpdate("ALTER TABLE solveCopy RENAME TO solve");
*
* } catch (SQLException e) { System.err.println(e.getMessage()); try { if
* (connection != null) connection.close(); } catch (SQLException e2) { } }
* }
*/
/***
 * @return the current time in the format yyyy-MM-dd HH:mm:ss
 */
public static String getCurrentTimeInSQLFormat() {
    DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    Date date = new Date();
    return dateFormat.format(date);
}
```

## Class SolveDatabasePopUp

```
package jCube;

import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.Insets;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import java.sql.SQLException;

import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableCellRenderer;
import javax.swing.table.TableColumnModel;

/**
 * @author Kelsey McKenna
 */
```

```
public class SolveDatabasePopUp extends JFrame implements KeyListener {  
    /**  
     * Default serialVersionUID  
     */  
    private static final long serialVersionUID = 1L;  
  
    /**  
     * This stores the initial width of the window.  
     */  
    private static final int WIDTH = 850;  
    /**  
     * This stores the padding of the elements in the window. The greater the  
     * padding, the further towards the centre of the window the elements will  
     * be.  
     */  
    private final int pad = 10;  
    /**  
     * This indicates the vertical spacing between the text boxes etc. in the  
     * window.  
     */  
    private final int fieldYSpacing = 50;  
    /**  
     * This indicates the vertical spacing between the buttons etc. in the  
     * window.  
     */  
    private final int buttonYSpacing = 40;  
  
    /**  
     * This variable keeps track of the current y position of the last element  
     * placed in the window.  
     */  
    private int y = 0;  
    /**  
     * Stores the font to be used in the text fields  
     */  
    private Font fieldFont = new Font("Arial", 0, 25);  
    /**  
     * This variable stores the text that is shown in the error message when the  
     * user enters an invalid time.  
     */  
    private String timeHelpMessage = "You entered the time incorrectly\n\n" + "Valid formats include:\n" + "MM:SS.ss\n"  
        + "MM:S.ss\n" + "M:SS.ss\n" + "M:S.ss\n" + "SS.ss\n" + "S.ss\n" + "DNF\n";  
    /**
```

```
* After choosing to edit a record in the table, the index of this row in
* the table is stored in this variable.
*/
private int selectedRowIndex;

/**
 * The lower bound for the times shown in the table
 */
private String lowerTimeBoundary = "-1";
/**
 * The upper bound for the times shown in the table
 */
private String upperTimeBoundary = "10000000000";

/**
 * If this variable is true, it indicates that a record is being edited,
 * otherwise a record is being added.
 */
private boolean editing = false;

/**
 * tableContainer is placed 'inside' this variable so that is can be
 * positioned and displayed in the window.
 */
private JPanel solveTablePanel;
/**
 * solveTable is placed inside this variable so that when the size of the
 * table exceeds the size of the window, the user can scroll to view the
 * rest of the table.
 */
private JScrollPane tableContainer;
/**
 * This stores the contents of the table displayed in the window.
 */
private final JTable solveTable;
/**
 * This variable can be customised so that certain cells of the table are
 * uneditable and the columns of the table can be given text. This variable
 * is then set as the model of solveTable.
 */
private final DefaultTableModel model;
/**
 * The buttons are placed in this panel.
 */
```

```
private JPanel buttonPanel;

/**
 * Clicking this button opens the Solve Form window.
 */
private JButton addSolveButton;
/**
 * Clicking this button opens the Solve Form window displaying the existing
 * data for the selected solve.
 */
private JButton editSolveButton;
/**
 * Clicking this button deletes the selected rows from the table.
 */
private JButton deleteSolveButton;
/**
 * Clicking this button loads the selected solves into the main window.
 */
private JButton loadIntoProgramButton;

/**
 * This indicates the attribute by which the records in the table should be
 * sorted.
 */
private String orderByAttribute = "solveTime";
/**
 * This indicates whether the records should be sorted in ascending or
 * descending order. This should be "ASC" or "DESC"
 */
private String orderDirection = "ASC";

/**
 * This label is shown in the Solve Form window with the text 'Time'.
 */
private JLabel timeLabel;
/**
 * This label is shown in the Solve Form window with the text 'Penalty'.
 */
private JLabel penaltyLabel;
/**
 * This label is shown in the Solve Form window with the text 'Comment'.
 */
private JLabel commentLabel;
```

```
* This label is shown in the Solve Form window with the text 'Scramble'.
*/
private JLabel scrambleLabel;
/**
 * This label is shown in the Solve Form window with the text 'Solution'.
*/
private JLabel solutionLabel;
/**
 * This label is shown in the Solve Form window with the text 'Date Added'.
*/
private JLabel dateAddedLabel;

/**
 * This field is shown in the Solve Form and the user can enter the time
 * into this field.
*/
private JTextField timeField;
/**
 * This field is shown in the Solve Form and the user can enter the penalty
 * into this field.
*/
private JTextField penaltyField;
/**
 * This field is shown in the Solve Form and the user can enter the date
 * added into this field.
*/
private JTextField dateAddedField;
/**
 * This is shown in the Solve Form and the user can enter the comment into
 * this field.
*/
private JTextArea commentField;
/**
 * This is shown in the Solve Form and the user can enter the scramble into
 * this field.
*/
private JTextArea scrambleField;
/**
 * This is shown in the Solve Form and the user can enter the solution into
 * this field.
*/
private JTextArea solutionField;

/**
```

```
* commentField is placed 'inside' this variable so that when the length of
* the comment exceeds the length of the commentField, the user can scroll
* to view the rest of the comment.
*/
private JScrollPane commentScrollPane;
/**
 * scrambleField is placed 'inside' this variable so that when the length of
 * the comment exceeds the length of the scrambleField, the user can scroll
 * to view the rest of the scramble.
*/
private JScrollPane scrambleScrollPane;
/**
 * solutionField is placed 'inside' this variable so that when the length of
 * the comment exceeds the length of the solutionField, the user can scroll
 * to view the rest of the solution.
*/
private JScrollPane solutionScrollPane;

/**
 * This represents Solve Form window.
*/
private JFrame solveInputForm;

/**
 * Clicking this button submits the data in the Solve Form window for
 * validation.
*/
private JButton submitButton;

/**
 * This stores the contents of the Solve Form.
*/
private JPanel contentPane;

/**
 * This stores the contents of the menu bar in the Solve Table window.
*/
privateMenuBar menuBar;

/**
 * Constructor - Sets up the Solve Table window.
*/
public SolveDatabasePopUp() {
    super("Solve Table");
```

```
setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
setLayout(new BorderLayout());
setPreferredSize(new Dimension(WIDTH, 600));
setVisible(false);
addWindowListener(new WindowListener() {

    @Override
    public void windowOpened(WindowEvent arg0) {
    }

    @Override
    public void windowIconified(WindowEvent arg0) {
    }

    @Override
    public void windowDeiconified(WindowEvent arg0) {
    }

    @Override
    public void windowDeactivated(WindowEvent arg0) {
    }

    @Override
    public void windowClosing(WindowEvent arg0) {
    }

    @Override
    public void windowClosed(WindowEvent arg0) {
        solveInputForm.setVisible(false);
    }

    @Override
    public void windowActivated(WindowEvent arg0) {
    }
});
menuBar = newMenuBar();
setJMenuBar(menuBar.createMenuBar());
setUpSolveInputForm();

solveTablePanel = new JPanel();
solveTablePanel.setOpaque(true);
```

```
final String[] columnNames = { "ID", "Time", "Penalty", "Comment", "Scramble", "Solution", "Date Added" };

model = new DefaultTableModel() {
    private static final long serialVersionUID = 1L;

    public int getColumnCount() {
        return columnNames.length;
    }

    public String getColumnName(int col) {
        return columnNames[col];
    }

    public boolean isCellEditable(int row, int col) {
        return false;
    }
};

model.addTableModelListener(new TableModelListener() {
    @Override
    public void tableChanged(TableModelEvent e) {
        /*
         * resizeColumnWidths(solveTable); int row = e.getFirstRow();
         *
         * TableModel model = (TableModel)e.getSource(); try {
         * SolveDatabaseConnection.executeUpdate( String.format(
         * "UPDATE solve " + "SET solveTime = \"%s\", " +
         * "penalty = \"%s\", " + "comment = \"%s\", " +
         * "scramble = \"%s\", " + "solution = \"%s\", " +
         * "dateAdded = \"%s\" " + "WHERE solveID = %d", " +
         * model.getValueAt(row, 1), "" + model.getValueAt(row, 2), "" +
         * model.getValueAt(row, 3), "" + model.getValueAt(row, 4), "" +
         * model.getValueAt(row, 5), "" + model.getValueAt(row, 6),
         * Integer.valueOf("" + model.getValueAt(row, 0))) ); } catch
         * (Exception exc) { }
        */
    }
});

solveTable = new JTable();
solveTable.setModel(model);
solveTable.setColumnSelectionAllowed(false);
solveTable.setPreferredScrollableViewportSize(new Dimension(WIDTH, 70));
solveTable.setFillsViewportHeight(true);
solveTable.setAutoscrolls(true);
```

```
solveTable.getTableHeader().setReorderingAllowed(false);
solveTable.setFont(new Font("Arial", 0, 15));
solveTable.setRowHeight(20);
solveTable.addMouseListener(new MouseListener() {
    @Override
    public void mouseReleased(MouseEvent arg0) {
    }

    @Override
    public void mousePressed(MouseEvent arg0) {
    }

    @Override
    public void mouseExited(MouseEvent arg0) {
    }

    @Override
    public void mouseEntered(MouseEvent arg0) {
    }

    @Override
    public void mouseClicked(MouseEvent arg0) {
        if (arg0.getClickCount() == 2) {
            editSolveButtonFunction();
        }
    }
});;

buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(1, 3));
buttonPanel.setPreferredSize(new Dimension(WIDTH, 40));
buttonPanel.setSize(500, 50);

addSolveButton = new JButton("Add Solve");
addSolveButton.setFocusable(false);
addSolveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        timeField.setText("");
        penaltyField.setText("");
        commentField.setText("");
        scrambleField.setText("");
        solutionField.setText("");
        dateAddedField.setText("");
        editing = false;
    }
});;
```

```
solveInputForm.setVisible(true);
menuBar.clearButtonGroupSelection();
menuBar.setEnabledButtonGroup(false);
}
});

editSolveButton = new JButton("Edit Solve");
editSolveButton.setFocusable(false);
editSolveButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        editSolveButtonFunction();
    }
});
};

deleteSolveButton = new JButton("Delete Solve");
deleteSolveButton.setFocusable(false);
deleteSolveButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {

        try {
            Object[] options = { "Yes", "No" };
            int choice = -1;

            if (solveTable.getSelectedRows().length >= 5) {
                choice = JOptionPane.showOptionDialog(null, "Are you sure you want to delete?", "Warning",
                        JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[1]);
            } else
                choice = 0;

            if (choice == 0) {
                setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
                deleteRow();
            }
        } catch (ClassNotFoundException | SQLException e) {
            JOptionPane.showMessageDialog(solveTablePanel, "Unable to delete record from database", "Error",
                    JOptionPane.ERROR_MESSAGE);
        } finally {
        }

        setCursor(Cursor.getDefaultCursor());
    }
});
};
```

```
loadIntoProgramButton = new JButton("Load into Program");
loadIntoProgramButton.setToolTipText("Loads the selected solves into the main window");
loadIntoProgramButton.setFocusable(false);
loadIntoProgramButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        int[] selectedIndices = solveTable.getSelectedRows();
        String test;

        for (int i = 0; i < selectedIndices.length; ++i) {
            test = "" + model.getValueAt(selectedIndices[i], 2);
            Main.addSolveToList(new Solve("" + model.getValueAt(selectedIndices[i], 1),
                test.trim().equals("") ? "0" : test, "" + model.getValueAt(selectedIndices[i], 3), ""
                + model.getValueAt(selectedIndices[i], 4), ""
                + model.getValueAt(selectedIndices[i], 5)));
        }
    }
});

buttonPanel.add(addSolveButton);
buttonPanel.add(editSolveButton);
buttonPanel.add(deleteSolveButton);
buttonPanel.add(loadIntoProgramButton);

tableContainer = new JScrollPane(solveTable);
tableContainer.setViewportView(null);

resizeColumnWidths(solveTable);
populateCellsWithDatabaseData();

add(tableContainer, null);
add(buttonPanel, BorderLayout.SOUTH);
pack();
}

/**
 * Sets up solveInputForm so that the data from the selected row in the
 * table is placed in its fields. It is also indicated that a record is
 * being edited (rather than added).
 */
private void editSolveButtonFunction() {
    int selectedRow = solveTable.getSelectedRow();
```

```
if (selectedRow != -1) {
    timeField.setText("") + solveTable.getValueAt(selectedRow, 1));
    penaltyField.setText("") + solveTable.getValueAt(selectedRow, 2));
    commentField.setText("") + solveTable.getValueAt(selectedRow, 3));
    scrambleField.setText("") + solveTable.getValueAt(selectedRow, 4));
    solutionField.setText("") + solveTable.getValueAt(selectedRow, 5));
    dateAddedField.setText("") + solveTable.getValueAt(selectedRow, 6));
    menuBar.clearButtonGroupSelection();
    menuBar.setEnabledButtonGroup(false);
    solveInputForm.setVisible(true);
    selectedRowIndex = selectedRow;
    editing = true;
}
}

/**
 * Retrieves the data from the 'solve' table and adds it to the table in the
 * window
 */
public void populateCellsWithDatabaseData() {
    int rowCount = model.getRowCount();
    String dateString;

    for (int i = 0; i < rowCount; ++i) { // Remove all rows from table
        model.removeRow(0);
    }

    SolveDBType[] solves = new SolveDBType[0];

    try {
        solves = SolveDatabaseConnection.executeQuery(String.format("SELECT * " + "FROM solve;"));

        switch (orderByAttribute) {
        case "solveTime":
            Sorter.sortByTime(solves, 0, solves.length - 1);
            break;
        case "dateAdded":
            try {
                Sorter.sortByDateAdded(solves, 0, solves.length - 1);
            } catch (Exception e) {
                System.out.println(e.getMessage() + "\n");
            }
            break;
        case "ID":
```

```
try {
    Sorter.sortBySolveID(solves, 0, solves.length - 1);
} catch (Exception e) {
    System.out.println(e.getMessage() + "\n");
}
}

if (orderDirection.equals("DESC"))
    Sorter.reverseArray(solves);

} catch (SQLException e) {
    // JOptionPane.showMessageDialog(null, "Could not load solves",
    // "Error", JOptionPane.ERROR_MESSAGE);
    System.err.print(e.getMessage());
} catch (ClassNotFoundException e) {
    // JOptionPane.showMessageDialog(null, "Could not load solves",
    // "Error", JOptionPane.ERROR_MESSAGE);
    System.err.print(e.getMessage());
}

if (solves != null) {
    for (int i = 0; i < solves.length; ++i) {
        if ((solves[i].getNumericTime() >= Double.valueOf(lowerTimeBoundary))
            && (solves[i].getNumericTime() <= Double.valueOf(upperTimeBoundary))) {
            dateString = solves[i].getDateAdded();
            model.addRow(new Object[] { "" + solves[i].getID(), solves[i].getStringTime(),
                solves[i].getPenalty(), solves[i].getComment(), solves[i].getScramble(),
                solves[i].getSolution(), (dateString.equals("null") ? "" : dateString) });
        }
    }
}

resizeColumnWidths(solveTable);
}

/**
 * Retrieves the data matching the specified query from the 'solve' table
 * and adds it to the table in the window
 *
 * @param query
 *          the query to be used on the database
 */
public void populateCellsWithDatabaseData(String query) {
    int rowCount = model.getRowCount();
```

```
String dateString;

for (int i = 0; i < rowCount; ++i) { // Remove all rows from table
    model.removeRow(0);
    // solveTable.removeAll();
}

SolveDBType[] solves = new SolveDBType[0];

try {
    solves = SolveDatabaseConnection.executeQuery(query);
} catch (SQLException e) {
    // JOptionPane.showMessageDialog(null, "Could not load solves",
    // // "Error", JOptionPane.ERROR_MESSAGE);
    System.err.print(e.getMessage());
} catch (ClassNotFoundException e) {
    // JOptionPane.showMessageDialog(null, "Could not load solves",
    // // "Error", JOptionPane.ERROR_MESSAGE);
    System.err.print(e.getMessage());
}

if (solves != null) {
    for (int i = 0; i < solves.length; ++i) {
        dateString = solves[i].getDateAdded();
        model.addRow(new Object[] { "" + solves[i].getID(), solves[i].getStringTime(), solves[i].getPenalty(),
            solves[i].getComment(), solves[i].getScramble(), solves[i].getSolution(),
            (dateString.equals("null") ? "" : dateString) });
    }
}

resizeColumnWidths(solveTable);
}

/**
 * Adds a row to the table and to the database
 *
 * @throws ClassNotFoundException
 *         if SQLite classes are missing
 * @throws SQLException
 *         if the table does not exist etc.
 */
private void addRow() throws ClassNotFoundException, SQLException {
    SolveDatabaseConnection
        .executeUpdate("INSERT INTO solve(solveTime, penalty, comment, scramble, solution, dateAdded) "
```

```
+ "VALUES (" + temp[0].getID() + ", "", "", "", "", "")";  
  
SolveDBType[] temp = SolveDatabaseConnection.executeQuery("SELECT * FROM solve ORDER BY solveID DESC LIMIT 1");  
  
model.addRow(new Object[] { "" + temp[0].getID(), "", "", "", "", "" });  
  
resizeColumnWidths(solveTable);  
solveTable.setRowSelectionInterval(solveTable.getRowCount() - 1, solveTable.getRowCount() - 1);  
}  
  
/**  
 * Deletes the selected rows from the table, and deletes the corresponding  
 * records from the database  
 *  
 * @throws ClassNotFoundException  
 *          if SQLite classes are missing  
 * @throws SQLException  
 *          if the table does not exist or the updates fail  
 */  
private void deleteRow() throws ClassNotFoundException, SQLException {  
    int[] selectedIndices = solveTable.getSelectedRows();  
  
    if (selectedIndices.length == 0)  
        return;  
  
    solveInputForm.setVisible(false);  
    menuBar.setEnabledButtonGroup(true);  
  
    for (int i = 0; i < selectedIndices.length; ++i) {  
        SolveDatabaseConnection.executeUpdate("DELETE FROM solve WHERE solveID = "  
            + ("'" + model.getValueAt(selectedIndices[i], 0)) + "');");
    }  
  
    for (int i = selectedIndices.length - 1; i >= 0; --i)
        model.removeRow(selectedIndices[i]);  
  
    // SolveDatabaseConnection.resetIDs();  
  
    try {
        solveTable.setRowSelectionInterval(selectedIndices[selectedIndices.length - 1],
            selectedIndices[selectedIndices.length - 1]);
    } catch (Exception e) {
        try {
            solveTable.setRowSelectionInterval((int) Math.max(solveTable.getRowCount() - 1, 0),
```

```
        (int) Math.max(solveTable.getRowCount() - 1, 0));
    } catch (Exception exc) {
    }
}

/**
 * Resizes the widths of the columns in the specified table so that there is
 * maximum visibility in each column
 *
 * @param table
 *         the table whose columns need resized
 */
private void resizeColumnWidths(JTable table) {
    final TableColumnModel columnModel = table.getColumnModel();
    for (int column = 0; column < table.getColumnCount(); column++) {
        int width = 50; // Min width

        for (int row = 0; row < table.getRowCount(); row++) {
            TableCellRenderer renderer = table.getCellRenderer(row, column);
            Component comp = table.prepareRenderer(renderer, row, column);
            width = Math.max(comp.getPreferredSize().width, width);
            width = Math.min(width, comp.getWidth() / 7);
        }

        columnModel.getColumn(column).setPreferredWidth(width);
    }
}

/**
 * This method is invoked if the user is trying to filter the data in the
 * table, but enters invalid data. The data will be reset and an error
 * message will be shown.
 *
 * @param originalLowerBoundary
 *         the value that will be assigned to the
 *         {@link #lowerTimeBoundary}
 * @param originalUpperBoundary
 *         the value that will be assigned to the
 *         {@link #upperTimeBoundary}
 */
private void resetTimeBoundariesAndShowErrorMessage(String originalLowerBoundary, String originalUpperBoundary) {
    lowerTimeBoundary = originalLowerBoundary;
    upperTimeBoundary = originalUpperBoundary;
```

```
JOptionPane.showMessageDialog(this, "The data you entered was invalid", "Error", JOptionPane.ERROR_MESSAGE);  
}  
  
/**  
 * @author Kelsey McKenna  
 */  
private class MenuBar {  
    /**  
     * This variable holds the contents of the menu bar  
     */  
    private JMenuBar menuBar;  
    /**  
     * This stores the contents of the sorting menu  
     */  
    private JMenu sortingMenu;  
    /**  
     * This stores the contents of the filter menu  
     */  
    private JMenu filterMenu;  
  
    /**  
     * The radio buttons are assigned to this button group so that only one  
     * of the radio buttons can selected at one time.  
     */  
    private ButtonGroup sortingButtonGroup;  
    /**  
     * Selecting this radio button causes the rows in the table to be sorted  
     * by the time column in ascending order.  
     */  
    private JRadioButtonMenuItem sortByTimeAscendingItem;  
    /**  
     * Selecting this radio button causes the rows in the table to be sorted  
     * by the time column in descending order.  
     */  
    private JRadioButtonMenuItem sortByTimeDescendingItem;  
    /**  
     * Selecting this radio button causes the rows in the table to be sorted  
     * by the date added column in ascending order.  
     */  
    private JRadioButtonMenuItem sortByDateAscendingItem;  
    /**  
     * Selecting this radio button causes the rows in the table to be sorted  
     * by the date added column in descending order.  
     */
```

```
/*
private JRadioButtonMenuItem sortByDateDescendingItem;
/**
 * Selecting this radio button causes the rows in the table to be sorted
 * by the ID column in ascending order.
 */
private JRadioButtonMenuItem sortByIDAscendingItem;
/**
 * Selecting this radio button causes the rows in the table to be sorted
 * by the ID column in descending order.
 */
private JRadioButtonMenuItem sortByIDDescendingItem;

/**
 * Clicking this menu item allows the user to filter the data in the
 * table by certain criteria relating to the time column.
 */
private JMenuItem filterByTimeItem;
/**
 * Clicking this menu item removes any filter present so that all data
 * from the table is shown
 */
private JMenuItem removeFilterItem;

/**
 * @return the menu bar for the Solve Form window
 */
public JMenuBar createMenuBar() {
    menuBar = new JMenuBar();

    //***** SORTING *****/
    sortingMenu = new JMenu("Sorting");
    sortingButtonGroup = new ButtonGroup();

    sortByTimeAscendingItem = new JRadioButtonMenuItem("Sort by Time (Ascending)");
    sortByTimeAscendingItem.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent arg0) {
            orderByAttribute = "solveTime";
            orderDirection = "ASC";
            populateCellsWithDatabaseData();
        }
    });
}
```

```
sortByTimeDescendingItem = new JRadioButtonMenuItem("Sort by Time (Descending)");
sortByTimeDescendingItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        orderByAttribute = "solveTime";
        orderDirection = "DESC";
        populateCellsWithDatabaseData();
    }
});

sortByDateAscendingItem = new JRadioButtonMenuItem("Sort by Date (Ascending)");
sortByDateAscendingItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        orderByAttribute = "dateAdded";
        orderDirection = "ASC";
        populateCellsWithDatabaseData();
    }
});

sortByDateDescendingItem = new JRadioButtonMenuItem("Sort by Date (Descending)");
sortByDateDescendingItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        orderByAttribute = "dateAdded";
        orderDirection = "DESC";
        populateCellsWithDatabaseData();
    }
});

sortByIDAscendingItem = new JRadioButtonMenuItem("Sort by ID (Ascending)");
sortByIDAscendingItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        orderByAttribute = "ID";
        orderDirection = "ASC";
        populateCellsWithDatabaseData();
    }
});

sortByIDDescendingItem = new JRadioButtonMenuItem("Sort by ID (Descending)");
sortByIDDescendingItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
```

```
        orderByAttribute = "ID";
        orderDirection = "DESC";
        populateCellsWithDatabaseData();
    }
});

sortingButtonGroup.add(sortByTimeAscendingItem);
sortingButtonGroup.add(sortByTimeDescendingItem);
sortingButtonGroup.add(sortByDateAscendingItem);
sortingButtonGroup.add(sortByDateDescendingItem);
sortingButtonGroup.add(sortByIDAscendingItem);
sortingButtonGroup.add(sortByIDDescendingItem);

sortingMenu.add(sortByTimeAscendingItem);
sortingMenu.add(sortByTimeDescendingItem);
sortingMenu.add(sortByDateAscendingItem);
sortingMenu.add(sortByDateDescendingItem);
sortingMenu.add(sortByIDAscendingItem);
sortingMenu.add(sortByIDDescendingItem);
/********** END SORTING *****/

/********** FILTER *****/
filterMenu = new JMenu("Filter");

filterByTimeItem = new JMenuItem("Filter by Time");
filterByTimeItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        String originalLowerBoundary = lowerTimeBoundary;
        String originalUpperBoundary = upperTimeBoundary;
        lowerTimeBoundary = JOptionPane.showInputDialog(null, "Enter lower time boundary",
            "Lower time boundary");
        if (lowerTimeBoundary != null) {
            if (!Solve.isValidTime(lowerTimeBoundary)) {
                resetTimeBoundariesAndShowErrorMessage(originalLowerBoundary, originalUpperBoundary);
                return;
            } else
                lowerTimeBoundary = "" + Solve.getFormattedStringToDouble(lowerTimeBoundary);
        } else
            lowerTimeBoundary = originalLowerBoundary;
        populateCellsWithDatabaseData();
        return;
    }
})
```

```
upperTimeBoundary = JOptionPane.showInputDialog(null, "Enter upper time boundary",
                                              "Upper time boundary");
if (upperTimeBoundary != null) {
    if (!Solve.isValidTime(upperTimeBoundary)) {
        resetTimeBoundariesAndShowErrorMessage(originalLowerBoundary, originalUpperBoundary);
        return;
    } else
        upperTimeBoundary = "" + Solve.getFormattedStringToDouble(upperTimeBoundary);
} else {
    upperTimeBoundary = originalUpperBoundary;
    populateCellsWithDatabaseData();
    return;
}

if (Solve.getFormattedStringToDouble(lowerTimeBoundary) > Solve
    .getFormattedStringToDouble(upperTimeBoundary))
    resetTimeBoundariesAndShowErrorMessage(originalLowerBoundary, originalUpperBoundary);

    populateCellsWithDatabaseData();
}
});

removeFilterItem = new JMenuItem("Remove Filter");
removeFilterItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        lowerTimeBoundary = "-1";
        upperTimeBoundary = "100000000000";
        populateCellsWithDatabaseData();
    }
});

filterMenu.add(filterByTimeItem);
filterMenu.add(removeFilterItem);
/******** END FILTER *****/

menuBar.add(sortingMenu);
menuBar.add(filterMenu);

return menuBar;
}

/**
 * Clear any selection in the sorting button group.
*/
```

```
/*
public void clearButtonGroupSelection() {
    sortingButtonGroup.clearSelection();
}

/**
 * Enables or disables all of the menu items in the menu bar.
 *
 * @param state
 *         <b>true</b> indicates that all should be enabled; <br>
 *         <b>false</b> indicates that all should be 'greyed' out.
 */
public void setEnabledButtonGroup(boolean state) {
    sortByTimeAscendingItem.setEnabled(state);
    sortByTimeDescendingItem.setEnabled(state);
    sortByDateAscendingItem.setEnabled(state);
    sortByDateDescendingItem.setEnabled(state);
    sortByIDAscendingItem.setEnabled(state);
    sortByIDDescendingItem.setEnabled(state);
    filterByTimeItem.setEnabled(state);
    removeFilterItem.setEnabled(state);
}
}

/**
 * Sets up the Solve Form
 */
private void setUpSolveInputForm() {
    solveInputForm = new JFrame("Solve Form");

    contentPane = new JPanel();
    contentPane.setLayout(null);

    solveInputForm.setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
    solveInputForm.setContentPane(contentPane);
    solveInputForm.setPreferredSize(new Dimension(480, 440));
    solveInputForm.setSize(new Dimension(480, 440));
    solveInputForm.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    solveInputForm.setResizable(false);
    solveInputForm.setLocation(600, 150);
    solveInputForm.setVisible(false);
    solveInputForm.addWindowListener(new WindowListener() {
        @Override
        public void windowActivated(WindowEvent arg0) {
```

```
}

@Override
public void windowClosed(WindowEvent arg0) {
    menuBar.setEnabledButtonGroup(true);
}

@Override
public void windowClosing(WindowEvent arg0) {

}

@Override
public void windowDeactivated(WindowEvent arg0) {

}

@Override
public void windowDeiconified(WindowEvent arg0) {

}

@Override
public void windowIconified(WindowEvent arg0) {

}

@Override
public void windowOpened(WindowEvent arg0) {

}

});

// *****Labels*****
timeLabel = new JLabel("Time ");
timeLabel.setSize(120, 40);
timeLabel.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

penaltyLabel = new JLabel("Penalty ");
penaltyLabel.setSize(120, 40);
penaltyLabel.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

commentLabel = new JLabel("Comment ");
commentLabel.setSize(120, 40);
```

```
commentLabel.setLocation(0 + pad, y + pad);  
  
y += fieldYSpacing;  
  
scrambleLabel = new JLabel("Scramble ");  
scrambleLabel.setSize(120, 40);  
scrambleLabel.setLocation(0 + pad, y + pad);  
  
y += fieldYSpacing;  
  
solutionLabel = new JLabel("Solution ");  
solutionLabel.setSize(120, 40);  
solutionLabel.setLocation(0 + pad, y + pad);  
  
y += fieldYSpacing + 50;  
  
dateAddedLabel = new JLabel("Date Added");  
dateAddedLabel.setSize(120, 40);  
dateAddedLabel.setLocation(0 + pad, y + pad);  
  
// *****Text Fields*****  
  
y = 0;  
  
timeField = new JTextField();  
timeField.setMargin(new Insets(0, 10, 0, 15));  
timeField.setFont(fieldFont);  
timeField.setSize(350, 40);  
timeField.setLocation(100 + pad, y + pad);  
timeField.addKeyListener(this);  
  
y += fieldYSpacing;  
  
penaltyField = new JTextField("");  
penaltyField.setMargin(new Insets(0, 10, 0, 15));  
penaltyField.setFont(new Font("Arial", 0, 15));  
penaltyField.setSize(350, 40);  
penaltyField.setLocation(100 + pad, y + pad);  
penaltyField.addKeyListener(this);  
  
y += fieldYSpacing;  
  
commentField = new JTextArea("");  
commentField.setMargin(new Insets(0, 10, 0, 15));
```

```
commentField.setLineWrap(true);
commentField.setWrapStyleWord(true);
commentField.setEditable(true);
commentField.setFont(new Font("Arial", 0, 15));
commentField.addKeyListener(this);

commentScrollPane = new JScrollPane(commentField);
commentScrollPane.setLocation(100 + pad, y + pad);
commentScrollPane.setSize(new Dimension(350, 40));
commentScrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);

y += fieldYSpacing;

scrambleField = new JTextArea("");
scrambleField.setMargin(new Insets(0, 10, 0, 15));
scrambleField.setEditable(true);
scrambleField.setLineWrap(true);
scrambleField.setWrapStyleWord(true);
scrambleField.setFont(new Font("Arial", 0, 15));
scrambleField.addKeyListener(this);

scrambleScrollPane = new JScrollPane(scrambleField);
scrambleScrollPane.setLocation(100 + pad, y + pad);
scrambleScrollPane.setSize(new Dimension(350, 40));
scrambleScrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);

y += fieldYSpacing;

solutionField = new JTextArea();
solutionField.setMargin(new Insets(0, 10, 0, 15));
solutionField.setEditable(true);
solutionField.setLineWrap(true);
solutionField.setWrapStyleWord(true);
solutionField.setFont(new Font("Arial", 0, 15));
solutionField.addKeyListener(this);

solutionScrollPane = new JScrollPane(solutionField);
solutionScrollPane.setLocation(100 + pad, y + pad);
solutionScrollPane.setSize(new Dimension(350, 80));
solutionScrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);

y += fieldYSpacing + 50;

dateAddedField = new JTextField("");
```

```
dateAddedField.setMargin(new Insets(0, 10, 0, 15));
dateAddedField.setFont(new Font("Arial", 0, 15));
dateAddedField.setLocation(100 + pad, y + pad);
dateAddedField.setSize(350, 40);
dateAddedField.setToolTipText("yyyy-mm-dd hh:mm:ss");
dateAddedField.addKeyListener(this);

// *****Submission Button*****
y = 370;

submitButton = new JButton("Submit");
submitButton.setSize(480, 30);
submitButton.setLocation(0, y + pad);
submitButton.setFocusable(false);
submitButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        String time = timeField.getText().trim();

        if (!Solve.isValidTime(time)) {
            JOptionPane.showMessageDialog(solveTablePanel, timeHelpMessage, "Error", JOptionPane.ERROR_MESSAGE,
                null);
            return;
        } else if ((dateAddedField.getText().trim().equals(""))
            || (!Solve.isValidDate(dateAddedField.getText().trim()))) {
            int choice = -1;
            Object[] options = { "Yes", "No" };
            choice = JOptionPane
                .showOptionDialog(
                    null,
                    "Date must be valid and have the form: yyyy-MM-dd HH:mm:ss\nWould you like to use the current
time?",

                    "Error", JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options,
                    options[0]);

            if (choice != 0)
                return;
            else {
                dateAddedField.setText(SolveDatabaseConnection.getCurrentTimeInSQLFormat());
            }
        }
    }

    try {
        int row;
```

```
if (!editing) {
    addRow();
    row = solveTable.getRowCount() - 1;
} else {
    row = selectedRowIndex;
}

if (time.trim().equalsIgnoreCase("DNF")) {
    solveTable.setValueAt("DNF", row, 1);
    solveTable.setValueAt("0", row, 2);
} else {
    solveTable.setValueAt(Solve.getPaddedTime(Solve.getSecondsToFormattedString(Solve
        .getFormattedString.ToDouble(time))), row, 1);
    solveTable.setValueAt(penaltyField.getText().trim(), row, 2);
}

solveTable.setValueAt(commentField.getText().trim(), row, 3);
solveTable.setValueAt(scrambleField.getText().trim(), row, 4);
solveTable.setValueAt(solutionField.getText().trim(), row, 5);
solveTable.setValueAt(dateAddedField.getText().trim(), row, 6);
solveInputForm.setVisible(false);
menuBar.setEnabledButtonGroup(true);
editing = false;

menuBar.clearButtonGroupSelection();

SolveDatabaseConnection.executeUpdate(String.format("UPDATE solve " + "SET solveTime = \"%s\", "
    + "penalty = \"%s\", " + "comment = \"%s\", " + "scramble = \"%s\", "
    + "solution = \"%s\", " + "dateAdded = \"%s\" " + "WHERE solveID = %d",
    "" + model.getValueAt(row, 1), "" + model.getValueAt(row, 2),
    "" + model.getValueAt(row, 3), "" + model.getValueAt(row, 4),
    "" + model.getValueAt(row, 5), "" + model.getValueAt(row, 6),
    Integer.valueOf("" + model.getValueAt(row, 0))));
resizeColumnWidths(solveTable);
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
}
}
});

y += buttonYSpacing;
```

```
contentPane.add(timeLabel);
contentPane.add(penaltyLabel);
contentPane.add(commentLabel);
contentPane.add(scrambleLabel);
contentPane.add(solutionLabel);
contentPane.add(dateAddedLabel);

contentPane.add(timeField);
contentPane.add(penaltyField);
contentPane.add(commentScrollPane);
contentPane.add(scrambleScrollPane);
contentPane.add(solutionScrollPane);
contentPane.add(dateAddedField);

contentPane.add(submitButton);

}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.KeyListener#keyPressed(java.awt.event.KeyEvent)
 */
@Override
public void keyPressed(KeyEvent arg0) {
    Main.transferFormFocus(arg0);
}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.KeyListener#keyReleased(java.awt.event.KeyEvent)
 */
@Override
public void keyReleased(KeyEvent arg0) {

}

/**
 * (non-Javadoc)
 *
 * @see java.awt.event.KeyListener#keyTyped(java.awt.event.KeyEvent)
 */
@Override
public void keyTyped(KeyEvent arg0) {
```

```
    if (arg0.getKeyChar() == KeyEvent.VK_ENTER) {  
        submitButton.doClick();  
    }  
}
```

## Class SolveDBType

```
package jCube;

/**
 * @author Kelsey McKenna
 */
public class SolveDBType extends Solve {

    /**
     * The ID of the solve
     */
    private int id;
    /**
     * The date the solve was added
     */
    private String dateAdded = "";

    /**
     * Constructor - assigns values to fields
     *
     * @param id
     *          the ID for the solve
     * @param time
     *          the time for the solve
     * @param penalty
     *          the penalty for the solve
     * @param comment
     *          the comment for the solve
     * @param scramble
     *          the scramble for the solve
     * @param solution
     *          the solution for the solve
     * @param dateAdded
     *          the solve was added
     */
    public SolveDBType(int id, String time, String penalty, String comment, String scramble, String solution,
                      String dateAdded) {
        super(time, penalty, comment, scramble, solution);
        this.id = id;
        this.dateAdded = dateAdded;
    }

    /**

```

```
* @return ID of the solve
*/
public int getID() {
    return id;
}

/**
 * @return the date the solve was added
 */
public String getDateAdded() {
    return dateAdded;
}

}
```

## Class SolveMaster

```
package jCube;

import java.awt.Color;
import java.util.LinkedList;

/**
 * @author Kelsey McKenna
 */
public class SolveMaster {

    /**
     * This is used for readability in the code.
     */
    public static final int CANCELLATIONS = 0;
    /**
     * This is used for readability in the code.
     */
    public static final int CORNER_EDGE = 1;
    /**
     * This is used for readability in the code.
     */
    public static final int CROSS = 2;

    /**
     * This represents the cube displayed on screen. It can be used to generate
     * a solution for the current state of the cube.
     */
    protected Cube cube;
    /**
     * This array indicates which slices are adjacent to each other such that
     * they share edges.
     */
    protected int[][] sliceEdgeSharing = { { 0, 5 }, { 0, 2 }, { 0, 4 }, { 0, 3 }, { 5, 3 }, { 5, 2 }, { 4, 2 },
        { 4, 3 }, { 1, 5 }, { 1, 3 }, { 1, 4 }, { 1, 2 } };
    /**
     * This array indicates which slices are adjacent to each other such that
     * they share corners.
     */
    private int[][] sliceCornerSharing = { { 0, 3, 5 }, { 0, 5, 2 }, { 0, 2, 4 }, { 0, 4, 3 }, { 1, 2, 5 },
        { 1, 5, 3 }, { 1, 3, 4 }, { 1, 4, 2 } };
    /**
     * This accumulates the moves required to solve the cube.
```

```
/*
private LinkedList<String> solveMoves;
/**
 * This accumulates the explanation of how to solve the cube.
 */
protected String solutionExplanation = "";

/**
 * Constructor - assigns a value to the cube field, i.e. the cube to be
 * solved.
 *
 * @param cube
 *         the cube for which a solution will be generated.
 */
public SolveMaster(Cube cube) {
    this.cube = cube;
    solveMoves = new LinkedList<>();
}

/**
 * @return the explanation of the solution (prose)
 */
public String getSolutionExplanation() {
    return solutionExplanation;
}

/**
 * Returns the index of the specified cubie on the cube
 *
 * @param cubie
 *         the cubie to be found
 * @return the index of the cubie on the cube
 */
public int getIndexOf(Cubie cubie) {
    /*
     * Slice currentSlice; Cubie currentCubie;
     *
     * for (int slice = 0; slice < 6; ++slice) { currentSlice =
     * cube.getSlice(slice); for (int piece = 0; piece < 4; ++piece) {
     * currentCubie = currentSlice.getCorner(piece);
     *
     * if (currentCubie.compareTo(cubie) == 0) return
     * currentCubie.getCubieIndex();
     */
}
```

```
* currentCubie = currentSlice.getEdge(piece);
*
* if (currentCubie.compareTo(cubie) == 0) return
* currentCubie.getCubieIndex(); } }
*/
for (int i = 0; i < 12; ++i) {
    try {
        if ((cube.getEdge(i).compareTo(cubie) == 0) || ((cube.getCorner(i).compareTo(cubie) == 0)))
            return i;
    } catch (ArrayIndexOutOfBoundsException e) {
    }
}
return -1;
}

/**
 * Performs the rotations required so that the cube shows the specified
 * centre colour on top. The rotations performed are recorded.
 *
 * @param color
 *          the colour which should end up on top
 */
public void rotateToTop(Color color) {
    if (getSliceIndexOfCentre(color) == 0)
        return;

    for (int i = 0; i < 4; ++i) {
        cube.rotate("x");
        catalogMove("x");
        if (getSliceIndexOfCentre(color) == 0)
            return;

        cube.rotate("z");
        catalogMove("z");
        if (getSliceIndexOfCentre(color) == 0)
            return;
    }
}

/**
 * Performs the rotations required so that the cube shows the specified
 * centre colour on top. The rotations performed are <i>not</i> recorded.
 *
```

```
* @param color
*      the colour which should end up on top
*/
public void rotateToTopDoNotRecord(Color color) {
    if (getSliceIndexOfCentre(color) == 0)
        return;

    for (int i = 0; i < 4; ++i) {
        cube.rotate("x");
        if (getSliceIndexOfCentre(color) == 0)
            return;

        cube.rotate("z");
        if (getSliceIndexOfCentre(color) == 0)
            return;
    }
}

/**
 * Performs the rotations required so that the specified centre colour end
 * up on top and front. This will get stuck in an infinite loop if the
 * centre are opposite, e.g. white and yellow. These parameters cannot be
 * given during runtime, so if this gets stuck, there is a problem in code.
 * The rotations are recorded.
 *
 * @param top
 *      the centre to end up on top.
 * @param front
 *      the centre to end up on front.
 */
public void rotateToTopFront(Color top, Color front) {
    rotateToTop(top);

    while (getSliceIndexOfCentre(front) != 4) {
        cube.performAbsoluteMoves("y");
        catalogMoves("y");
    }
}

/**
 * Performs the rotations required so that the specified centre colour end
 * up on top and front. This will get stuck in an infinite loop if the
 * centre are opposite, e.g. white and yellow. These parameters cannot be
 * given during runtime, so if this gets stuck, there is a problem in code.
```

```
* The rotations are <i>not</i> recorded.
*
* @param top
*        the centre to end up on top.
* @param front
*        the centre to end up on front.
*/
public void rotateToTopFrontDoNotRecord(Color top, Color front) {
    rotateToTopDoNotRecord(top);

    while (getSliceIndexOfCentre(front) != 4) {
        cube.performAbsoluteMoves("y");
    }
}

/**
 * Simplifies the specified moves in the specified way.
 *
 * @param originalMoves
 *        the moves to be simplified
 * @param type
 *        the type of simplification to be applied
 */
public static void simplifyMoves(LinkedList<String> originalMoves, int type) {

    switch (type) {
    case CANCELLATIONS:
        cancelMoves(originalMoves);
        break;
    case CORNER_EDGE:
        cancelMoves(originalMoves);
        simplifyCornerEdgeSolution(originalMoves);
        break;
    case CROSS:
        simplifyCross(originalMoves);
        cancelMoves(originalMoves);
        break;
    }
}

/**
 * Simplifies the specified moves so that cancellations occur, <br>
 * e.g. R R R = R' <br>
```

```
* and so that split rotations are grouped, <br>
* e.g. y U y U = y2 U2
*
* @param originalMoves
*        the moves to be simplified
*/
private static void simplifyCornerEdgeSolution(LinkedList<String> originalMoves) {
    String temp;

    for (int i = originalMoves.size() - 3; i >= 0; --i) {
        if ((originalMoves.get(i).substring(0, 1).equals(originalMoves.get(i + 2).substring(0, 1)))
            && ("yU".contains(originalMoves.get(i).substring(0, 1))) && ("yU".contains(originalMoves
                .get(i + 1).substring(0, 1)))) {
            temp = originalMoves.get(i);
            originalMoves.set(i, originalMoves.get(i + 1));
            originalMoves.set(i + 1, temp);
            cancelMoves(originalMoves);
        }
    }
}

/**
 * Simplifies the cross solution so that there are no rotations, <br>
 * e.g. R2 U y2 R = R2 U L
 *
 * @param originalMoves
 *        the moves to be simplified
*/
private static void simplifyCross(LinkedList<String> originalMoves) {
    // Stores the current move being examined.
    String current;

    /*
     * Starts at the penultimate element since last element will be a z
     * rotation.
     */
    for (int i = originalMoves.size() - 2; i >= 0; --i) {
        current = originalMoves.get(i);

        /*
         * If y rotation is found then, alter all elements after the
         * rotation so that the rotation is not required.
         */
    }
}
```

```
if (current.substring(0, 1).equals("y") && (current.length() <= 2)) {
    for (int j = i + 1; j < originalMoves.size(); ++j) {
        originalMoves.set(j, applyRotationToMove(current, originalMoves.get(j)));
    }
    originalMoves.remove(i);
}
}

/**
 * Returns the move that performs the same action on the cube if the cube
 * was not rotated before the move.
 *
 * @param rotation
 *         the rotation that would have been made.
 * @param move
 *         the move that would have been made after the rotation
 * @return the move that performs the same action as the specified move
 *         <i>without</i> a rotation
 */
private static String applyRotationToString(String rotation, String move) {
    // Stores the moves so that
    String[] movePairings = { "D", "U", "R", "L", "B", "F" };
    // Stores the offset used in later calculations for the index of the
    // element to return.
    int offset = (rotation.contains("") ? 1 : 0);
    // Stores the index of the element which is equal to the first character
    // of 'move'
    int index;

    if (!rotation.contains("2")) {
        switch (rotation.substring(0, 1)) {
            case "x":
                switch (move.substring(0, 1)) {
                    case "U":
                        return movePairings[5 - offset] + move.substring(1);
                    case "D":
                        return movePairings[4 + offset] + move.substring(1);
                    case "F":
                        return movePairings[offset] + move.substring(1);
                    case "B":
                        return movePairings[1 - offset] + move.substring(1);
                    default:
```

```
        return move;
    }
    case "y":
        switch (move.substring(0, 1)) {
            case "R":
                return movePairings[4 + offset] + move.substring(1);
            case "L":
                return movePairings[5 - offset] + move.substring(1);
            case "F":
                return movePairings[2 + offset] + move.substring(1);
            case "B":
                return movePairings[3 - offset] + move.substring(1);
            default:
                return move;
        }
    case "z":
        switch (move.substring(0, 1)) {
            case "U":
                return movePairings[3 - offset] + move.substring(1);
            case "D":
                return movePairings[2 + offset] + move.substring(1);
            case "L":
                return movePairings[offset] + move.substring(1);
            case "R":
                return movePairings[1 - offset] + move.substring(1);
            default:
                return move;
        }
    }
} else {
    switch (rotation.substring(0, 1)) {
        case "x":
            if ("LR".contains(move.substring(0, 1)))
                return move;
        case "y":
            if ("UD".contains(move.substring(0, 1)))
                return move;
        case "z":
            if ("FB".contains(move.substring(0, 1)))
                return move;
    }
}

index = LinearSearch.linearSearch(movePairings, move.substring(0, 1));
return movePairings[index + ((index % 2 == 0) ? 1 : -1)];
```

```
    }

    return "-1";
}

/**
 * Cancels moves such as L2 L' -> L.
 *
 * @param originalMoves
 *         the moves to simplify
 */
private static void cancelMoves(LinkedList<String> originalMoves) {
    // Stores the index of the current move being examined.
    int index = 0;
    // Stores the resulting combination the two moves being examined.
    String combination;
    // Stores the first element being examined.
    String one;
    // Stores the second element being examined.
    String two;
    // Stores the plane of the move, e.g. R2 has plane = R
    String moveType;

    while (index < (originalMoves.size() - 1)) {
        one = originalMoves.get(index).trim();
        two = originalMoves.get(index + 1).trim();

        // If the moves act on the same slice then
        if (one.substring(0, 1).equals(two.substring(0, 1))) {
            moveType = one.substring(0, 1);
            one = one.substring(1);
            two = two.substring(1);

            combination = getCombination(one, two);

            if (combination.equals("NOT_MATCHING"))
                ++index;
            else {
                if (combination.equals("-1")) {
                    /*
                     * The moves cancel, so remove each of them.
                     */
                    originalMoves.remove(index);
                    originalMoves.remove(index);
                }
            }
        }
    }
}
```

```
        } else {
            /*
             * The two moves can be simplified to one, so remove the
             * second move and alter the first one.
             */
            originalMoves.remove(index + 1);
            originalMoves.set(index, moveType + combination);
        }

        /*
         * The moves have been simplified, so index needs to go back
         * to check if further simplifications will occur.
         */
        index = (index == 0) ? 0 : index - 1;
    }
} else {
    ++index;
}
}

/*
 * This simplifies "x y x" to "z2 y"
 */
for (int i = 0; i < originalMoves.size() - 2; ++i) {
    if ((originalMoves.get(i).equals("x")) && (originalMoves.get(i + 1).equals("z"))
        && (originalMoves.get(i + 2).equals("x"))) {
        originalMoves.remove(i);
        originalMoves.set(i, "z2");
        originalMoves.set(i + 1, "y");
    }
}

/**
 * @param one
 *          the first direction
 * @param two
 *          the second direction
 * @return the resultant direction, e.g. ("2", "") would return ""
 */
private static String getCombination(String one, String two) {

    if (((one.startsWith("w")) || (two.startsWith("w")))) {
        try {
```

```
/*
 * if they are not both wide moves, then return NOT_MATCHING
 */
if (!one.substring(0, 1).equals(two.substring(0, 1)))
    return "NOT_MATCHING";
else {
    one = one.substring(1);
    two = two.substring(1);
}
} catch (IndexOutOfBoundsException e) {
    // This will happen when one
    // of them is of the form
    // Xw* and the other is only
    // X
    return "NOT_MATCHING";
}

if ((one.equals("")) && (two.equals ""))
    return "2";
else if (((one.equals("")) && (two.equals("')))) || ((one.equals("')) && (two.equals("")))))
    return "-1"; // cancels
else if (((one.equals("")) && (two.equals("2")))) || ((one.equals("2")) && (two.equals(""))))
    return "";
else if ((one.equals("')) && (two.equals("''))))
    return "2";
else if (((one.equals("')) && (two.equals("2")))) || ((one.equals("2")) && (two.equals("')))))
    return "";
else
    /* if ((one.equals("2")) && (two.equals("2")))*/
    return "-1"; // cancels
}

/**
 * Determines whether or not the specified corner is solved. This method
 * uses the actual argument in the further method calls.
 *
 * @param corner
 *         the corner to be analysed
 * @return <b>true</b> if the specified corner is solved; <br>
 *         <b>false</b> otherwise
 */
protected boolean isPieceSolved(Corner corner) {
```

```
        return (isPieceIsInCorrectPosition(cubie) && (cube.getCorner(indexOf(cubie)).getOrientation() == 0));
    }

    /**
     * Determines whether or not the specified cubie is solved. This method uses
     * a retrieved cubie in the further method calls rather than the original
     * argument. This is useful if the argument does not actually belong to the
     * same instance of Cube as the field in this class.
     *
     * @param cubie
     *         the cubie to be analysed
     * @return <b>true</b> if the specified cubie is solved; <br>
     *         <b>false</b> otherwise
     */
    protected boolean newPieceSolved(Cubie cubie) {
        int index = indexOf(cubie);

        if (cubie.getStickers().length == 2)
            return isPieceSolved(cube.getEdge(index));
        else
            return isPieceSolved(cube.getCorner(index));

        /*
         * int orientation = cube.getEdge(index).getOrientation();
         *
         * if ((cube.getSlice(sliceEdgeSharing[index][orientation]).getCentre()
         * .equals(edge.getStickers()[0])) && (cube.getSlice(
         * sliceEdgeSharing[index][(orientation + 1) % 2])
         * .getCentre().equals(edge.getStickers()[1]))) return true;
         *
         * return false;
         */
    }

    /**
     * Determines whether or not the specified edge is solved. This method uses
     * the actual argument in the further method calls.
     *
     * @param edge
     *         the edge to be analysed
     * @return <b>true</b> if the specified edge is solved; <br>
     *         <b>false</b> otherwise
     */
    protected boolean isPieceSolved(Edge edge) {
```

```
int index = getIndexOf(edge);
Edge tempEdge = cube.getEdge(index);

if ((cube.getSlice(sliceEdgeSharing[index][0]).getCentre().equals(tempEdge.getStickers()[0]))
    && (cube.getSlice(sliceEdgeSharing[index][1]).getCentre().equals(tempEdge.getStickers()[1])))
    return true;

return false;
}

/*
 * protected boolean pieceSolved(Cubie cubie) { if
 * (cubie.getStickers().length == 2) return (pieceIsInCorrectPosition(cubie)
 * && (cube.getEdge(getIndexOf(cubie)).getOrientation() == 0)); else return
 * (pieceIsInCorrectPosition(cubie) &&
 * (cube.getCorner(getIndexOf(cubie)).getOrientation() == 0)); }
 */

/**
 * Returns the index of the specified centre colour on the cube. <br>
 * 0 = Top <br>
 * 1 = Bottom <br>
 * 2 = Right <br>
 * 3 = Left <br>
 * 4 = Front <br>
 * 5 = Back <br>
 *
 * @param centre
 *          the centre colour to be found
 * @return the index of the slice whose centre is the specified colour
 */
private int getSliceIndexOfCentre(Color centre) {
    for (int i = 0; i < 6; ++i)
        if (cube.getSlice(i).getCentre().equals(centre))
            return i;

    return -1;
}

/**
 * Records the specified move
 *
 * @param move
 *          the move to be recorded
```

```
/*
private void catalogMove(String move) {
    move = move.trim();

    if (!move.equals(""))
        solveMoves.add(move);
}

/*
 * public void catalogMoves(String moves) { moves = moves.trim();
 *
 * if (moves.equals("")) return;
 *
 * int i = 0; int indexOfSpace; String remainingMoves;
 *
 * while (i < moves.length()) { remainingMoves = moves.substring(i);
 * indexOfSpace = remainingMoves.indexOf(" ");
 *
 * if (indexOfSpace == -1) indexOfSpace = remainingMoves.length();
 *
 * catalogMove(remainingMoves.substring(0, indexOfSpace)); i += indexOfSpace
 * + 1; } }
 */

/**
 * Checks that the argument is not null, then records the moves.
 *
 * @param moves
 *          the moves to be recorded
 */
public void catalogMoves(String moves) {
    if ((moves == null) || (moves.trim().equals("")))
        return;

    catMoves(moves.trim(), 0);
}

/**
 * Recursively records the specified moves, one by one. <br>
 * E.g. "R U R' F" will be recorded as "R", "U", "R'", "F"
 *
 * @param moves
 *          the moves to be recorded
 * @param index
 */
```

```
*           the last index of a space
*/
private void catMoves(String moves, int index) {
    // This indicates that all moves have been recorded.
    if (index >= moves.length())
        return;

    // Stores the remaining moves to be recorded.
    String remainingMoves = moves.substring(index).trim();
    // Stores the index of the first space in the remaining moves to be
    // recorded so that the next move can be identified.
    int indexOfSpace = remainingMoves.indexOf(" ");

    if (indexOfSpace == -1)
        indexOfSpace = remainingMoves.length();

    catalogMove(remainingMoves.substring(0, indexOfSpace));
    catMoves(moves, index + indexOfSpace + 1);
}

/*
 * public String[] getCatalogMoves() { String[] moves = new
 * String[solveMoves.size()];
 *
 * for (int i = 0; i < solveMoves.size(); ++i){ moves[i] =
 * solveMoves.get(i); }
 *
 * return moves; }
 */

/**
 * @return the recorded moves in a linked list of strings
 */
public LinkedList<String> getCatalogMoves() {
    return this.solveMoves;
}

/**
 * @param moves
 *          the moves to be analysed
 * @return the number of moves contained within <b>moves</b>
 */
public int getNumMoves(String[] moves) {
    int numMoves = 0;
```

```
boolean u = false;

for (int i = 0; i < moves.length; ++i) {
    if (!("xyz").contains(moves[i].substring(0, 1))) {
        if (moves[i].substring(0, 1).equals("U")) {
            if (!u) {
                ++numMoves;
                u = true;
            }
        } else {
            ++numMoves;
            u = false;
        }
    }
}

return numMoves;
}

/**
 * Clears all moves and the associated explanation
 */
public void clearMoves() {
    solveMoves.clear();
    solutionExplanation = "";
}

/**
 * @param cubie
 *          the cubie to be analysed
 * @return <b>true</b> if the specified cubie is in the correct position on
 *          the cube (regardless of orientation); <br>
 *          <b>false</b> otherwise
 */
public boolean isPieceIsInCorrectPosition(Cubie cubie) {
    // Stores the index of the cubie on the cube.
    int index = getIndexOf(cubie);
    // Stores the colours of stickers on the cubie so that the expected
    // centres around the cubie can be compared with the actual centres.
    Color[] centres = new Color[cubie.getStickers().length];
    // Stores true if the current centre sticker being examined is found.
    boolean found;
    // Stores the indices of the slices that immediately surround the cubie.
```

```
int[] slicesIndex = (centres.length == 2) ? sliceEdgeSharing[index] : sliceCornerSharing[index];

for (int i = 0; i < centres.length; ++i)
    centres[i] = cube.getSlice(slicesIndex[i]).getCentre();

/*
 * This compares each expected centre colour with each colour of the
 * cubie. If all colours are the shared, then the piece is in the
 * correct position.
 */
for (int i = 0; i < centres.length; ++i) {
    found = false;
    for (int j = 0; j < centres.length; ++j) {
        if (centres[i].equals(cubie.getStickers()[j])) {
            found = true;
            break;
        }
    }
    if (!found)
        return false;
}

return true;
}

/**
 * @param cubie
 *          the cubie to be analysed
 * @return the destination of the cubie on a solved cube according the
 *         current permutation of the centres.
 */
protected int getIndexOfDestination(Cubie cubie) {
    // Stores the number of stickers on the cubie, indicating whether it is
    // a corner or edge.
    int numStickers = cubie.getStickers().length;
    // Stores the indices of the slices that surround each cubie at each
    // location.
    int[][] slicesIndices = (numStickers == 2) ? sliceEdgeSharing : sliceCornerSharing;
    // Stores true if the current centre has been found on the cubie.
    boolean foundCentre;
    // Stores true if the current slicesIndices element represents the
    // destination of the cubie.
    boolean foundPosition;
```

```
/*
 * Goes through each position on the cube to see if the centre
 * surrounding that position match the stickers of the specified cubie.
 */
for (int i = 0; i < slicesIndices.length; ++i) {
    foundPosition = true;
    for (int j = 0; j < slicesIndices[i].length; ++j) {
        foundCentre = false;
        for (int k = 0; k < numStickers; ++k) {
            if (cube.getSlice(slicesIndices[i][j]).getCentre().equals(cubie.getStickers()[k])) {
                foundCentre = true;
                break;
            }
        }
        if (!foundCentre) {
            foundPosition = false;
            break;
        }
    }
    if (foundPosition)
        return i;
}

return -1;
}

/**
 * @param start
 *          the starting position
 * @param end
 *          the ending position
 * @return the shortest offset between the start and end, e.g. 3 -> -1
 */
@SuppressWarnings("unused")
private int getShortestOffset(int start, int end) {
    return Math.abs(((4 * (int)) ((Math.abs(start - end) % 4) / 2)) + (Math.abs(start - end) % 4));
}

/**
 * @return the recorded moves as a string with a space between each move
 */
public String getStringMoves() {
```

```
        return getStringMoves(this.solveMoves);
    }

    /**
     * @param moves
     *          the moves to be returned as a string
     * @return the specified moves as a string with a space between each move
     */
    public static String getStringMoves(LinkedList<String> moves) {
        String stringMoves = "";
        int size = moves.size();

        for (int i = 0; i < size; ++i) {
            stringMoves += (moves.get(i) + " ");
        }

        return stringMoves;
    }

    /**
     * Returns the number of trailing U moves in the recorded moves and removes
     * them simultaneously.
     *
     * @return the number of trailing U moves in the recorded moves
     */
    public int getNumTrailingU() {
        int i = solveMoves.size() - 1;
        int numTrailing = 0;

        while ((i >= 0) && (solveMoves.get(i).equals("U"))) {
            solveMoves.remove(i);
            ++numTrailing;
            --i;
        }

        return numTrailing;
    }

    /**
     * @param originalMoves
     *          the moves to be reversed
     * @return the original moves in reverse order as a string with a space
     *         between each move
     */
}
```

```
public static String getReverseStringMoves(LinkedList<String> originalMoves) {  
    String reverseMoves = "";  
    String current;  
  
    for (int i = originalMoves.size() - 1; i >= 0; --i) {  
        current = originalMoves.get(i);  
        if (current.substring(1).equals(""))  
            reverseMoves += current.substring(0, 1) + " " + " ";  
        else if (current.substring(1).equals("')))  
            reverseMoves += current.substring(0, 1) + " " + " ";  
        else  
            reverseMoves += current + " ";  
    }  
  
    return reverseMoves;  
}  
  
/**  
 * @param moves  
 *          the moves to be reversed  
 * @return the moves in reverse order as a string with a space between each  
 *         move  
 */  
public static String getReverseStringMoves(String moves) {  
    if (moves == null)  
        return "";  
  
    moves = moves.trim();  
    if (moves.equals(""))  
        return "";  
  
    int i = 0;  
    int indexOfSpace;  
    String remainingMoves;  
    LinkedList<String> reverseString = new LinkedList<>();  
  
    while (i < moves.length()) {  
        remainingMoves = moves.substring(i);  
        indexOfSpace = remainingMoves.indexOf(" ");  
  
        if (indexOfSpace == -1)  
            indexOfSpace = remainingMoves.length();  
  
        reverseString.add(remainingMoves.substring(0, indexOfSpace));  
    }  
}
```

```
i += indexOfSpace + 1;
}

return getReverseStringMoves(reverseString);
}

/**
 * @param key
 *          the key that was pressed on the keyboard
 * @return the move associated with that key press
 */
public static String getKeyToMove(String key) {
    key = key.toLowerCase();

    switch (key) {
        case "j":
            return "U";
        case "s":
            return "D";
        case "i":
            return "R";
        case "d":
            return "L";
        case "h":
            return "F";
        case "w":
            return "B";
        case "f":
            return "U'";
        case "l":
            return "D'";
        case "k":
            return "R'";
        case "e":
            return "L'";
        case "g":
            return "F'";
        case "o":
            return "B'";
        case "u":
            return "Rw";
        case "m":
            return "Rw'";
        case "r":
    }
}
```

```
        return "Lw'";
    case "v":
        return "Lw";
    case "x":
        return "M";
    case ".":
        return "M'";
    case ";":
        return "y";
    case "a":
        return "y'";
    case "y":
        return "x";
    case "n":
        return "x'";
    case "p":
        return "z";
    case "q":
        return "z'";
    default:
        return "";
    }
}

/**
 * @param move
 *          the move to be checked
 * @return <b>true</b> if the move is valid; <br>
 *         <b>false</b> otherwise
 */
public static boolean isValidMove(String move) {
    String[] validMoves = { "U", "D", "R", "L", "F", "B", "U'", "D'", "R'", "L'", "F'", "B'", "Rw", "Lw", "Rw'",
                           "Lw'", "M", "M'", "Y", "Y'", "X", "X'", "Z", "Z'", "U2", "D2", "R2", "L2", "F2", "B2", "Rw2", "Lw2",
                           "M2", "Y2", "X2", "Z2" };

    if ((LinearSearch.linearSearch(validMoves, move) == -1))
        return false;

    return true;
}

/**
 * This returns a string representation of the current state of the cube so
```

```
* that it can be saved/loaded to/from file. The first two colours are top
* and front centres, and other colours are each sticker of each corner then
* each edge.
*
* @return the current state of the cube represented as a string
*/
public String getStateString() {
    // Accumulates the resulting state-string
    String result = "";
    // Stores the stickers of the current cubie being examined.
    Color[] cStickers;

    /*
     * Stores the top and front centre colours.
     */
    result = Cubie.getColorToWord(cube.getSlice(0).getCentre()) + ","
        + Cubie.getColorToWord(cube.getSlice(4).getCentre()) + ",";

    /*
     * Stores the corner stickers.
     */
    for (int i = 0; i < 8; ++i) {
        cStickers = cube.getCorner(i).getStickers();

        for (int j = 0; j < 3; ++j)
            result += Cubie.getColorToWord(cStickers[j]) + ",";
    }

    /*
     * Stores the edge stickers.
     */
    for (int i = 0; i < 12; ++i) {
        cStickers = cube.getEdge(i).getStickers();

        for (int j = 0; j < 2; ++j)
            result += Cubie.getColorToWord(cStickers[j]) + ",";
    }

    return result;
}

/**
 * Extracts the state stored in <b>stateString</b> and applies the
 * corresponding data to the cube so that the cube is in the state specified
```

```
* in <b>stateString</b>
*
* @param stateString
*         the state to be given to the cube
*/
public void applyStateString(String stateString) {
    int stringIndex = 0, indexOfComma = 0;
    Color[] colors = new Color[50];

    for (int i = 0; i < 50; ++i) {
        indexOfComma = stateString.substring(stringIndex).indexOf(",");
        colors[i] = Cubie.getWordToColor(stateString.substring(stringIndex, stringIndex + indexOfComma));
        stringIndex += indexOfComma + 1;
    }

    rotateToTopFrontDoNotRecord(colors[0], colors[1]);

    for (int i = 0; i < 8; ++i) {
        if (!cube.getCorner(i).getStickers()[0].equals(Color.LIGHT_GRAY))
            cube.getCorner(i).setStickers(colors[i * 3 + 2], colors[i * 3 + 3], colors[i * 3 + 4]);
    }
    for (int i = 0; i < 12; ++i) {
        if (!cube.getEdge(i).getStickers()[0].equals(Color.LIGHT_GRAY))
            cube.getEdge(i).setStickers(colors[i * 2 + 26], colors[i * 2 + 27]);
    }

    Main.assignOrientationsToCubies();
    cube.updateAll();
}

}
```

## Class Sorter

```
package jCube;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * @author Kelsey McKenna
 */
public class Sorter {

    /**
     * Checks that the list is not null or empty then sorts the list with the
     * fastest time first and the slowest time last.
     *
     * @param list
     *         the list to be sorted
     * @param start
     *         the index of the first element to be sorted
     * @param end
     *         the index of the last element to be sorted
     */
    public static void sortByTime(SolveDBType[] list, int start, int end) {
        /*
         * If the list is null, then it cannot be sorted, so return; If the
         * list's length is less than 2, then it is already sorted, so return.
         */
        if ((list == null) || (list.length <= 1))
            return;

        sBT(list, start, end);
    }

    /**
     * Sorts a list with the fastest time first and the slowest time last.
     *
     * @param list
     *         the list to be sorted
     * @param start
     *         the index of the first element to be sorted
     * @param end
     *         the index of the last element to be sorted
     */
    private static void sBT(SolveDBType[] list, int start, int end) {
        if (start >= end)
            return;

        int pivotIndex = partition(list, start, end);
        sBT(list, start, pivotIndex - 1);
        sBT(list, pivotIndex + 1, end);
    }

    private static int partition(SolveDBType[] list, int start, int end) {
        SolveDBType pivot = list[start];
        int i = start + 1;
        int j = end;

        while (true) {
            while (i <= j && list[i].getTime() <= pivot.getTime())
                i++;

            while (j >= i && list[j].getTime() >= pivot.getTime())
                j--;

            if (i > j)
                break;

            swap(list, i, j);
        }

        swap(list, start, j);
        return j;
    }

    private static void swap(SolveDBType[] list, int i, int j) {
        SolveDBType temp = list[i];
        list[i] = list[j];
        list[j] = temp;
    }
}
```

```
/*
public static void sBT(SolveDBType[] list, int start, int end) {
    double pivot = Solve.getFormattedStringToDouble(list[((start + end) / 2)].getStringTime());

    // This points to the element on the left of the pivot
    int i = start;
    // This points to the element on the right of the pivot
    int j = end;

    // Stores the numeric representation of the time currently being
    // examined
    double current = 0;

    if (pivot == -1)
        pivot = 1e10;

    while (i <= j) {
        /*
         * This converts the ith element to its numerical value, e.g.
         * 1:10.50 becomes 70.5. If the element is -1, then set it to
         * infinity (1e10), otherwise take its numerical value. This results
         * in the list being sorted with -1s at the end rather than at the
         * start because -1 represents DNF which represents infinity.
         */
        while (((current = Solve.getFormattedStringToDouble(list[i].getStringTime())) == -1 ? 1e10 : current) < pivot)
            ++i;

        while (((current = Solve.getFormattedStringToDouble(list[j].getStringTime())) == -1 ? 1e10 : current) > pivot)
            --j;

        if (i <= j) {
            swap(list, i, j);
            ++i;
            --j;
        }
    }

    if (i < end)
        sBT(list, i, end);

    if (start < j)
        sBT(list, start, j);
}
```

```
/**  
 * Sorts the list with the earliest date first and the latest date last.  
 *  
 * @param list  
 *        the list to be sorted  
 * @param start  
 *        the index of the first element to be sorted  
 * @param end  
 *        the index of the last element to be sorted  
 * @throws ParseException  
 *        if one of the dates cannot be parsed, i.e. the format is  
 *        wrong  
 */  
public static void sortByDateAdded(SolveDBType[] list, int start, int end) throws ParseException {  
    Date pivot = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse(list[((start + end) / 2)].getDateAdded());  
    int i = start, j = end;  
  
    while (i <= j) {  
        while (new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse(list[i].getDateAdded()).before(pivot))  
            ++i;  
  
        while (new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse(list[j].getDateAdded()).after(pivot))  
            --j;  
  
        if (i <= j) {  
            swap(list, i, j);  
            ++i;  
            --j;  
        }  
    }  
  
    if (i < end)  
        sortByDateAdded(list, i, end);  
  
    if (start < j)  
        sortByDateAdded(list, start, j);  
}  
  
/**  
 * @param list  
 *        the list to be sorted  
 * @param start  
 *        the index of the first element to be sorted  
 * @param end
```

```
*           the index of the last element to be sorted
* @throws Exception
*           if one of the elements has not been instantiated etc.
*/
public static void sortBySolveID(SolveDBType[] list, int start, int end) throws Exception {
    int pivot = list[((start + end) / 2)].getID();
    int i = start, j = end;

    while (i <= j) {
        while (list[i].getID() < pivot)
            ++i;

        while (list[j].getID() > pivot)
            --j;

        if (i <= j) {
            swap(list, i, j);
            ++i;
            --j;
        }
    }

    if (i < end)
        sortBySolveID(list, i, end);

    if (start < j)
        sortBySolveID(list, start, j);
}

/**
 * @param list
 *           the list in which elements will be swapped
 * @param i
 *           the index of the first element to be swapped
 * @param j
 *           the index of the second element to be swapped
 */
private static void swap(SolveDBType[] list, int i, int j) {
    SolveDBType temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}

/**
```

```
* Reverses the order of the specified array, <br>
* e.g. {1, 7, 4} would be come {4, 7, 1}
*
* @param list
*        the array to be reversed
*/
public static void reverseArray(SolvedDBType[] list) {
    for (int i = 0; i < list.length / 2; ++i)
        swap(list, i, list.length - 1 - i);
}

/**
 * Checks that the list is not null or empty then sorts the list with the
 * best average first and the worst average last.
*
* @param list
*        the list to be sorted
* @param start
*        the index of the first element to be sorted
* @param end
*        the index of the last element to be sorted
*/
public static void sortByAverageThenTime(MemberCompetition[] list, int start, int end) {
    if ((list == null) || (list.length <= 1))
        return;

    sBA(list, start, end);
}

/**
 * Sorts the list with the best average first and the worst average last.
*
* @param list
*        the list to be sorted
* @param start
*        the index of the first element to be sorted
* @param end
*        the index of the last element to be sorted
*/
public static void sBA(MemberCompetition[] list, int start, int end) {
    MemberCompetition pivot = list[((start + end) / 2)];
    int i = start, j = end;

    while (i <= j) {
```

```
        while (list[i].isBetterThan(pivot))
            ++i;

        while (pivot.isBetterThan(list[j]))
            --j;

        if (i <= j) {
            swap(list, i, j);
            ++i;
            --j;
        }
    }

    if (i < end)
        sBA(list, i, end);

    if (start < j)
        sBA(list, start, j);
}

/**
 * @param list
 *          the list in which the elements will be swapped
 * @param i
 *          the index of the first element to be swapped
 * @param j
 *          the index of the second element to be swapped
 */
private static void swap(MemberCompetition[] list, int i, int j) {
    MemberCompetition temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}

/**
 * Sorts the specified list into ascending order. The argument is checked to
 * see if it is null or has fewer than 2 elements.
 *
 * @param list
 *          the list to sort
 */
public static void quickSort(double[] list) {
    /*
     * If the list is null, then it cannot be sorted, so return; If the

```

```
* list's length is less than 2, then it is already sorted, so return.  
*/  
if ((list == null) || (list.length <= 1))  
    return;  
  
    qSort(list, 0, list.length - 1);  
}  
  
/**  
 * Sorts the specified list in to ascending order.  
 *  
 * @param list  
 *        the list to be sorted  
 * @param start  
 *        the index of the first element to be sorted  
 * @param end  
 *        the index of the last element to be sorted  
 */  
private static void qSort(double[] list, int start, int end) {  
    double pivot = list[(start + end) / 2];  
    // This points to the element on the left of the pivot  
    int i = start;  
    // This points to the element on the right of the pivot  
    int j = end;  
  
    while (i <= j) {  
        while (list[i] < pivot)  
            ++i;  
  
        while (list[j] > pivot)  
            --j;  
  
        if (i <= j) {  
            swap(list, i, j);  
            ++i;  
            --j;  
        }  
    }  
  
    if (i < end)  
        qSort(list, i, end);  
  
    if (start < j)  
        qSort(list, start, j);
```

```
}

/**
 * Swaps the elements at the specified indices in the specified list
 *
 * @param list
 *          the list in which elements will be swapped
 * @param i
 *          the index of the first element to be swapped
 * @param j
 *          the index of the last element to be swapped
 */
private static void swap(double[] list, int i, int j) {
    double temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}

}
```

## Class Statistics

```
package jCube;

import java.util.Arrays;
import java.util.LinkedList;

/**
 * @author Kelsey McKenna
 */
public class Statistics {

    /**
     * @author Kelsey McKenna
     */
    private static class Average {
        /**
         * This stores the name of the average, e.g. 'Average of 5'
         */
        String name;
        /**
         * This stores the calculated numerical interpretation of the average.
         */
        double average;
        /**
         * This stores the number of times in the average, e.g. average of 5
         * will have a length of 5.
         */
        int length;
        /**
         * This stores the times of the average in a formatted fashion; the
         * fastest and slowest times are surrounded by brackets.
         */
        String formattedTimes;
    }

    /**
     * This variable shares the same memory location as the list in the main
     * window. This means that statistics of recent times can be found.
     */
    private LinkedList<Solve> times;
    /**
     * Stores the calculated averages.
     */
}
```

```
private static Average[] averages;

/**
 * Constructor - assigns the times from which averages are calculated
 *
 * @param times
 *          the times to be used
 */
public Statistics(LinkedList<Solve> times) {
    this.times = times;
    averages = new Average[8];

    for (int i = 0; i < averages.length; ++i)
        averages[i] = new Average();
}

/**
 * Sets the times to be used for averages
 *
 * @param times
 *          the times to be used for averages
 */
public void setTimes(LinkedList<Solve> times) {
    this.times = times;
}

/**
 * Returns an average of the specified size. The times used are the most
 * recent ones.
 *
 * @param sizeOfAverage
 *          the size of the average
 * @return the average of the specified size
 */
public double getRecentAverageOf(int sizeOfAverage) {
    return getAverageOf(sizeOfAverage, times);
}

/**
 * @return the overall mean of all times, ignoring DNFs
 */
public double getOverallMean() {
    return getOverallMean(times);
}
```

```
/**  
 * @return a string representing all calculated statistics in a formatted  
 *         fashion  
 */  
public String getRecentFormattedStandardStatistics() {  
    return getFormattedStandardStatisticsString(times);  
}  
  
/**  
 * @return a array containing all calculated statistics in a formatted  
 *         fashion  
 */  
public String[] getRecentFormattedStatisticsArray() {  
    return getFormattedStatisticsArray(times);  
}  
  
/**  
 * @param sizeOfAverage  
 *        must be 5 or over  
 * @param times  
 *        the times from which the average is calculated  
 * @return the average of the specified size.  
 */  
public static double getAverageOf(int sizeOfAverage, LinkedList<Solve> times) {  
    // Stores the number of DNF times in the past 'sizeOfAverage' times.  
    int numDNF = getNumDNF(sizeOfAverage, times);  
    int size = times.size();  
  
    if ((times.size() < sizeOfAverage) || (sizeOfAverage < 5) || (numDNF > 1))  
        return -1;  
  
    // Stores the raw numerical representation of the times.  
    double[] raw = new double[sizeOfAverage];  
    // Accumulates the sum of the times to be used in the mean, i.e. the  
    // middle (x - 2) times.  
    double sum = 0;  
    // Stores the number of times to be used in the mean so that the mean  
    // can be found.  
    double factor = /* (x < 5) ? x : */sizeOfAverage - 2;  
    // Stores the index of the first element to be used in the mean.  
    int start;  
    // Stores the index of the last element to be used in the mean.  
    int end;
```

```
start = 1;
end = sizeOfAverage - 1;

/*
 * This allows the number of times specified to be processed.
 */
for (int i = 0; i < sizeOfAverage; ++i)
    raw[i] = times.get(size - sizeOfAverage + i).getNumericTime();

/*
 * This gets all DNFs (-1s) at the start and sorts other times.
 */
Sorter.quickSort(raw);

/*
 * If there is a DNF time, then the first element will represent the
 * slowest time and the second element the fastest, so the last three
 * elements need to be processed.
 */
if (numDNF > 0) {
    ++start;
    ++end;
}

for (int i = start; i < end; ++i) {
    sum += raw[i];
}

return (sum / factor);
}

/**
 * @param sizeOfAverage
 *         the size of the average (must be greater than or equal to 5)
 * @param stringTimes
 *         the times from which the average will be calculated
 * @return the average of the specified times
 */
public static double getAverageOf(int sizeOfAverage, String[] stringTimes) {
    LinkedList<Solve> tempTimes = new LinkedList<>();

    for (int i = 0; i < stringTimes.length; ++i)
        tempTimes.add(new Solve(stringTimes[i], "", ""));
```

```
        return getAverageOf(sizeOfAverage, tempTimes);
    }

    /**
     * @param times
     *          the times from which the overall mean will be calculated
     * @return the overall mean of all specified times, ignoring DNFs
     */
    public static double getOverallMean(LinkedList<Solve> times) {
        int size = times.size();
        int numDNF = getNumDNF(size, times);
        double result = 0, current;

        if ((size == 0) || (numDNF == size))
            return -1;

        for (int i = 0; i < size; ++i) {
            current = times.get(i).getNumericTime();
            if (current != -1)
                result += current;
        }

        return (result) / ((double) size - getNumDNF(size, times));
    }

    /**
     * Returns a formatted average (i.e. all times are separated by commas and
     * the fastest and slowest times are surrounded with brackets) of the
     * specified size as the string
     *
     * @param sizeOfAverage
     *          the size of the average to be returned
     * @param times
     *          the times from which the average is calculated
     * @return a formatted string of the average in the form
     *          "[average] [time_1], [time_2], ..., [time_n]"
     */
    public static String getFormattedAverage(int sizeOfAverage, LinkedList<Solve> times) {
        int size = times.size();
        // Stores the times in order of their numerical value.
        double[] orderedTimes = new double[sizeOfAverage];
        // Stores a copy of the times in the average.
        double[] timesCopy = new double[sizeOfAverage];
```

```
/*
 * The first element indicates whether or not the slowest time has been
 * found and enclosed in brackets; The second element indicates whether
 * or no the fastest time has been found and enclosed in brackets.
 */
boolean[] found = { false, false };
String formattedStatistics = "";

if (size < sizeOfAverage)
    return "";

for (int i = 0; i < sizeOfAverage; ++i)
    orderedTimes[i] = times.get(size - sizeOfAverage + i).getNumericTime();

timesCopy = Arrays.copyOf(orderedTimes, orderedTimes.length);
Sorter.quickSort(orderedTimes);
sortByDNF(orderedTimes);

for (int i = 0; i < timesCopy.length; ++i) {
    /*
     * If the current time is the slowest time the average, and the
     * slowest time has not already been identified, then put brackets
     * around it.
     */
    if ((!found[1]) && (timesCopy[i] == orderedTimes[orderedTimes.length - 1])) {
        found[1] = true;
        formattedStatistics += String.format("(%s)",
            (timesCopy[i] == -1) ? "DNF" : Solve.getSecondsToFormattedString(timesCopy[i]));
    }
    /*
     * If the current time is the fastest time the average, and the
     * fastest time has not already been identified, then put brackets
     * around it.
     */
    else if ((timesCopy[i] == orderedTimes[0]) && (!found[0])) {
        found[0] = true;
        formattedStatistics += String.format("(%s)",
            (timesCopy[i] == -1) ? "DNF" : Solve.getSecondsToFormattedString(timesCopy[i]));
    } else
        formattedStatistics += String.format("%s",
            (timesCopy[i] == -1) ? "DNF" : Solve.getSecondsToFormattedString(timesCopy[i]));

    formattedStatistics += ", ";
}
```

```
/*
 * Remove the last comma
 */
if (formattedStatistics.contains(","))
    formattedStatistics = formattedStatistics.substring(0, formattedStatistics.lastIndexOf(","));

return formattedStatistics;
}

/**
 * Sorts the specified list so that '-1' elements are at the end
 *
 * @param list
 *          the list to be sorted
 */
private static void sortByDNF(double[] list) {
    // Stores the index of the element whose value will become list[j]
    int i = 0;
    // Stores the index of the element being examined.
    int j = 0;

    /*
     * Pushes all non-DNF times to start of array.
     */
    while ((i < list.length) && (j < list.length)) {
        if (list[j] != -1) {
            list[i] = list[j];
            ++i;
        }

        ++j;
    }

    /*
     * All non-DNF times are before i, so set all elements after to -1
     */
    for (; i < list.length; ++i) {
        list[i] = -1;
    }
}

/**
 * @param sizeOfAverage
 *          the size of the average to be calculated
```

```
* @param averageIndex
*           the index of the element in <b>averages</b> that contains the
*           properties for the average.
* @param times
*           the times from which the average will be calculated
* @return the best average of the specified size.
*/
public static double getBestAverageOf(int sizeOfAverage, int averageIndex, LinkedList<Solve> times) {
    if (sizeOfAverage < 1)
        return -1;

    LinkedList<Solve> currentTimes = new LinkedList<>();
    int size = times.size();
    double bestAverage = 1e10; // infinite
    double currentAverage;
    double bestAverageBestTime = 0;

    averages[averageIndex].formattedTimes = "";

    for (int i = 0; i <= size - sizeOfAverage; ++i) {
        currentTimes.clear();
        for (int j = i; j < i + sizeOfAverage; ++j) {
            currentTimes.add(times.get(j));
        }
        if ((currentAverage = getAverageOf(sizeOfAverage, currentTimes)) <= bestAverage) {
            if (currentAverage == bestAverage) {
                if (getFastestTimeInPrevious(sizeOfAverage, currentTimes) < bestAverageBestTime) {
                    bestAverage = currentAverage;
                    bestAverageBestTime = getFastestTimeInPrevious(sizeOfAverage, currentTimes);
                    averages[averageIndex].formattedTimes = getFormattedAverage(sizeOfAverage, currentTimes);
                }
            } else {
                bestAverage = currentAverage;
                bestAverageBestTime = getFastestTimeInPrevious(sizeOfAverage, currentTimes);
                averages[averageIndex].formattedTimes = getFormattedAverage(sizeOfAverage, currentTimes);
            }
        }
    }

    return (bestAverage == 1e10) ? -1 : bestAverage;
}

/**
 * @param times
```

```
*           the times from which the formatted statistics will be
*           generated.
* @return a string containing the all calculated statistics in a formatted
*         fashion.
*/
public static String getFormattedStandardStatisticsString(LinkedList<Solve> times) {
    String formattedStatistics = "";
    double overallMean = getOverallMean(times);
    int size = times.size();

    averages[0].name = "Current Average of 5";
    averages[1].name = "Current Average of 12";
    averages[2].name = "Current Average of 50";
    averages[3].name = "Current Average of 100";
    averages[4].name = "Best Average of 5";
    averages[5].name = "Best Average of 12";
    averages[6].name = "Best Average of 50";
    averages[7].name = "Best Average of 100";

    averages[0].length = 5;
    averages[1].length = 12;
    averages[2].length = 50;
    averages[3].length = 100;
    averages[4].length = 5;
    averages[5].length = 12;
    averages[6].length = 50;
    averages[7].length = 100;

    for (int i = 0; i < averages.length / 2; ++i) {
        averages[i].average = getAverageOf(averages[i].length, times);
        formattedStatistics += String.format("%s: \t", averages[i].name);
        formattedStatistics += (averages[i].average == -1) ? "DNF" : String.format("%s\t%s", Solve
            .getSecondsToFormattedString(averages[i].average),
            (averages[i].length <= 12) ? getFormattedAverage(averages[i].length, times) : "");
        formattedStatistics += String.format("\n");
    }

    formattedStatistics += String.format("%n%nOverall Mean (%d/%d):\t", size - getNumDNF(size, times), size)
        + ((overallMean == -1) ? "DNF" : String.format("%s", Solve.getSecondsToFormattedString(overallMean)));
    formattedStatistics += String.format("%n%n");

    for (int i = averages.length / 2; i < averages.length; ++i) {
        averages[i].average = getBestAverageOf(averages[i].length, i, times);
        formattedStatistics += String.format("%s: \t", averages[i].name);
```

```
        formattedStatistics += (averages[i].average == -1) ? "DNF" : String.format("%s\t%s", Solve
            .getSecondsToFormattedString(averages[i].average),
            (averages[i].length <= 12) ? averages[i].formattedTimes : ""));
        formattedStatistics += String.format("%n");
    }

    return formattedStatistics;
}

/**
 * Returns each the formatted string for each average in an array
 *
 * @param times
 *         the times from which the statistics will be generated
 * @return an array containing strings which represent the statistics in a
 *         formatted fashion
 */
public static String[] getFormattedStatisticsArray(LinkedList<Solve> times) {
    String[] formattedStatistics = new String[(averages.length * 3) + 3];
    double overallMean = getOverallMean(times);
    int size = times.size(), fIndex = 0;

    averages[0].name = "Current Average of 5";
    averages[1].name = "Current Average of 12";
    averages[2].name = "Current Average of 50";
    averages[3].name = "Current Average of 100";
    averages[4].name = "Best Average of 5";
    averages[5].name = "Best Average of 12";
    averages[6].name = "Best Average of 50";
    averages[7].name = "Best Average of 100";

    averages[0].length = 5;
    averages[1].length = 12;
    averages[2].length = 50;
    averages[3].length = 100;
    averages[4].length = 5;
    averages[5].length = 12;
    averages[6].length = 50;
    averages[7].length = 100;

    for (int i = averages.length / 2; i < averages.length; ++i) {
        averages[i].average = getBestAverageOf(averages[i].length, i, times);
        formattedStatistics[fIndex++] = averages[i].name;
        formattedStatistics[fIndex++] = (averages[i].average == -1) ? "DNF" : Solve
```

```
        .getSecondsToFormattedString(averages[i].average);
    formattedStatistics[fIndex++] = averages[i].formattedTimes;
}

for (int i = 0; i < averages.length / 2; ++i) {
    averages[i].average = getAverageOf(averages[i].length, times);
    formattedStatistics[fIndex++] = averages[i].name;
    formattedStatistics[fIndex++] = (averages[i].average == -1) ? "DNF" : Solve
        .getSecondsToFormattedString(averages[i].average);
    formattedStatistics[fIndex++] = getFormattedAverage(averages[i].length, times);
}

formattedStatistics[fIndex++] = String.format("Overall Mean (%d/%d)", size - getNumDNF(size, times), size);
formattedStatistics[fIndex++] = (overallMean == -1) ? "DNF" : Solve.getSecondsToFormattedString(overallMean);

String formattedOverallAverage = "";
double current;

for (int i = 0; i < size; ++i) {
    current = times.get(i).getNumericTime();
    formattedOverallAverage += String.format("%s, ",
        (current == -1) ? "DNF" : Solve.getSecondsToFormattedString(current));
}

if (formattedOverallAverage.contains(","))
    formattedOverallAverage = formattedOverallAverage.substring(0, formattedOverallAverage.lastIndexOf(","));

formattedStatistics[fIndex] = formattedOverallAverage;

return formattedStatistics;
}

/**
 * @param sizeOfAverage
 *         the size of the average currently being calculated
 * @param times
 *         the times from which the number of DNFs will be determined
 * @return the number of 'DNF'/-1 times in the past <b>sizeOfAverage</b>
 *         times
 */
public static int getNumDNF(int sizeOfAverage, LinkedList<Solve> times) {
    int num = 0, size = times.size();
    if ((size == 0) || (size < sizeOfAverage))
        return 0;
```

```
for (int i = 0; i < sizeOfAverage; ++i) {
    if (times.get(size - sizeOfAverage + i).getNumericTime() == -1)
        ++num;
}

return num;
}

/*
 * public double getFastestTimeInPrevious(int x) { return
 * getFastestTimeInPrevious(x, times); } public double
 * getSlowestTimeInPrevious(int x) { return getFastestTimeInPrevious(x,
 * times); }
 */

/**
 * Returns the fastest time in the previous <b>sizeOfAverage</b> times
 *
 * @param sizeOfAverage
 *          the size of the average being calculated
 * @param times
 *          the times from which the fastest time will be determined
 * @return the fastest time in the previous <b>sizeOfAverage</b> times
 */
public static double getFastestTimeInPrevious(int sizeOfAverage, LinkedList<Solve> times) {
    if ((times.size() < sizeOfAverage) || (sizeOfAverage < 1))
        return -1;

    double best = times.getLast().getNumericTime();
    int start = times.size() - 1, end = times.size() - sizeOfAverage;
    double current;

    for (int i = start; i >= end; --i) {
        current = times.get(i).getNumericTime();

        best = Math.min(best, current);
    }

    return best;
}

/**
 * Returns the slowest time in the previous <b>sizeOfAverage</b> times

```

```
* @param sizeOfAverage
*       the size of the average being calculated
* @param times
*       the times from which the slowest time will be determined
* @return the slowest time in the previous <b>sizeOfAverage</b> times
*/
public static double getSlowestTimeInPrevious(int sizeOfAverage, LinkedList<Solve> times) {
    if ((times.size() < sizeOfAverage) || (sizeOfAverage < 1))
        return -1;

    double slowest = times.getLast().getNumericTime();
    int start = times.size() - 1, end = times.size() - sizeOfAverage;
    double current;

    for (int i = start; i >= end; --i) {
        current = times.get(i).getNumericTime();

        slowest = Math.max(slowest, current);
    }

    return slowest;
}
```

## Class TextFile

```
package jCube;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.PrintWriter;

/**
 * @author Kelsey McKenna
 */
public class TextFile {

    /**
     * A constant used for readability and to prevent external argument errors
     * that can occur when string parameters are used.
     */
    public static final int READ = 0;
    /**
     * A constant used for readability and to prevent external argument errors
     * that can occur when string parameters are used.
     */
    public static final int WRITE = 1;
    /**
     * A constant used for readability and to prevent external argument errors
     * that can occur when string parameters are used.
     */
    public static final int WRITE_APPEND = 2;
    /**
     * This stores the file path of the text file.
     */
    private String filePath = null;
    /**
     * This is used with fr0 to read the data in a file.
     */
    private FileReader fr0 = null;
    /**
     * This is used with br0 to read the data in a file.
     */
    private BufferedReader br0 = null;
    /**
     * This is used with pw0 to read the data in a file.
     */
}
```

```
private FileWriter fwO = null;
/**
 * This is used with fwO to read the data in a file.
 */
private PrintWriter pwO = null;

/**
 * @param filePath
 */
/**
 * Sets the file path of the text file
 *
 * @param filePath
 *      the file path of the file to be accessed
 */
public void setFilePath(String filePath) {
    close();
    this.filePath = filePath;
}

/**
 * Closes the buffers accessing the file
 */
public void close() {
    try {
        if (frO != null) {
            frO.close();
            brO.close();
            frO = null;
            brO = null;
        }

        if (fwO != null) {
            fwO.close();
            pwO.close();
            fwO = null;
            pwO = null;
        }
    } catch (Exception e) {
    }
}

/**
 * Changes the I/O method to the specified I/O method, it can be either
```

```
* read, write, or writing and appending.  
*  
* @param io  
*       the IO method  
* @throws Exception  
*       if the file cannot be found, opened etc.  
*/  
public void setIO(int io) throws Exception {  
    close();  
  
    switch (io) {  
        case (READ):  
            frO = new FileReader(filePath);  
            brO = new BufferedReader(frO);  
            break;  
  
        case (WRITE):  
            fwO = new FileWriter(filePath, false);  
            pwO = new PrintWriter(fwO);  
            break;  
  
        case (WRITE_APPEND):  
            fwO = new FileWriter(filePath, true);  
            pwO = new PrintWriter(fwO);  
            break;  
    }  
}  
  
/**  
 * @return the number of lines in the text file  
 * @throws Exception  
 *       if the file cannot be accessed properly.  
 */  
public int getNumLines() throws Exception {  
    return TextFile.getNumLines(this.filePath);  
}  
  
/**  
 * @param filePath  
*       the path of the file whose number of lines will be counted.  
* @return the number of lines in the specified file  
* @throws Exception  
*       if the file cannot be accessed properly  
*/
```

```
public static int getNumLines(String filePath) throws Exception {
    FileReader fr20 = null;
    BufferedReader br20 = null;
    int numLines = 0;

    fr20 = new FileReader(filePath);
    br20 = new BufferedReader(fr20);

    while (br20.readLine() != null)
        numLines++;

    fr20.close();
    br20.close();

    return numLines;
}

/**
 * @return the next letter read from the file.
 * @throws Exception
 *          if the file cannot be accessed properly
 */
public String readLetter() throws Exception {
    String data = null;
    int charCode;

    if (fr20 != null) {
        if ((charCode = br20.read()) != -1)
            data = Character.toString((char) charCode);
    }

    return data;
}

/**
 * @return the next line read in the file
 * @throws Exception
 *          if the file cannot be accessed properly
 */
public String readLine() throws Exception {
    String data = null;

    if ((fr20 != null) && (br20 != null)) {
        data = br20.readLine();
    }
}
```

```
    }

    return data;
}

/***
 * @return an array of strings containing the data of each line in the file
 * @throws Exception
 *         if the file cannot be accessed properly
 */
public String[] readAllLines() throws Exception {
    int numLines = getNumLines();
    String[] lines = new String[numLines];

    for (int i = 0; i < numLines; ++i) {
        lines[i] = this.readLine();
    }

    return lines;
}

/***
 * @param data
 *         writes the specified data to the file
 * @throws Exception
 *         if the file cannot be accessed properly
 */
public void writeLine(String data) throws Exception {
    writeLineGeneral(data);
}

/***
 * @param data
 *         writes the specified data to the file
 * @throws Exception
 *         if the file cannot be accessed properly
 */
public void writeLine(char data) throws Exception {
    writeLineGeneral(Character.toString(data));
}

/***
 * @param data
 *         writes the specified data to the file
 */
```

```
* @throws Exception
*           if the file cannot be accessed properly
*/
public void writeLine(double data) throws Exception {
    writeLineGeneral(Double.toString(data));
}

/**
 * @param data
 *           writes the specified data to the file
 * @throws Exception
*           if the file cannot be accessed properly
*/
public void writeLine(long data) throws Exception {
    writeLineGeneral(Long.toString(data));
}

/**
 * @param data
 *           writes the specified data to the file and adds a new line
 *           character.
 * @throws Exception
*           if the file cannot be accessed properly
*/
private void writeLineGeneral(String data) throws Exception {
    pwo.printf("%s" + "%n", data);
}

/**
 * @param data
 *           writes the specified data to the file
 * @throws Exception
*           if the file cannot be accessed properly
*/
public void write(String data) throws Exception {
    writeGeneral(data);
}

/**
 * @param data
 *           writes the specified data to the file
 * @throws Exception
*           if the file cannot be accessed properly
*/
```

```
public void write(char data) throws Exception {
    writeGeneral(Character.toString(data));
}

/**
 * @param data
 *          writes the specified data to the file
 * @throws Exception
 *          if the file cannot be accessed properly
 */
public void write(double data) throws Exception {
    writeGeneral(Double.toString(data));
}

/**
 * @param data
 *          writes the specified data to the file
 * @throws Exception
 *          if the file cannot be accessed properly
 */
public void write(long data) throws Exception {
    writeGeneral(Long.toString(data));
}

/**
 * @param data
 *          writes the specified data to the file.
 * @throws Exception
 *          if the file cannot be accessed properly
 */
private void writeGeneral(String data) throws Exception {
    pw0.printf("%s", data);
}

}
```

## Class TimeGraph

```
package jCube;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Event;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.io.File;
import java.io.IOException;

import javax.swing.ButtonGroup;
import javax.swing.ImageIcon;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.KeyStroke;
import javax.swing.filechooser.FileFilter;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.ChartUtilities;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.CategoryPlot;
import org.jfree.data.category.DefaultCategoryDataset;

/**
 * @author Kelsey McKenna
 */
public class TimeGraph extends JFrame {
    /**
     * Default serialVersionUID
     */
    private static final long serialVersionUID = 1L;
    /**
     * This is the chart displayed on the screen.
     */
}
```

```
private JFreeChart chart;
/**
 * The chart is placed 'inside' this variable so that it can be displayed.
 */
private ChartPanel chartPanel;
/**
 * This stores the dataset that is rendered onto the graph.
 */
private DefaultCategoryDataset dataset;
/**
 * This stores the main title of the chart.
 */
private String chartTitle;
/**
 * This stores the label that is shown on the x-axis.
 */
private String xLabel;
/**
 * This stores the label that is shown on the y-axis.
 */
private String yLabel;
/**
 * This variable stores either '2' or '3', indicating that the graph should
 * be displayed in 2D or 3D respectively.
 */
private int dimension = 2;
/**
 * This stores the properties of the stroke used to paint the graph.
 */
private BasicStroke stroke = new BasicStroke(4.0f);
/**
 * If true, then the window will be on top of all other windows, otherwise
 * it can be hidden.
 */
private boolean alwaysOnTop = true;

/**
 * Constructor - assigns values to fields and sets up Time Graph window.
 *
 * @param applicationTitle
 *          the main title of the window
 * @param chartTitle
 *          the title of the chart
 * @param xLabel
```

```
*           the label shown on the x axis
* @param yLabel
*           the label shown on the y axis
*/
public TimeGraph(String applicationTitle, String chartTitle, String xlabel, String ylabel) {
    super(applicationTitle);
    this.chartTitle = chartTitle;
    this.xlabel = xlabel;
    this.ylabel = ylabel;

    setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
    setAlwaysOnTop(true);
    setJMenuBar(new MenuBar().createMenuBar());
}

/**
 * Draws the graph in the window
 */
public void draw() {
    if (dimension == 2)
        chart = ChartFactory.createLineChart(chartTitle, xlabel, ylabel, dataset);
    else
        chart = ChartFactory.createLineChart3D(chartTitle, xlabel, ylabel, dataset);

    chart.removeLegend();

    CategoryPlot plot = (CategoryPlot) chart.getPlot();
    plot.getRenderer().setSeriesPaint(0, Color.BLUE);
    plot.getRenderer().setSeriesStroke(0, stroke);

    chartPanel = new ChartPanel(chart);
    setContentPane(chartPanel);
}

/**
 * Sets the title of the chart
 *
 * @param chartTitle
 *           the title of the chart
 */
public void setGraphTitle(String chartTitle) {
    this.chartTitle = chartTitle;
}
```

```
/**  
 * Resets the graph to its default zoom.  
 */  
public void resetGraphView() {  
    chartPanel.restoreAutoBounds();  
}  
  
/**  
 * Sets the dataset for the graph  
 *  
 * @param dataset  
 *         the dataset to be used to draw the graph  
 */  
public void setDataset(DefaultCategoryDataset dataset) {  
    this.dataset = dataset;  
}  
  
/**  
 * @author Kelsey McKenna  
 */  
private class MenuBar {  
    /**  
     * @author Kelsey McKenna  
     */  
    private class ImageFilter extends FileFilter {  
        /**  
         * Used to prevent typing errors etc.  
         */  
        public final static String png = "png";  
  
        /**  
         * @see javax.swing.filechooser.FileFilter#accept(java.io.File)  
         */  
        public boolean accept(File f) {  
            if (f.isDirectory()) {  
                return true;  
            }  
  
            String extension = getExtension(f);  
            if (extension != null) {  
                if (extension.equals(png)) {  
                    return true;  
                } else {  
                    return false;  
                }  
            }  
        }  
    }  
}
```

```
        }
    }

    return false;
}

/**
 * @see javax.swing.filechooser.FileFilter#getDescription()
 */
public String getDescription() {
    return ".png - Extension will be appended automatically.";
}

/**
 * @param file
 *          the file to be checked
 * @return the extension of the file
 */
public String getExtension(File file) {
    String extension = null;
    String fileName = file.getName();
    int i = fileName.lastIndexOf('.');

    if ((i > 0) && (i < fileName.length() - 1)) {
        extension = fileName.substring(i + 1).toLowerCase();
    }
    return extension;
}

/**
 * This stores the contents of the menu bar
 */
private JMenuBar menuBar;
/**
 * Stores the contents of the file menu
 */
private JMenu fileMenu;
/**
 * Stores the contents of the view menu
 */
private JMenu viewMenu;
/**
 * Clicking this menu item opens a save dialog so that the graph can be
```

```
* saved as an image in the specified location.  
*/  
private JMenuItem saveImageItem;  
/**  
 * Clicking this button resets the graph to its default zoom.  
 */  
private JMenuItem resetZoomItem;  
/**  
 * Clicking this button closes the Time Graph window.  
 */  
private JMenuItem closeWindowItem;  
/**  
 * Clicking this button toggles whether the window is always on top.  
 */  
private JCheckBoxMenuItem alwaysOnTopItem;  
/**  
 * This holds the two dimension radio buttons so that only one of the  
 * radio buttons can be selected.  
 */  
private ButtonGroup dimensionButtonGroup;  
/**  
 * Selecting this radio button changes the graph so that it is rendered  
 * in 2D  
 */  
private JRadioButtonMenuItem radioButton2D;  
/**  
 * Selecting this radio button changes the graph so that it is rendered  
 * in 3D  
 */  
private JRadioButtonMenuItem radioButton3D;  
/**  
 * This variable is used to select a location to save or load scrambles.  
 */  
private JFileChooser fileChooser = new JFileChooser(System.getProperty("user.home") + "/Desktop") {  
    private static final long serialVersionUID = 1L;  
  
    @Override  
    public void approveSelection() {  
        setSelectedFile(new File(((("") + getSelectedFile()).replaceAll("\\\\.", "") + ".png"));  
        File f = new File(getSelectedFile().toString());  
  
        if ((f.exists()) && (getDialogType() == SAVE_DIALOG)) {  
            int result = JOptionPane.showConfirmDialog(this, String.format(
```

```
        "The file %s already exists. Do you want to overwrite?", getFile().toString(),
        "Existing file", JOptionPane.YES_NO_OPTION);
    switch (result) {
        case JOptionPane.YES_OPTION:
            super.approveSelection();
            return;
        case JOptionPane.NO_OPTION:
            return;
        case JOptionPane.CLOSED_OPTION:
            return;
    }
    super.approveSelection();
};

/**
 * @return the menu bar for the Time Graph window
 */
public JMenuBar createMenuBar() {
    ImageIcon icon;
    menuBar = new JMenuBar();
    dimensionButtonGroup = new ButtonGroup();
    fileChooser.setAcceptAllFileFilterUsed(false);
    fileChooser.addChoosableFileFilter(new ImageFilter());

    // **** FILE ****
    fileMenu = new JMenu("File");
    menuBar.add(fileMenu);

    icon = Main.createImageIcon("res/images/SaveIcon.png");
    saveImageItem = new JMenuItem("Save as Image", icon);
    saveImageItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            boolean alwaysOnTopValue = alwaysOnTop;

            alwaysOnTop = false;
            setAlwaysOnTop(false);
            if (fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
                int width = 1024, height = 768;
                File lineChart = new File(fileChooser.getSelectedFile() + "");

                try {
                    ChartUtilities.saveChartAsPNG(lineChart, chart, width, height);
                }
            }
        }
    });
}
```

```
        } catch (IOException exc) {
    }
}
alwaysOnTop = alwaysOnTopValue;
setAlwaysOnTop(alwaysOnTop);
});
saveImageItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S, Event.CTRL_MASK));

closeWindowItem = new JMenuItem("Close Window");
closeWindowItem.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        setVisible(false);
    }
});
closeWindowItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_W, Event.CTRL_MASK));

fileMenu.add(saveImageItem);
fileMenu.addSeparator();
fileMenu.add(closeWindowItem);
/************* END FILE *******/

/*********** VIEW *****/
viewMenu = new JMenu("View");
menuBar.add(viewMenu);

alwaysOnTopItem = new JCheckBoxMenuItem("Window Always on Top");
alwaysOnTopItem.setSelected(true);
alwaysOnTopItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        alwaysOnTop = !alwaysOnTop;
        setAlwaysOnTop(alwaysOnTop);
    }
});
resetZoomItem = new JMenuItem("Reset Zoom");
resetZoomItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        resetGraphView();
        Main.refreshTimeGraph(false);
    }
});
resetZoomItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_R, Event.CTRL_MASK));
```

```
radioButton2D = new JRadioButtonMenuItem("2D");
radioButton2D.setSelected(true);
radioButton2D.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        dimension = 2;
        Main.refreshTimeGraph(false);
    }
});
radioButton2D.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_2, Event.CTRL_MASK));

radioButton3D = new JRadioButtonMenuItem("3D");
radioButton3D.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        dimension = 3;
        Main.refreshTimeGraph(false);
    }
});
radioButton3D.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_3, Event.CTRL_MASK));

dimensionButtonGroup.add radioButton2D;
dimensionButtonGroup.add radioButton3D;

viewMenu.add(alwaysOnTopItem);
viewMenu.add(resetZoomItem);
viewMenu.addSeparator();
viewMenu.add radioButton2D;
viewMenu.add radioButton3D;
/********** END VIEW *****/
}

return menuBar;
}
}
```

## Class TimeListPopUp

```
package jCube;

import java.awt.Dimension;
import java.awt.Font;
import java.awt.Insets;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.io.File;

import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

/**
 * @author Kelsey McKenna
 */
public class TimeListPopUp extends JFrame implements KeyListener {
    /**
     * Default serialVersionUID
     */
    private static final long serialVersionUID = 1L;
    /**
     * This stores the padding of the elements in the window. The greater the
     * padding, the further towards the centre of the window the elements will
     * be.
     */
    private final int pad = 10;
    /**
     * This indicates the vertical spacing between the text boxes etc. in the
     * window.
     */
    private final int fieldYSpacing = 50;
    /**
     * This indicates the vertical spacing between the buttons in the window.
```

```
/*
private final int buttonYSpacing = 40;

/**
 * This variable keeps track of the current y position of the last element
 * placed in the window.
 */
private int y = 0;

/**
 * This panel stores the elements of the Solve Editor window.
 */
private JPanel contentPane;

/**
 * This label is shown in the Solve Editor window with the text 'Time'.
 */
private JLabel timeLabel;
/**
 * This label is shown in the Solve Editor window with the text 'Penalty'.
 */
private JLabel penaltyLabel;
/**
 * This label is shown in the Solve Editor window with the text 'Comment'.
 */
private JLabel commentLabel;
/**
 * This label is shown in the Solve Editor window with the text 'Scramble'.
 */
private JLabel scrambleLabel;
/**
 * This label is shown in the Solve Editor window with the text 'Solution'.
 */
private JLabel solutionLabel;

/**
 * This field is shown in the Solve Editor window and the user can enter the
 * time into this field.
 */
private JTextField timeField;
/**
 * This field is shown in the Solve Editor window and the user can enter the
 * penalty into this field.
 */
```

```
private JTextField penaltyField;
/**
 * This field is shown in the Solve Editor window and the user can enter the
 * comment into this field.
 */
private JTextArea commentField;
/**
 * This field is shown in the Solve Editor window and the user can enter the
 * scramble into this field.
 */
private JTextArea scrambleField;
/**
 * This field is shown in the Solve Editor window and the user can enter the
 * solution into this field.
 */
private JTextArea solutionField;

/**
 * commentField is placed 'inside' this variable so that when the length of
 * the comment exceeds the size of commentField, the user can scroll to view
 * the rest of the comment.
 */
private JScrollPane commentScrollPane;
/**
 * scrambleField is placed 'inside' this variable so that when the length of
 * the scramble exceeds the size of scrambleField, the user can scroll to
 * view the rest of the scramble.
 */
private JScrollPane scrambleScrollPane;
/**
 * solutionField is placed 'inside' this variable so that when the length of
 * the solution exceeds the size of solutionField, the user can scroll to
 * view the rest of the solution.
 */
private JScrollPane solutionScrollPane;

/**
 * Clicking this button submits the data in the Solve Editor form for
 * validation.
 */
private JButton submitButton;
/**
 * Clicking this button opens a save dialog so that the information in the
 * Solve Editor window can be saved to a chosen location.
```

```
/*
private JButton saveToFileButton;
/**
 * Clicking this button discards any changes and reloads the information in
 * the window.
 */
private JButton restoreValuesButton;
/**
 * Clicking this button deletes the time/solve being viewed
 */
private JButton deleteTimeButton;
/**
 * Clicking this button scrambles the cube in the main window and performs
 * the solution in real time.
 */
private JButton viewExecutionButton;

/**
 * This stores the Solve currently being edited.
 */
private Solve currentTime = new Solve("0", "0", "");
/**
 * Stores the font to be used in the text fields.
 */
private Font fieldFont = new Font("Arial", 0, 25);
/**
 * This variable stores the text that is shown in the error message when the
 * user enters an invalid time.
 */
private String helpMessage = "You entered the time incorrectly\n\n" + "Valid formats include:\n" + "MM:SS.ss\n"
    + "MM:S.ss\n" + "M:SS.ss\n" + "M:S.ss\n" + "SS.ss\n" + "S.ss\n" + "DNF\n";

/**
 * This variable is used to select a location to save or load scrambles.
 */
private JFileChooser fileChooser = new JFileChooser(System.getProperty("user.home") + "/Desktop") {
    private static final long serialVersionUID = 1L;

    @Override
    public void approveSelection() {
        // String filePath = getSelectedFile() + ".txt";
        // filePath = filePath.substring(0, filePath.indexOf(".txt")) +
        // ".txt";
        setSelectedFile(new File(("" + getSelectedFile()).replaceAll("\\\\.", "") + ".txt"));
    }
}
```

```
File f = new File(getSelectedFile().toString());

if ((f.exists()) && (getDialogType() == SAVE_DIALOG)) {
    int result = JOptionPane.showConfirmDialog(this, String.format(
        "The file %s already exists. Do you want to overwrite?", getSelectedFile().toString()),
        "Existing File", JOptionPane.YES_NO_OPTION);
    switch (result) {
        case JOptionPane.YES_OPTION:
            super.approveSelection();
            return;
        case JOptionPane.NO_OPTION:
            return;
        case JOptionPane.CLOSED_OPTION:
            return;
    }
}
super.approveSelection();
};

/***
 * Constructor - sets up the Solve Editor window
 *
 * @param applicationTitle
 *          the title for the window
 */
public TimeListPopUp(String applicationTitle) {
    super(applicationTitle);

    contentPane = new JPanel();
    contentPane.setLayout(null);

    setIconImage(Main.createImage("res/images/RubikCubeBig.png"));
    setContentPane(contentPane);
    setPreferredSize(new Dimension(480, 520));
    setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    setResizable(false);
    setLocation(600, 150);

    // *****Labels*****
    timeLabel = new JLabel("Time ");
    timeLabel.setSize(120, 40);
    timeLabel.setLocation(0 + pad, y + pad);
```

```
y += fieldYSpacing;

penaltyLabel = new JLabel("Penalty ");
penaltyLabel.setSize(120, 40);
penaltyLabel.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

commentLabel = new JLabel("Comment ");
commentLabel.setSize(120, 40);
commentLabel.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

scrambleLabel = new JLabel("Scramble ");
scrambleLabel.setSize(120, 40);
scrambleLabel.setLocation(0 + pad, y + pad);

y += fieldYSpacing;

solutionLabel = new JLabel("Solution ");
solutionLabel.setSize(120, 40);
solutionLabel.setLocation(0 + pad, y + pad);

// *****Text Fields*****
y = 0;

timeField = new JTextField();
timeField.setMargin(new Insets(0, 10, 0, 15));
timeField.setFont(fieldFont);
timeField.setSize(350, 40);
timeField.setLocation(100 + pad, y + pad);
timeField.addKeyListener(this);

y += fieldYSpacing;

penaltyField = new JTextField(currentTime.getPenalty());
penaltyField.setMargin(new Insets(0, 10, 0, 15));
penaltyField.setFont(new Font("Arial", 0, 15));
penaltyField.setSize(350, 40);
penaltyField.setLocation(100 + pad, y + pad);
penaltyField.addKeyListener(this);
```

```
y += fieldYSpacing;

commentField = new JTextArea();
commentField.setMargin(new Insets(0, 10, 0, 15));
commentField.setLineWrap(true);
commentField.setWrapStyleWord(true);
commentField.setEditable(true);
commentField.setFont(new Font("Arial", 0, 15));
commentField.addKeyListener(this);

commentScrollPane = new JScrollPane(commentField);
commentScrollPane.setLocation(100 + pad, y + pad);
commentScrollPane.setSize(new Dimension(350, 40));
commentScrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);

y += fieldYSpacing;

scrambleField = new JTextArea();
scrambleField.setMargin(new Insets(0, 10, 0, 15));
scrambleField.setEditable(true);
scrambleField.setLineWrap(true);
scrambleField.setWrapStyleWord(true);
scrambleField.setFont(new Font("Arial", 0, 15));
scrambleField.addKeyListener(this);

scrambleScrollPane = new JScrollPane(scrambleField);
scrambleScrollPane.setLocation(100 + pad, y + pad);
scrambleScrollPane.setSize(new Dimension(350, 40));
scrambleScrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);

y += fieldYSpacing;

solutionField = new JTextArea();
solutionField.setMargin(new Insets(0, 10, 0, 15));
solutionField.setEditable(true);
solutionField.setLineWrap(true);
solutionField.setWrapStyleWord(true);
solutionField.setFont(new Font("Arial", 0, 15));
solutionField.addKeyListener(this);

solutionScrollPane = new JScrollPane(solutionField);
solutionScrollPane.setLocation(100 + pad, y + pad);
solutionScrollPane.setSize(new Dimension(350, 80));
solutionScrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
```

```
// *****Submission Button*****
y = 290;

submitButton = new JButton("Submit");
submitButton.setSize(480, 30);
submitButton.setLocation(0, y + pad);
submitButton.setFocusable(false);
submitButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        String time = timeField.getText().trim();

        if (!Solve.isValidTime(time)) {
            JOptionPane.showMessageDialog(contentPane, helpMessage, "Error", JOptionPane.ERROR_MESSAGE, null);
        } else {
            if (time.equalsIgnoreCase("DNF")) {
                currentTime.setStringTime("DNF");
                currentTime.setPenalty("0");
            } else {
                currentTime.setStringTime(Solve.getPaddedTime(Solve.getSecondsToFormattedString(Solve
                    .getFormattedString.ToDouble(time))));
                currentTime.setPenalty(penaltyField.getText().trim());
            }
        }

        if (penaltyField.getText().trim().equals(""))
            currentTime.setPenalty("0");

        currentTime.setComment(commentField.getText().trim());
        currentTime.setScramble(scrambleField.getText().trim());
        currentTime.setSolution(solutionField.getText().trim());
        setVisible(false);
        Main.refreshTimeGraph(true);
        Main.refreshStatistics();
    }
});
y += buttonYSpacing;

saveToFileButton = new JButton("Save to File");
saveToFileButton.setSize(480, 30);
saveToFileButton.setLocation(0, y + pad);
saveToFileButton.setFocusable(false);
```

```
saveToFileButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        String time = timeField.getText().trim();

        if (!Solve.isValidTime(time)) {
            JOptionPane.showMessageDialog(contentPane, helpMessage, "Error", JOptionPane.ERROR_MESSAGE, null);
        } else {
            if (time.equalsIgnoreCase("DNF")) {
                currentTime.setStringTime("DNF");
                currentTime.setPenalty("0");
            } else {
                currentTime.setStringTime(Solve.getPaddedTime(time));
                currentTime.setPenalty(penaltyField.getText().trim());
            }

            currentTime.setComment(commentField.getText().trim());
            currentTime.setScramble(scrambleField.getText().trim());
            currentTime.setSolution(solutionField.getText().trim());
            setVisible(false);

            if (fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
                TextFile currentFile = new TextFile();

                try {
                    currentFile.setFilePath(fileChooser.getSelectedFile().toString());
                    currentFile.setIO(TextFile.WRITE);

                    String comment = commentField.getText().trim();
                    String scramble = scrambleField.getText().trim();
                    String solution = solutionField.getText().trim();

                    currentFile.writeLine(timeField.getText());
                    currentFile.writeLine(penaltyField.getText());
                    currentFile.writeLine(comment.equals("") ? "*" : comment);
                    currentFile.writeLine(scramble.equals("") ? "*" : scramble);
                    currentFile.write(solution.equals("") ? "*" : solutionField.getText());
                } catch (Exception e) {
                    JOptionPane.showMessageDialog(contentPane, "Could not save to file", "Error",
                        JOptionPane.ERROR_MESSAGE);
                } finally {
                    currentFile.close();
                }
            }
        }
    }
})
```

```
        }
    });
});

y += buttonYSpacing;

viewExecutionButton = new JButton("View Execution");
viewExecutionButton.setSize(480, 30);
viewExecutionButton.setLocation(0, y + pad);
viewExecutionButton.setFocusable(false);
viewExecutionButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        if (!(currentTime.getStringTime()).equals(timeField.getText()))
            || !(currentTime.getPenalty()).equals(penaltyField.getText())
            || !(currentTime.getComment()).equals(commentField.getText())
            || !(currentTime.getScramble()).equals(scrambleField.getText())
            || !(currentTime.getSolution()).equals(solutionField.getText())) {
            JOptionPane.showMessageDialog(contentPane, "You must submit the form first.", "Error",
                JOptionPane.ERROR_MESSAGE);
        } else {
            Main.performRealTimeSolving(currentTime.getScramble(), currentTime.getSolution());
        }
    }
});
y += buttonYSpacing;

restoreValuesButton = new JButton("Restore");
restoreValuesButton.setSize(480, 30);
restoreValuesButton.setLocation(0, y + pad);
restoreValuesButton.setFocusable(false);
restoreValuesButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        timeField.setText(currentTime.getStringTime());
        penaltyField.setText(currentTime.getPenalty());
        commentField.setText(currentTime.getComment());
        scrambleField.setText(currentTime.getScramble());
        solutionField.setText(currentTime.getSolution());
    }
});
y += buttonYSpacing;
```

```
deleteTimeButton = new JButton("Delete");
deleteTimeButton.setSize(480, 30);
deleteTimeButton.setLocation(0, y + pad);
deleteTimeButton.setFocusable(false);
deleteTimeButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        currentTime.setStringTime("-1");
        Main.copyAllTimesToDisplay();
        Main.refreshTimeList();
        setVisible(false);
    }
});

contentPane.add(timeLabel);
contentPane.add(penaltyLabel);
contentPane.add(commentLabel);
contentPane.add(scrambleLabel);
contentPane.add(solutionLabel);

contentPane.add(timeField);
contentPane.add(penaltyField);
contentPane.add(commentScrollPane);
contentPane.add(scrambleScrollPane);
contentPane.add(solutionScrollPane);

contentPane.add(submitButton);
contentPane.add(saveToFileButton);
contentPane.add(viewExecutionButton);
contentPane.add	restoreValuesButton);
contentPane.add(deleteTimeButton);
}

/**
 * Selects the text in the time field so that the user can edit it without
 * having to use backspace.
 */
public void selectAllTimeText() {
    timeField.selectAll();
}

/**
 * Sets the current solve to work with in the Solve Editor window.
 *
 * @param currentSolve

```

```
*           the solve to work with.  
*/  
public void setSolve(Solve currentSolve) {  
    this.currentTime = currentSolve;  
    currentSolve.setStringTime(Solve.getPaddedTime(currentSolve.getStringTime()));  
    restoreValuesButton.doClick();  
}  
  
/**  
 * @see java.awt.event.KeyListener#keyPressed(java.awt.event.KeyEvent)  
 */  
@Override  
public void keyPressed(KeyEvent arg0) {  
    Main.transferFormFocus(arg0);  
}  
  
/**  
 * @see java.awt.event.KeyListener#keyReleased(java.awt.event.KeyEvent)  
 */  
@Override  
public void keyReleased(KeyEvent arg0) {  
}  
  
/**  
 * @see java.awt.event.KeyListener#keyTyped(java.awt.event.KeyEvent)  
 */  
@Override  
public void keyTyped(KeyEvent arg0) {  
    Object source = arg0.getSource();  
  
    if (arg0.getKeyChar() == KeyEvent.VK_ENTER) {  
        if (!arg0.isShiftDown())  
            submitButton.doClick();  
        else if (source.equals(commentField) ||  
                 source.equals(scrambleField) ||  
                 source.equals(solutionField)) {  
            ((JTextArea) source).setText(((JTextArea) source).getText() + "\n");  
        }  
    }  
}
```

## Class Tutorial

```
package jCube;

import java.awt.Color;
import java.util.Arrays;
import java.util.LinkedList;

/**
 * @author Kelsey McKenna
 */
public class Tutorial {

    /**
     * This represents the cube displayed on screen. It can be used to generate
     * a solution for the current state of the cube.
     */
    private Cube cube;
    /**
     * This variable allows certain methods to be accessed to perform general
     * operations on the cube.
     */
    private SolveMaster solveMaster;
    /**
     * This variable allows a solution for the cross to be generated.
     */
    private CrossSolver crossSolver;
    /**
     * This variable allows a solution for the first-layer corners to be
     * generated.
     */
    private CornerSolver cornerSolver;
    /**
     * This variable allows a solution for the middle-layer edges to be
     * generated.
     */
    private EdgeSolver edgeSolver;
    /**
     * This variable allows a solution for the orientation of the last layer to
     * be generated.
     */
    private OrientationSolver orientationSolver;
    /**
     * This variable allows a solution for the permutation of the last layer to

```

```
* be generated.  
*/  
private PermutationSolver permutationSolver;  
  
/**  
 * -The ith element of this array stores the scramble for the ith  
 * sub-tutorial.  
 */  
private String[] scrambles;  
/**  
 * The ith element of this array stores the description for the ith  
 * sub-tutorial.  
 */  
private String[] descriptions;  
/**  
 * The ith element of this array stores the pieces that are expected to be  
 * solved for the ith sub-tutorial.  
 */  
private String[][] expectedSolvedPieces;  
/**  
 * The ith element of this array stores the hints for the ith sub-tutorial.  
 */  
private String[][] hints;  
/**  
 * The ith element of this array stores the optimal solutions for the ith  
 * sub-tutorial.  
 */  
private String[][] optimalSolutions;  
/**  
 * The ith element of this array stores the solution for the ith  
 * sub-tutorial.  
 */  
private String[] explanations;  
/**  
 * This variable is used to open the file containing the tutorial.  
 */  
private TextFile currentFile;  
/**  
 * This stores the index of the next hint to be shown.  
 */  
private int hintIndex = -1;  
/**  
 * This stores the index of the current sub-tutorial.  
 */
```

```
private int subTutorialIndex = 0;
/**
 * Each element of this array stores a single line in the text file being
 * read.
 */
private String[] fileData;
/**
 * This stores the number of sub-tutorials.
 */
private int numSubTutorials;
/**
 * If true, then a tutorial has been loaded.
 */
private boolean tutorialLoaded;
/**
 * If the ith element of the array is true, then during the ith tutorial the
 * user is granted permission to perform moves.
 */
private boolean[] tutorialsRequiringUserAction;
/**
 * If the ith element of the array is true, then during the ith tutorial the
 * users actions will be checked to see if they fulfil certain criteria.
 */
private boolean[] tutorialsRequiringUserSolution;
/**
 * This stores the string "cross". Used for readability.
 */
private static final String CROSS_SOLVED = "cross";
/**
 * This stores the string "corners". Used for readability.
 */
private static final String CORNERS_SOLVED = "corners";
/**
 * This stores the string "edges". Used for readability.
 */
private static final String EDGES_SOLVED = "edges";
/**
 * This stores the string "edge orientation". Used for readability.
 */
private static final String EDGE_ORIENTATION_SOLVED = "edge orientation";
/**
 * This stores the string "corner orientation". Used for readability.
 */
private static final String CORNER_ORIENTATION_SOLVED = "corner orientation";
```

```
/**  
 * This stores the string "oll". Used for readability.  
 */  
private static final String ORIENTATION_SOLVED = "oll";  
/**  
 * This stores the string "corner permutation". Used for readability.  
 */  
private static final String CORNER_PERMUTATION_SOLVED = "corner permutation";  
/**  
 * This stores the string "edge permutation". Used for readability.  
 */  
private static final String EDGE_PERMUTATION_SOLVED = "edge permutation";  
/**  
 * This stores the string "pll". Used for readability.  
 */  
private static final String PERMUTATION_SOLVED = "pll";  
  
/**  
 * Constructor - sets up fields  
 *  
 * @param cube  
 *          the cube to be used during tutorials.  
 */  
public Tutorial(Cube cube) {  
    this.cube = cube;  
    solveMaster = new SolveMaster(cube);  
    crossSolver = new CrossSolver(cube);  
    cornerSolver = new CornerSolver(cube);  
    edgeSolver = new EdgeSolver(cube);  
    orientationSolver = new OrientationSolver(cube);  
    permutationSolver = new PermutationSolver(cube);  
  
    scrambles = new String[1];  
    descriptions = new String[1];  
    expectedSolvedPieces = new String[1][1];  
    hints = new String[1][1];  
    optimalSolutions = new String[1][1];  
    explanations = new String[1];  
    tutorialsRequiringUserAction = new boolean[1];  
}  
  
/**  
 * Loads the tutorial stored in the file at the specified file path. The  
 * fields store the data from the file.
```

```
*  
* @param filePath  
*         the path of the file which contains the tutorial  
* @throws Exception  
*         if the file cannot be accessed properly or is not formatted  
*         correctly  
*/  
public void loadTutorial(String filePath) throws Exception {  
    // Stores the number of lines in the the file.  
    int numLines;  
    // Stores the index of the current line in the file being processed.  
    int currentIndex = 0;  
    // Stores the index of the next heading in the file.  
    int headingIndex;  
    hintIndex = -1;  
    subTutorialIndex = 0;  
  
    currentFile = new TextFile();  
    currentFile.setFilePath(filePath);  
    currentFile.setIO(TextFile.READ);  
  
    fileData = currentFile.readAllLines();  
    numLines = fileData.length;  
    currentFile.close();  
  
    numSubTutorials = getNumSubTutorials();  
    scrambles = new String[numSubTutorials];  
    descriptions = new String[numSubTutorials];  
    expectedSolvedPieces = new String[numSubTutorials][20];  
    hints = new String[numSubTutorials][20];  
    optimalSolutions = new String[numSubTutorials][20];  
    explanations = new String[numSubTutorials];  
    tutorialsRequiringUserAction = new boolean[numSubTutorials];  
    tutorialsRequiringUserSolution = new boolean[numSubTutorials];  
  
    for (int i = 0; i < numSubTutorials; ++i) {  
        if (fileData[currentIndex].contains("ENABLE"))  
            tutorialsRequiringUserAction[i] = true;  
        else  
            tutorialsRequiringUserAction[i] = false;  
  
        currentIndex += 2;  
        scrambles[i] = fileData[currentIndex];
```

```
currentIndex += 2;

/*
 * This finds the next section in the sub-tutorial
 */
descriptions[i] = "";
while ((!fileData[currentIndex].equals("EXPECTED FINAL STATE:")) && (!fileData[currentIndex].equals("*"))) {
    descriptions[i] += fileData[currentIndex] + "\n";
    ++currentIndex;
}

descriptions[i] = descriptions[i].substring(0, descriptions[i].lastIndexOf("\n"));

if (fileData[currentIndex].equals("EXPECTED FINAL STATE:")) {
    tutorialsRequiringUserSolution[i] = true;
    ++currentIndex;

    /*
     * Finds and stores the pieces that are expected to be solved.
     * It does this by finding the index of the next heading and
     * then storing each line before that heading.
     */
    headingIndex = LinearSearch.linearSearchStartsWith(
        Arrays.copyOfRange(fileData, currentIndex, numLines), "HINT:");
    expectedSolvedPieces[i] = new String[headingIndex];
    for (int j = 0; j < headingIndex; ++j) {
        expectedSolvedPieces[i][j] = fileData[currentIndex];
        ++currentIndex;
    }
    ++currentIndex;

    /*
     * Finds and stores the hints. It does this by finding the index
     * of the next heading and then storing each line before that
     * heading.
     */
    headingIndex = LinearSearch.linearSearchStartsWith(
        Arrays.copyOfRange(fileData, currentIndex, numLines), "OPTIMAL SOLUTION:");
    hints[i] = new String[headingIndex];
    for (int j = 0; j < headingIndex; ++j) {
        hints[i][j] = fileData[currentIndex];
        ++currentIndex;
    }
    ++currentIndex;
```

```
/*
 * Finds and stores the optimal solutions. It does this by
 * finding the index of the next heading and then storing each
 * line before that heading.
 */
headingIndex = LinearSearch.linearSearchStartsWith(
    Arrays.copyOfRange(fileData, currentIndex, numLines), "EXPLANATION:");
optimalSolutions[i] = new String[headingIndex];
for (int j = 0; j < headingIndex; ++j) {
    optimalSolutions[i][j] = fileData[currentIndex];
    ++currentIndex;
}
++currentIndex;

/*
 * Finds and stores the explanation. It does this by finding the
 * index of the next heading and then appending each line before
 * the heading to explanations[i].
 */
headingIndex = LinearSearch.linearSearchStartsWith(
    Arrays.copyOfRange(fileData, currentIndex, numLines), "*");
explanations[i] = "";
for (int j = 0; j < headingIndex; ++j) {
    explanations[i] += fileData[currentIndex] + "\n";
    ++currentIndex;
}
++currentIndex;

explanations[i] = explanations[i].substring(0, explanations[i].lastIndexOf("\n"));
} else {
    ++currentIndex;
}
}

tutorialLoaded = true;
}

/**
 * @return the number of sub-tutorials in the tutorial
 */
private int getNumSubTutorials() {
    int length = fileData.length;
    int num = 0;
```

```
for (int i = 0; i < length; ++i) {
    if (fileData[i].startsWith("SCRAMBLE:"))
        ++num;
}

return num;
}

/**
 * Loads the next sub-tutorial, i.e. the sub-tutorial index is incremented
 * and other indices are reset.
 */
public void loadNextSubTutorial() {
    if (subTutorialIndex < numSubTutorials - 1) {
        ++subTutorialIndex;
        hintIndex = -1;
    }
}

/**
 * Loads the previous sub-tutorial, i.e. the sub-tutorial index is
 * decremented and other indices are reset.
 */
public void loadPreviousSubTutorial() {
    if (subTutorialIndex > 0) {
        --subTutorialIndex;
        hintIndex = -1;
    }
}

/**
 * @return the scramble for the current sub-tutorial
 */
public String getScramble() {
    return scrambles[subTutorialIndex];
}

/**
 * @return the description for the current sub-tutorial
 */
public String getDescription() {
    return String.format("(%d/%d): %n%s", subTutorialIndex + 1, scrambles.length, descriptions[subTutorialIndex]);
}
```

```
/**  
 * Loads the next hint for the current sub-tutorial, i.e. the hint index is  
 * incremented. The hint index will follow 0, 1, 2, ..., n, 0, 1, 2, ...  
 */  
public void loadNextHint() {  
    hintIndex = (hintIndex + 1) % hints[subTutorialIndex].length;  
}  
  
/**  
 * Loads the previous hint for the current sub-tutorial, i.e. the hint index  
 * is decremented. The hint index will follow ..., 2, 1, 0, n, n - 1, ...,  
 */  
public void loadPreviousHint() {  
    hintIndex = (hintIndex - 1 + hints.length) % hints[subTutorialIndex].length;  
}  
  
/**  
 * @return the current hint for the current sub-tutorial  
 */  
public String getHint() {  
    return String.format("Hint %d: %s", hintIndex + 1, hints[subTutorialIndex][hintIndex]);  
}  
  
/**  
 * @return the explanation for the current sub-tutorial  
 */  
public String getExplanation() {  
    return explanations[subTutorialIndex];  
}  
  
/**  
 * @return the optimal solutions for the current sub-tutorial  
 */  
public String[] getOptimalSolutions() {  
    return optimalSolutions[subTutorialIndex];  
}  
  
/**  
 * @return <b>true</b> if the criteria for the current sub-tutorial has been  
 *         filled by the user; <br>  
 *         <b>false</b> otherwise  
 */  
public boolean criteriaFilled() {
```

```
Cubie currentCubie;
String current;

try {
    for (int i = 0; i < expectedSolvedPieces[subTutorialIndex].length; ++i) {
        current = expectedSolvedPieces[subTutorialIndex][i].toLowerCase();

        switch (current) {
            case CROSS_SOLVED:
                if (!isCrossSolved())
                    return false;
                break;
            case CORNERS_SOLVED:
                if (!cornerSolver.firstLayerCornersSolved())
                    return false;
                break;
            case EDGES_SOLVED:
                if (!edgeSolver.middleLayerEdgesSolved())
                    return false;
                break;
            case EDGE_ORIENTATION_SOLVED:
                if (!orientationSolver.isEdgeOrientationSolved())
                    return false;
                break;
            case CORNER_ORIENTATION_SOLVED:
                if (!orientationSolver.isCornerOrientationSolved())
                    return false;
                break;
            case ORIENTATION_SOLVED:
                if (!orientationSolver.isOrientationSolved())
                    return false;
                break;
            case CORNER_PERMUTATION_SOLVED:
                if (!isCornerPermutationSolved())
                    return false;
                break;
            case EDGE_PERMUTATION_SOLVED:
                if (!isEdgePermutationSolved())
                    return false;
                break;
            case PERMUTATION_SOLVED:
                if (!(isCornerPermutationSolved() && isEdgePermutationSolved()))
                    return false;
                break;
        }
    }
}
```

```
default:
    boolean solved = true;
    currentCubie = new Cubie(getStickers(expectedSolvedPieces[subTutorialIndex][i]));
    solveMaster.clearMoves();
    solveMaster.rotateToTop(Color.white);
    solved = solveMaster.newPieceSolved(currentCubie);

    cube.performAbsoluteMoves(SolveMaster.getReverseStringMoves(solveMaster.getCatalogMoves()));

    return solved;
}
}

return true;
} catch (Exception e) {
    return false;
}
}

/***
 * This returns an array of Colors containing all colours denoted in the
 * <code>stickerFormat</code> parameter. <br>
 * For example, Input: <code>"r-g"</code> will return
 * <code>{Color.red, Color.green}</code>
 *
 * @param stickerFormat
 *         denotes the stickers with letters separated by '-' characters.
 * @return an array of Colors containing all colours denoted in the
 *         <code>stickerFormat</code> parameter.
 */
private Color[] getStickers(String stickerFormat) {
    Color[] stickers = new Color[(stickerFormat.length() + 1) / 2];

    for (int i = 0; i < stickers.length; ++i) {
        stickers[i] = Cubie.getWordToColor(stickerFormat.substring(i * 2, (i * 2) + 1));
    }

    return stickers;
}

/***
 * @return <b>true</b> if the current sub-tutorial requires/allows the user
 *         to perform moves, i.e. moves are not locked; <br>
 *         <b>false</b> otherwise
 */
```

```
/*
public boolean requiresUserAction() {
    return (tutorialsRequiringUserAction[subTutorialIndex]);
}

/**
 * @return <b>true</b> if the current sub-tutorial requires the user to
 *         solve a problem; <br>
 *         <b>false</b> otherwise
 */
public boolean requiresUserSolution() {
    return (tutorialsRequiringUserSolution[subTutorialIndex]);
}

/**
 * @return the number of moves in the first optimal solution.
 */
public int getOptimalSolutionLength() {
    return getNumMovesWithoutRotations(optimalSolutions[subTutorialIndex][0]);
}

/**
 * @param moves
 *         the moves to be analysed
 * @return the number of moves in <code>moves</code> but not counting
 *         rotations
 */
public static int getNumMovesWithoutRotations(String moves) {
    moves = moves.toUpperCase();
    int numMoves = 0;

    for (int i = 0; i < moves.length(); ++i) {
        if ("UDRLFBM".contains(moves.substring(i, i + 1)))
            ++numMoves;
    }

    return numMoves;
}

/**
 * @param moves
 *         the moves to be analysed
 * @return the number of moves in <code>moves</code> but not counting
 *         rotations
 */
```

```
/*
public static int getNumMovesWithoutRotations(LinkedList<String> moves) {
    String stringMoves = "";
    int size = moves.size();

    for (int i = 0; i < size; ++i) {
        stringMoves += moves.get(i) + " ";
    }

    return getNumMovesWithoutRotations(stringMoves);
}

/**
 * @return <b>true</b> if the tutorial is loaded; <br>
 *         <b>false</b> otherwise
 */
public boolean isLoaded() {
    return tutorialLoaded;
}

/**
 * @return <b>true</b> if the cross is solved; <br>
 *         <b>false</b> otherwise
 */
private boolean isCrossSolved() {
    Edge[] crossEdges = new Edge[4];

    for (int i = 0; i < 4; ++i)
        crossEdges[i] = new Edge(Edge.getInitialStickers(i));

    for (int i = 0; i < 4; ++i) {
        if (!crossSolver.isPieceSolved(crossEdges[i]))
            return false;
    }

    return true;
}

/**
 * @return <b>true</b> if the permutation of the corners is solved; <br>
 *         <b>false</b> otherwise
 */
private boolean isCornerPermutationSolved() {
    Corner[] corners = new Corner[4];
```

```
for (int i = 4; i < 8; ++i)
    corners[i - 4] = new Corner(Corner.getInitialStickers(i));

for (int i = 0; i < 4; ++i)
    if (!permutationSolver.isPieceSolved(corners[i]))
        return false;

return true;
}

/**
 * @return <b>true</b> if the permutation of the edges is solved; <br>
 *         <b>false</b> otherwise
 */
private boolean isEdgePermutationSolved() {
    Edge[] edges = new Edge[4];

    for (int i = 8; i < 12; ++i)
        edges[i - 8] = new Edge(Edge.getInitialStickers(i));

    for (int i = 0; i < 4; ++i)
        if (!permutationSolver.isPieceSolved(edges[i]))
            return false;

    return true;
}

/**
 * @return the index of the current sub-tutorial
 */
public int getSubTutorialIndex() {
    return subTutorialIndex;
}

/**
 * @return <b>true</b> if the current sub-tutorial is the first
 *         sub-tutorial; <br>
 *         <b>false</b> otherwise
 */
public boolean isFirstSubTutorial() {
    return subTutorialIndex == 0;
}
```

```
/**  
 * @return <b>true</b> if the current sub-tutorial is the last sub-tutorial; <br>  
 *         <b>false</b> otherwise  
 */  
public boolean isLastSubTutorial() {  
    return subTutorialIndex == (numSubTutorials - 1);  
}  
  
/*  
 * public static void main(String[] args) { Cube cube = new Cube(); Tutorial  
 * t = new Tutorial(cube); t.loadTutorial("CrossTutorial.txt");  
 *  
 * System.out.println("Description: " + t.getDescription());  
 *  
 * if (t.requiresUserAction()) { System.out.println("Hint: " +  
 * t.getNextHint()); System.out.println("Optimal Solution: " +  
 * t.getOptimalSolutions()[0]); System.out.println("Explanation:\n" +  
 * t.getExplanation()); System.out.println("Optimal Solution Length: " +  
 * t.getOptimalSolutionLength());  
 *  
 * if (t.criteriaFilled()) System.out.println("All Criteria Filled!");  
 *  
 * else System.out.println("Not all criteria met"); } }  
*/  
}
```

## Procedure List

## Procedure List

### Class – Algorithm

**public int getAlgorithmID()**

Returns:  
an integer representing the ID of the algorithm.

**public void setAlgorithmID(int algorithmID)**

Parameters:  
algorithmID – an integer representing the ID of the algorithm.

**public java.lang.String getMoveSequence()**

Returns:  
a string containing the moves of the algorithm.

**public void setMoveSequence(java.lang.String moveSequence)**

Parameters:  
moveSequence – a string containing the moves of the algorithm.

**public java.lang.String getComment()**

Returns:  
a string containing the comment for the algorithm.

**public void setComment(java.lang.String comment)**

Parameters:  
comment – a string containing the comment for the algorithm.

### Class – AlgorithmDatabaseConnection

**public static Algorithm[] executeQuery(java.lang.String query) throws java.sql.SQLException, java.lang.ClassNotFoundException**

The try/catch block is invoked if the table does not exist or the query is invalid.

Parameters:  
query - the SQLite query to be performed on 'algorithm' table.

Returns:  
an array of Algorithms representing the result of the specified query.

Throws:  
java.sql.SQLException - if the query is invalid.  
java.lang.ClassNotFoundException - if SQLite classes are missing.

**private static Algorithm[] executeSafeQuery(java.lang.String query) throws java.lang.ClassNotFoundException, java.sql.SQLException**

This method will only be called once the table exists

Parameters:  
query - the SQLite query to be performed on 'algorithm' table.

Returns:  
an array of Algorithms representing the result of the specified query.

Throws:  
java.lang.ClassNotFoundException - if SQLite classes are missing.  
java.sql.SQLException - if the table does not exist or the query is invalid.

**public static void executeUpdate(java.lang.String update) throws java.lang.ClassNotFoundException, java.sql.SQLException**

Executes the specified update on the 'algorithm' table.

Parameters:  
update - the update to be performed on the table.

Throws:

`java.lang.ClassNotFoundException` - if SQLite classes are missing.  
`java.sql.SQLException` - if the query is invalid.

**private static void initTable()  
throws java.lang.ClassNotFoundException**

Initialises the table in the database. This method will be called if the `executeQuery(...)` method cannot find the table in the database.

Throws:

`java.lang.ClassNotFoundException` - if SQLite classes are missing.

**public static void resetIDs()  
throws java.lang.ClassNotFoundException,  
java.sql.SQLException**

Resets the IDs in the 'algorithm' table so that they are continuous, e.g. if the IDs were 1, 2, 5, 6, 7, 12, 13 then they would be reset to 1, 2, 3, 4, 5, 6, 7.

Throws:

`java.lang.ClassNotFoundException` - if SQLite classes are missing.

`java.sql.SQLException` - if the table does not exist etc.

## Class – AlgorithmDatabasePopUp

**private void populateCellsWithDatabaseData()**

Retrieves the data from the 'algorithm' table and adds it to the table in the window

**private void addRow() throws java.lang.ClassNotFoundException,  
java.sql.SQLException**

Adds a row to the table and to the database

Throws:

`java.lang.ClassNotFoundException` - if SQLite classes are missing.

`java.sql.SQLException` - if the table does not exist etc.

**private void deleteRow()**

**throws java.lang.ClassNotFoundException,  
java.sql.SQLException**

Deletes the selected rows from the table, and deletes the corresponding records from the database.

Throws:

`java.lang.ClassNotFoundException` - if SQLite classes are missing.

`java.sql.SQLException` - if the table does not exist or the updates fail.

## Class – ColorSelection

**public java.awt.Color getSelectedColor()**

Returns:

the selected colour to be painted on the cube in the main window.

**public void mouseClicked(java.awt.event.MouseEvent arg0)**

Specified by:

`mouseClicked` in interface `java.awt.event.MouseListener`

See Also:

`MouseListener.mouseClicked(java.awt.event.MouseEvent)`

**public void mouseEntered(java.awt.event.MouseEvent arg0)**

Specified by:

`mouseEntered` in interface `java.awt.event.MouseListener`

See Also:

`MouseListener.mouseEntered(java.awt.event.MouseEvent)`

**public void mouseExited(java.awt.event.MouseEvent arg0)**

Specified by:

`mouseExited` in interface `java.awt.event.MouseListener`

See Also:

`MouseListener.mouseExited(java.awt.event.MouseEvent)`

**`public void mousePressed(java.awt.event.MouseEvent arg0)`**

Specified by:

`mousePressed` in interface `java.awt.event.MouseListener`

See Also:

`MouseListener.mousePressed(java.awt.event.MouseEvent)`

**`public void mouseReleased(java.awt.event.MouseEvent arg0)`**

Specified by:

`mouseReleased` in interface `java.awt.event.MouseListener`

See Also:

`MouseListener.mouseReleased(java.awt.event.MouseEvent)`

## Class – Competition

**`public int getID()`**

Returns:

the ID of the competition

**`public void setID(int competitionID)`**

Parameters:

`competitionID` - the ID of the competition

**`public java.lang.String getDate()`**

Returns:

the date of the competition

**`public void setDate(java.lang.String date)`**

Parameters:

`date` - the date of the competition

**`public static boolean isValidDate(java.lang.String dateString)`**

Returns a value indicating whether the argument is a valid date for a competition

Parameters:

`dateString` - the date to be analysed

Returns:

true if the argument is valid and is in the format dd/MM/yyyy; false otherwise

**`private static int getNumDaysInMonth(int month, int year)`**

Parameters:

`month` - the month of the date in question

`year` - the year of the date in question

Returns:

the number of days in the month

## Class – CompetitionDatabaseConnection

**`public static Competition[] executeQuery(java.lang.String query)`**

`throws java.sql.SQLException,`

`java.lang.ClassNotFoundException`

The try/catch block is invoked if the table does not exist or the query is invalid

Parameters:

`query` - the SQLite query to be performed on 'competition' table

Returns:

an array of Competitions representing the result of the specified query

Throws:

`java.sql.SQLException` - if the query is invalid

`java.lang.ClassNotFoundException` - if SQLite classes are missing

```
private static Competition[] executeSafeQuery(java.lang.String query)
throws java.lang.ClassNotFoundException,
java.sql.SQLException
```

Parameters:

query - the SQLite query to be performed on 'competition' table

Returns:

an array of Competitions representing the result of the specified query

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the table does not exist or the query is invalid

```
public static void executeUpdate(java.lang.String update)
```

```
throws java.lang.ClassNotFoundException,
```

```
java.sql.SQLException
```

Executes the specified update on the 'competition' table

Parameters:

update - the update to be performed on the table

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the query is invalid

```
private static void initTable()
```

```
throws java.lang.ClassNotFoundException
```

Initialises the table in the database. This method will be called if the executeQuery(...) method cannot find the table in the database.

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

## Class - CompetitionDatabasePopUp

```
private void viewRankingsButtonFunction()
```

Sets up the memberCompetitionDatabasePopUp window so that it is visible and the cells are populated with data relating to the selecting row in the table

```
private void editButtonFunction()
```

Performs the operations required to allow the selected row to be edited and validation to be performed on the data entered

```
private void populateCellsWithDatabaseData()
```

Populates the cells of the table with the data from the database

```
private void addRow()
```

```
throws java.lang.ClassNotFoundException,
```

```
java.sql.SQLException
```

Adds a row to the table and a blank record to the 'competition' table in the database

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the table does not exist etc.

```
private void deleteRow()
```

```
throws java.lang.ClassNotFoundException,
```

```
java.sql.SQLException
```

Deletes the selected rows from competitionTable

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the table does not exist etc.

## Class – Corner

```
public static java.awt.Color[][] getAllInitialStickers()
```

Returns: the INITIAL_CORNERS array
<b>public static java.awt.Color[] getInitialStickers(int index)</b>
Returns the stickers for the i-th corner Parameters: index - the index of the corner whose stickers are to be returned
Returns: the stickers of the specified corner
<b>public void setStickers(java.awt.Color zero, java.awt.Color one, java.awt.Color two)</b>
Sets the three stickers of the Corner Parameters: zero - the first sticker one - the second sticker two - the third sticker
<b>public void twist(int direction)</b>
Changes the saved orientation and stickers accordingly Parameters: direction - the direction in which to twist the corner. 1 if clockwise; -1 if anti-clockwise

## Class – CornerSolver

<b>public void solveFirstLayerCorners()</b>
Solves the corners in the first layer of the cube and records the solution and explanation at the same time
<b>private boolean lastMoveWasU(java.util.LinkedList&lt;java.lang.String&gt; originalMoves)</b>
Parameters: originalMoves - the moves to be examined Returns: true if the last move (after simplification and ignoring rotations) was U; false otherwise
<b>public void solveCorner(int currentIndex)</b>
Solves the specified corner, and records the solution and explanation at the same time Parameters: currentIndex - the index of the corner to be solved
<b>public static boolean isFLCorner(Corner corner)</b>
Parameters: corner - the corner to be analysed Returns: true if the corner belongs to the first layer; false otherwise
<b>public boolean firstLayerCornersSolved()</b>
Returns: true if the first-layer corners are solved; false otherwise
<b>private int getScore(int index)</b>
Parameters: index - the index of the corner to be examined Returns: the number of moves required to solve the specified corner
<b>private int getIndexOfMinScore(int length)</b>
Parameters:

length - the number of unsolved corners remaining

Returns:

the index in the solveCandidates array which represents the corner that requires the fewest moves to solve

## Class – CrossSolver

**public void solveCross()**

Solves the cross of the cube in the main window, and records the solution and explanation at the same time

**private boolean edgesAreOpposite(Edge edgeOne,  
Edge edgeTwo)**

Parameters:

edgeOne - this Edge is compared to the second Edge

edgeTwo - this Edge is compared to the first Edge

Returns:

true if the edges are opposite each other and are on the same face on a solved cube;  
false otherwise

**private boolean sliceContainsCrossEdgeOriented(int sliceIndex,  
int orientation)**

Parameters:

sliceIndex - the index of the slice to be analysed

orientation - the orientation to be found

Returns:

true if the specified slice contains a cross edge with the specified orientation;  
false otherwise

**public static boolean isCrossEdge(Edge edge)**

Parameters:

edge - the edge to be analysed

Returns:

true if the specified edge is a cross edge, i.e. the edge contains a white sticker;  
false otherwise

**private int getExpectedOffset(java.awt.Color relativeSolvedEdge,  
java.awt.Color workingEdge)**

Parameters:

relativeSolvedEdge - the secondary colour of a cross edge that is solved (relatively) to other pieces

workingEdge - the secondary colour of a cross edge that is not solved at all

Returns:

the expected offset between the two cross edges on a solved cube

**private int getIndexOfCrossEdgeInSlice(int sliceIndex, int orientation)**

Parameters:

sliceIndex - the index of the slice whose edges are to be searched

orientation - the orientation of the found cross edge must be of this orientation

Returns:

the index of a cross edge in the specified slice with the specified orientation

**public boolean isCrossSolved()**

Indicates whether or not the cross is solved. Solves AUF at the same time.

Returns:

true if the cross is solved;

false otherwise

## Class – Cube

**private void resetCubieIndices()**

Resets the indices of the cubies to their original state

**public Slice getSlice(int index)**

Parameters:

index - the index of the slice to be returned

Returns:

the indexth slice

**public Slice getSlice(java.lang.String face)**

Parameters:

face - the name of the face which represents the slice to be returned

Returns:

the slice which is associated with the specified face

**private void performAbsoluteMove(java.lang.String m)**

Performs the specified move on the cube

Parameters:

m - the move to be performed in WCA notation

**public void performAbsoluteMoves(java.lang.String moves)**

Performs the specified moves on the cube

Parameters:

moves - the moves to be performed in WCA notation

**public void performMove(char m)**

Performs the specified move on the cube

Parameters:

m - the move to be performed represented by a key on keyboard (i.e. not WCA notation)

**public void rotate(java.lang.String rotation)**

Performs the specified rotation on the cube

Parameters:

rotation - the rotation to be performed - this should one of the following:

{x, x', y, y', z, z'}

**private void rot(java.lang.String rotation)**

Performs the specified rotation on the cube

Parameters:

rotation - the rotation to be performed - must be "x" or "y" without "2"s

**private void performRotationMaintenance(java.lang.String rotation, int i)**

Fixes the the cubies after an x rotation

Parameters:

rotation - the rotation to be performed

i - the index of the slice affected

**public void updateCubies(int sliceIndex)**

This method acts as an interface between the cubies of each slice and the all the cubies in this class. This means that when one slice is changed, you can change any corresponding slices using the 'updateAll()' method

Parameters:

sliceIndex - the slice from which the updates should be taken

**public void updateAll()**

Updates the cubies in each slice based on the cubies in this instance of Cube.

**public Edge[] getEdges()**

Returns:

the edges of the cube

**public Corner[] get Corners()**

Returns:

the corners of the cube

<b>public Slice[] getSlices()</b>
Returns: the slices of the cube
<b>public Edge getEdge(int index)</b>
Parameters: index - the index of the edge to be returned
Returns: the indexth edge
<b>public Corner getCorner(int index)</b>
Parameters: index - the index of the corner to be returned
Returns: the indexth corner
<b>public void resetCube()</b>
Resets the cube to its initial solved state with white on top and green on front.

## Class – Cubie

<b>public void setStickers(java.awt.Color[] stickers)</b>
Sets the stickers of the cubie to the stickers parameter
Parameters: stickers - the stickers to be assigned to the cubie
<b>public void setCubieIndex(int cubieIndex)</b>
Sets the cubieIndex of the cubie to the parameter
Parameters: cubieIndex - the new cubie index
<b>public void setOrientation(int orientation)</b>
Sets the orientation of the cubie to the parameter
Parameters: orientation - the value to be assigned to the orientation
<b>public void setSticker(int index, java.awt.Color color)</b>
Sets the indexth sticker of the cubie to color
Parameters: index - the index of the sticker to be changed color - the new color for the indexth sticker
<b>public int getOrientation()</b>
Returns: the orientation of the cubie
<b>public java.awt.Color[] getStickers()</b>
Returns: an array of colors representing the stickers of the cubie
<b>public int getCubieIndex()</b>
Returns: the cubie index associated with this cubie
<b>public int compareTo(Cubie otherCubie)</b>
Specified by: compareTo in interface java.lang.Comparable<Cubie>
Parameters: otherCubie - the cubie to which this cubie is compared
Returns: 0 if this cubie represents the same cubie as otherCubie; -1 otherwise
<b>public int strictCompareTo(Cubie otherCubie)</b>

Parameters:

otherCubie - the cubie to which this cubie is compared

Returns:

0 if the cubies are the same and the stickers are in the same order (but not necessarily positions, e.g. the red-green-white corner will be shown to be the same as the green-white-red corner;  
-1 otherwise

#### **public static java.lang.String getColorToWord(java.awt.Color color)**

Parameters:

color - the colour to be analysed

Returns:

the english word used for the colour;

-1 otherwise

#### **public static java.awt.Color getWordToColor(java.lang.String color)**

Parameters:

color - the english word for the colour to be returned

Returns:

a Color object with the associated characteristics of color

## Class – CustomPdfWriter

#### **public static void createStatisticsPdf(java.lang.String filePath, java.lang.String[] timeData)**

Creates the statistics pdf

Parameters:

filePath - the path to which the pdf file should be saved

timeData - the times from which the statistics are calculated

#### **private static void addMetaData(com.itextpdf.text.Document document)**

Adds meta data to the pdf document

Parameters:

document - the document to which the meta data is to be added

#### **private static void addTimesPage(com.itextpdf.text.Document document, java.lang.String[] timeData)**

**throws com.itextpdf.text.DocumentException**

Adds the specific contents to the document

Parameters:

document - the document to which the contents are added

timeData - the data from which the statistics are calculated

Throws:

com.itextpdf.text.DocumentException - if there was an error writing to the document

#### **private static com.itextpdf.text.pdf.PdfPTable getStatisticsTable(java.lang.String[] timeData)**

**throws com.itextpdf.text.DocumentException**

Parameters:

timeData - the times from which statistics are calculated

Returns:

a formatted table containing the statistics

Throws:

com.itextpdf.text.DocumentException - if there was an error writing to the document

## Class – Edge

#### **public static java.awt.Color[][] getAllInitialStickers()**

Returns:

the initial edges of a solved cube with white on top and green on front
<b>public static java.awt.Color[] getInitialStickers(int index)</b>
Parameters: index - the index of the edge whose stickers to be returned
Returns: the indexth Edge's stickers
<b>public void setStickers(java.awt.Color zero, java.awt.Color one)</b>
Parameters: zero - the first sticker of the Edge one - the second sticker of the Edge
<b>private void flipStickers()</b>
Flip the positions of the stickers so that the first sticker in place of the second sticker and vice-versa.
<b>public void flip()</b>
Flips the edge so that its appearance (stickers) and properties (orientation) change
<b>public void flipOrientation()</b>
Change the orientation in the following mapping: 0 -> 1 1 -> 0
<b>public java.awt.Color getSecondaryColor()</b>
Returns the secondary colour of an edge, i.e. the colour that is not white or yellow. If the edge does not contain white or yellow, then the first sticker will be returned.
Returns: secondary colour of the edge if the Edge has a yellow or white sticker; first sticker otherwise

## Class – EdgeSolver

<b>public void solveMiddleLayerEdges()</b>
Solves the edges in the middle layer, and records the solution and explanation at the same time.
<b>public void solveEdge(int currentIndex)</b>
Solves the Edge at the specified index, and record the solution and explanation at the same time.
Parameters: currentIndex - the index of the Edge to be solved.
<b>private boolean edgeInSetupPosition(Edge edge)</b>
Parameters: edge - the edge to be analysed
Returns: true if the specified Edge is in the correct setup position for solving; false otherwise
<b>public static boolean isMLEdge(Edge edge)</b>
Parameters: edge - the edge to be analysed
Returns: true if the specified edge belongs in the middle layer, i.e. it does not have a yellow or white sticker false otherwise
<b>public boolean middleLayerEdgesSolved()</b>
Returns: true if the Edges in the middle layer are solved; false otherwise

**`private int getScore(int index)`**

This determines how many moves are required to solve the edge at the specified index.

Parameters:

index - the index of Edge to be analysed

Returns:

the number of moves required to solve the Edge at the specified index

**`private int getIndexOfMinScore(int length)`**

Returns the index of the element in the solveCandidates array that requires the fewest number of moves to solve

Parameters:

length - the number of remaining unsolved edges

Returns:

the index of the element in solveCandidates that requires the fewest number of moves to solve

## Class – LinearSearch

**`public static int linearSearchCornerOrientation(java.awt.Color[] list, java.awt.Color element)`**

Searches an array of Colors for a specified Color

Parameters:

list - the list to be searched

element - the element to be found

Returns:

the index of element in list.

If the index is 2, then -1 is returned.

If the element cannot be found then -2 is returned.

**`public static int linearSearch(java.awt.Color[] list, java.awt.Color element)`**

Searches an array of Colors for a specified Color

Parameters:

list - the list to be searched

element - the element to be found

Returns:

the index of element in list if the element can be found;

-1 otherwise

**`public static int linearSearch(java.lang.String[] list, java.lang.String element)`**

Searches an array of Strings for a specified String

Parameters:

list - the list to be searched

element - the element to be found

Returns:

the index of element in list if the element can be found;

-1 otherwise

**`public static int linearSearch(java.util.LinkedList<java.lang.String> list, java.lang.String element)`**

Searches a linked list of strings for a specified element

Parameters:

list - the list to be searched

element - the element to be found

Returns:

the index of element in list

**`public static int linearSearchStartsWith(java.lang.String[] list,`**

**`java.lang.String element)`**

Searches an array of Strings for a String that starts with the specified element, e.g. "Hello" starts with "He"

Parameters:

list - the list to be searched

element - the element to be found

Returns:

the index of element in list

**`public static boolean linearSearchContains(java.lang.Integer[] list, java.lang.Integer element)`**

Searches an array of Integers to see if it contains a specified element (index is not important)

Parameters:

list - the list to be searched

element - to be found

Returns:

true if the list contains the specified element;

false otherwise

**Class – Main****`public static javax.swing.ImageIcon createIcon(java.lang.String path)`**

Returns an ImageIcon, or null if the path was invalid.

Parameters:

path - the path of the icon

Returns:

the ImageIcon to be used

**`public static java.awt.Image createImage(java.lang.String path)`**

Returns an Image, or null if the path was invalid.

Parameters:

path - the path of the icon

Returns:

the Image to be used

**`public static void resizeColumnWidths(javax.swing.JTable table)`**

Resizes the widths of the columns in the specified table so that there is maximum visibility in each column

Parameters:

table - the table whose columns need resized

**`private static org.jfree.data.category.DefaultCategoryDataset getDatasetOfSelectedTimes()`**

Returns the dataset to be used by timeGraph

Returns:

the dataset of the past numTimesToGraph times

**`public static boolean isValidCubeState(boolean clearStickers)`**

Determines whether or not the cube is in a valid state

Parameters:

clearStickers - if true, then when an invalid piece is found, its stickers will be cleared, i.e. turn grey;

if false then the stickers will remain as they are

Returns:

true if the cube is in a valid state;

false otherwise

**`private static int getValidCubieIndex(Cubie cubie, Cubie[] validCubies)`**

Returns the index of cubie in validCubies
Parameters:
cubie - the cubie to find
validCubies - the array in which cubie will be found
Returns:
the index of cubie in validCubies.
If the cubie cannot be found, then -1 will be returned
<b>public static void assignOrientationsToCubies()</b>
Assigns the corresponding orientation to each cubie on the cube. This method is used to find the orientation of each piece after, e.g. painting a custom state or loading a state from file
<b>public void keyReleased(java.awt.event.KeyEvent e)</b>
This method handles the release of spacebar or the solving of the cube after a key is released.
<b>public void keyTyped(java.awt.event.KeyEvent e)</b>
Specified by:
keyTyped in interface java.awt.event.KeyListener
<b>private void assignFaceletPaintingColors()</b>
Assigns the sticker colours to the appropriate elements of faceColors
<b>public void paintComponent(java.awt.Graphics g)</b>
Overrides:
paintComponent in class javax.swing.JComponent
<b>public static void setRandomScrambleEnabled(boolean state)</b>
Sets whether or not the 'Apply Random Scramble' menu item is enabled.
Parameters:
state - the resulting enabled/disabled state of the menu item
<b>public static void addSolveToList(Solve solve)</b>
Records the specified list in solves and appends the time to the display-list
Parameters:
solve - the solve to be added
<b>public static void performRealTimeSolving(java.lang.String scramble, java.lang.String solution)</b>
Scrambles the cube (instantly) with the specified scramble and performs the solution in real-time as an animation at the speed defined in Preferences
Parameters:
scramble - the scramble to be applied before the animation starts
solution - the moves to be performed in real-time
<b>public static void performRealTimeSolving()</b>
This solves the cube and shows the moves as an animation (real-time) at the speed specified in Preferences
<b>private static void cancelSolve()</b>
Cancels all animations, timers, etc., and labels reset to their default values.
<b>private static void endSolve()</b>
Performs the operations required to end the solve in the appropriate way. After the user completes a solve after clicking the 'Start New Solve' button, this method will be invoked and the time will be recorded. But in other situations, the time will not be recorded.
<b>public static java.lang.String getFormattedCubeSolution()</b>
Returns:
the solution to the cube state in a formatted fashion
<b>public static void refreshTimeList()</b>
Refreshes the list of times after a time is added, edited or deleted.
<b>public static void copyAllTimesToDisplay()</b>
Copies all times from solves to timeDisplayList

<b>public static void refreshStatistics()</b>
Refreshes the statistics and resets the scrolling position to the top of the text area
<b>public static void refreshTimeGraph(boolean cubePanelFocus)</b>
Refreshes the dataset being drawn in the Time Graph window.
Parameters:
cubePanelFocus - if true, then the cube panel should be focused after method executes; otherwise the existing focus-window should keep focus
<b>public static void initListPanel()</b>
Initialises the components relating to the list of times at the right-hand side of the main window.
<b>public static boolean isCubeSolved()</b>
Returns:
true if the cube is solved;
false otherwise
<b>public static void requestCubePanelFocus()</b>
Requests focus for cubePanel
<b>private static void resetCube()</b>
Resets the cube to the solve state with white on top and green on front
<b>public static void setScrambleTextSize(int size)</b>
Sets the size of the scramble text to the specified size
Parameters:
size - the size for the text of the scramble
<b>public static void handleScramble(java.lang.String scramble)</b>
Scrambles the cube with the specified scramble and shows the scramble in scrambleLabel
Parameters:
scramble - the scramble to be applied
<b>public static void clearAllSolverMoves()</b>
Clears all moves stored by the children of solveMaster
<b>private static java.awt.event.MouseListener getCubePanelMouseListener()</b>
Returns:
a MouseListener for the cube panel so that the clicked piece can be solved when in Click-to-Solve mode
<b>public static void createAndShowGUI()</b>
Sets up the main window
<b>public static void main(java.lang.String[] s)</b>
Runs the program
Parameters:
s - runtime argument

## Class – Main.InspectionTimer

<b>public void actionPerformed(java.awt.event.ActionEvent e)</b>
Specified by:
actionPerformed in interface java.awt.event.ActionListener
See Also:

## Class – Main.MenuBar

<b>public javax.swing.JMenuBar createMenuBar()</b>
Returns:
the menu bar for the main window
<b>private void setUpTutorial()</b>
Switches the main window from timing mode to tutorial mode. The text area at the bottom of

the window displays the correct information.

#### **private void switchToTutorialView()**

Switches the main window from timing mode to tutorial mode. The buttons at the bottom of the window are changed and the appropriate fields are changed. Menu items are enabled and disabled accordingly

#### **private static void switchToTimingView()**

Switches the main window from tutorial mode to timing mode. The buttons at the bottom of the window are changed and the appropriate fields are changed. Menu items are enabled and disabled accordingly

#### **public static void setSolvePieceSelected(boolean state)**

Determines whether clickToSolveItem is enabled or not

Parameters:

state - the resulting state for the clickToSolve item

#### **public static void setRandomScrambleEnabled(boolean state)**

Determines whether applyRandomScrambleItem is enabled or not

Parameters:

state - the resulting state for the applyRandomScramble item

#### **public static boolean isUsingScramblesInList()**

Returns:

true if useScramblesInListItem is selected;

false otherwise

## Class – Main.PopUpWindowListener

#### **public void windowActivated(java.awt.event.WindowEvent arg0)**

Specified by:

windowActivated in interface java.awt.event.WindowListener

#### **public void windowClosed(java.awt.event.WindowEvent arg0)**

Specified by:

windowClosed in interface java.awt.event.WindowListener

#### **public void windowClosing(java.awt.event.WindowEvent arg0)**

Specified by:

windowActivated in interface java.awt.event.WindowListener

#### **public void windowDeactivated(java.awt.event.WindowEvent arg0)**

Updates the times and statistics

#### **public void windowDeiconified(java.awt.event.WindowEvent arg0)**

Specified by:

windowDeiconified in interface java.awt.event.WindowListener

#### **public void windowIconified(java.awt.event.WindowEvent arg0)**

Specified by:

windowIconified in interface java.awt.event.WindowListener

#### **public void windowOpened(java.awt.event.WindowEvent arg0)**

Specified by:

windowOpened in interface java.awt.event.WindowListener

## Class – Main.RealTimeTimerListener

#### **public void actionPerformed(java.awt.event.ActionEvent e)**

Specified by:

actionPerformed in interface java.awt.event.ActionListener

## Class – Main.TimerListener

#### **public void actionPerformed(java.awt.event.ActionEvent evt)**

Specified by:  
actionPerformed in interface java.awt.event.ActionListener

## Class – Member

**public int getMemberID()**

Returns:  
the ID of the member

**public void setMemberID(int memberID)**

Assigns the specified ID to the member

Parameters:  
memberID - the new ID to be assigned to the member

**public java.lang.String getForenames()**

Returns:  
the forenames of the member

**public void setForenames(java.lang.String forenames)**

Assigns the specified forenames to the member

Parameters:  
forenames - the new forenames to be assigned to the member

**public java.lang.String getSurname()**

Returns:  
the surname of the member

**public void setSurname(java.lang.String surname)**

Assigns the specified surname to the member

Parameters:  
surname - the new surname to be assigned to the member

**public java.lang.String getGender()**

Returns:  
the gender of the member - "male" or "female"

**public void setGender(java.lang.String gender)**

Assigns the specified gender to the member

Parameters:  
gender - the new gender to be assigned to the member - "male" or "female"

**public java.lang.String getDateOfBirth()**

Returns:  
the date of birth of the member

**public void setDateOfBirth(java.lang.String dateOfBirth)**

Assigns the specified date of birth to the member

Parameters:  
dateOfBirth - the new date of birth for the member

**public java.lang.String getEmail()**

Returns:  
the email address of the member

**public void setEmail(java.lang.String email)**

Assigns the specified email to the member

Parameters:  
email - the new email address to be assigned to the member

**public java.lang.String getFormClass()**

Returns:  
the form class to which the member belongs

**public void setFormClass(java.lang.String formClass)**

Assigns the specified form class to the member

Parameters:

formClass - the new form class to be assigned to the member

#### **public static boolean isValidEmail(java.lang.String email)**

Determines whether the specified email is in a valid format or not. This does not verify that the address exists

Parameters:

email - the email to be analysed

Returns:

true if the email address is valid;

false otherwise

#### **public static boolean isValidFormClass(java.lang.String formClass)**

Determines whether the specified form class is valid or not. Valid form classes are of the form:

08 | M

09 | R

10 | S

11 | T

12 | W

13 |

14 |

Leading zeros are ignored

Parameters:

formClass - the form class to be analysed

Returns:

true if the form class is valid;

false otherwise

## Class – MemberCompetition

#### **public int getCompetitionID()**

Returns:

the ID of the competition

#### **public void setCompetitionID(int competitionID)**

Assigns an ID to the competition

Parameters:

competitionID - the new ID of the competition

#### **public int getMemberID()**

Returns:

the ID of the member

#### **public void setMemberID(int memberID)**

Assigns an ID to the member

Parameters:

memberID - the ID of the member

#### **public java.lang.String[] getTimes()**

Returns:

an array of Strings representing the times of the average

#### **public void setTimes(java.lang.String[] times)**

Assigns the times of the average

Parameters:

times - the times of the average

#### **public double getAverage()**

Returns:

the calculated WCA average of the 5 times

#### **public double get2DPAverage()**

Returns:

the calculated WCA average of the 5 times to 2 decimal places

**public double[] getNumericTimeArray()**

Returns:

the 5 times of the average in a numerical representation

**public boolean isBetterThan(MemberCompetition other)**

This method compares this average to another average. An average is better than another average if the 'average of 5' is faster, or the fastest time of this is faster than the fastest time of other. If the fastest times are the same, then the second fastest times are compared in the same way. If all times are the same, then other will be assumed to be the better average

Parameters:

other - the other MemberCompetition to which this MemberCompetition is compared

Returns:

true if this is better than other;

false otherwise

## Class – MemberCompetitionDatabaseConnection

**public static MemberCompetition[] executeQuery(java.lang.String query)**

throws java.sql.SQLException,

**java.lang.ClassNotFoundException**

The try/catch block is invoked if the table does not exist or the query is invalid

Parameters:

query - the SQLite query to be performed on 'memberCompetition' table

Returns:

an array of MemberCompetitions representing the result of the specified query

Throws:

java.sql.SQLException - if the query is invalid

java.lang.ClassNotFoundException - if SQLite classes are missing

**private static MemberCompetition[] executeSafeQuery(java.lang.String query)**

throws java.lang.ClassNotFoundException,

**java.sql.SQLException**

Parameters:

query - the SQLite query to be performed on 'memberCompetition' table

Returns:

an array of MemberCompetitions representing the result of the specified query

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the table does not exist or the query is invalid

**public static void executeUpdate(java.lang.String update)**

throws java.lang.ClassNotFoundException,

**java.sql.SQLException**

Executes the specified update on the 'memberCompetition' table

Parameters:

update - the update to be performed on the table

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the query is invalid

**private static void initTable()**

throws java.lang.ClassNotFoundException

Initialises the table in the database. This method will be called if the executeQuery(...)

method cannot find the table in the database.

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

## Class – MemberCompetitionDatabasePopUp

### **private void editRecordFunction()**

Performs the operations required to set up the 'Member-Competition Form' window with the contents of the selected row in the table and indicates that the row is being edited (and not added)

### **private boolean memberIDComboBoxContains(java.lang.Integer item)**

Determines whether memberIDComboBox contains the specified item

Parameters:

item - the item to be found

Returns:

true if the item is in memberIDComboBox;

false otherwise

### **public void populateCellsWithData()**

Retrieves the data from the 'memberCompetition' table and adds it to the table in the window

### **private void addRow()**

throws java.lang.ClassNotFoundException,

java.sql.SQLException

Adds a row to the table and to the database

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the table does not exist etc.

### **private void deleteRow()**

throws java.lang.ClassNotFoundException,

java.sql.SQLException

Deletes the selected rows from the table, and deletes the corresponding records from the database

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the table does not exist or the updates fail

### **public void setCurrentCompetitionID(int competitionID)**

Sets the competition ID to competitionID so that the correct records are retrieved from the database

Parameters:

competitionID - the ID of the competition will be set to this value

### **private void setUpMemberCompetitionInputForm()**

Sets up the 'Member-Competition Form' window

### **private void resetMemberIDComboBoxItems()**

This resets the items in the drop-down list in the 'Member-Competition Form' window so that it contains the correct items

### **public void keyPressed(java.awt.event.KeyEvent arg0)**

Specified by:

keyPressed in interface java.awt.event.KeyListener

### **public void keyReleased(java.awt.event.KeyEvent arg0)**

Specified by:

keyReleased in interface java.awt.event.KeyListener

### **public void keyTyped(java.awt.event.KeyEvent arg0)**

Specified by:

keyTyped in interface java.awt.event.KeyListener

## Class – MemberDatabaseConnection

### **public static Member[] executeQuery(java.lang.String query)**

```
throws java.sql.SQLException,
java.lang.ClassNotFoundException
```

The try/catch block is invoked if the table does not exist or the query is invalid

Parameters:

query - the SQLite query to be performed on 'member' table

Returns:

an array of Members representing the result of the specified query

Throws:

java.sql.SQLException - if the query is invalid

java.lang.ClassNotFoundException - if SQLite classes are missing

```
private static Member[] executeSafeQuery(java.lang.String query)
```

```
throws java.lang.ClassNotFoundException,
```

```
java.sql.SQLException
```

This method will only be called once the table exists

Parameters:

query - the SQLite query to be performed on 'Member' table

Returns:

an array of Members representing the result of the specified query

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the table does not exist or the query is invalid

```
public static void executeUpdate(java.lang.String update)
```

```
throws java.lang.ClassNotFoundException,
```

```
java.sql.SQLException
```

Executes the specified update on the 'member' table

Parameters:

update - the update to be performed on the table

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the query is invalid

```
private static void initTable()
```

```
throws java.lang.ClassNotFoundException
```

Initialises the table in the database. This method will be called if the executeQuery(...)

method cannot find the table in the database.

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

```
public static void resetIDs()
```

```
throws java.lang.ClassNotFoundException,
```

```
java.sql.SQLException
```

Resets the IDs in the 'member' table so that they are continuous, e.g. if the IDs were 1, 2, 5, 6, 7, 12, 13 then they would be reset to 1, 2, 3, 4, 5, 6, 7

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the table does not exist etc.

## Class – MemberDatabasePopUp

```
private void editMemberButtonFunction()
```

Performs the operations required to set up the 'Member Form' window with the contents of the selected row in the table and indicates that the row is being edited (and not added)

```
private void populateCellsWithDatabaseData()
```

Retrieves the data from the 'member' table and adds it to the table in the window

```
private void addRow()
```

```
throws java.lang.ClassNotFoundException,
```

**java.sql.SQLException**

Adds a row to the table and to the database

Throws:

`java.lang.ClassNotFoundException` - if SQLite classes are missing

`java.sql.SQLException` - if the table does not exist etc.

**private void deleteRow()**

`throws java.lang.ClassNotFoundException,`

`java.sql.SQLException`

Deletes the selected rows from the table, and deletes the corresponding records from the database

Throws:

`java.lang.ClassNotFoundException` - if SQLite classes are missing

`java.sql.SQLException` - if the table does not exist or the updates fail

**private static boolean isValidDate(java.lang.String dateString)**

Returns a value indicating whether the argument is a valid date for a date of birth

Parameters:

`dateString` - the date to be analysed

Returns:

true if the argument is valid and is in the format dd/MM/yyyy; false otherwise

**private static int getNumDaysInMonth(int month, int year)**

Parameters:

`month` - the month of the date in question

`year` - the year of the date in question

Returns:

the number of days in the month

**private void setUpMemberInputForm()**

Sets up the 'Member Form' window

**public void keyPressed(java.awt.event.KeyEvent arg0)**

Specified by:

`keyPressed` in interface `java.awt.event.KeyListener`

**public void keyReleased(java.awt.event.KeyEvent arg0)**

Specified by:

`keyReleased` in interface `java.awt.event.KeyListener`

**public void keyTyped(java.awt.event.KeyEvent arg0)**

Specified by:

`keyTyped` in interface `java.awt.event.KeyListener`

**Class – MouseSelectionSolver****public java.lang.String getSolution()**

Returns:

the generated solution

**public void solvePiece(int index)**

Solves the piece at the specified index, and records the solution and explanation simultaneously.

Parameters:

`index` - the index of the piece to be solved

**public static int getIndexOfPieceOnScreen(int x, int y)**

Returns the index of the piece on screen with the coordinates (x, y).

Parameters:

`x` - the x coordinate of the piece on screen

y - the y coordinate of the piece on screen

Returns:

Corners: 0, 1, 2, ..., 7

Edges: 8, 9, 10, ..., 19

-2 if the selection is invalid

#### **public static int getIndexOfFaceletOnScreen(int x, int y)**

Returns the index of the facelet/sticker on screen with the coordinates (x, y).

Parameters:

x - the x coordinate of the facelet/sticker on screen

y - the y coordinate of the facelet/sticker on screen

Returns:

index of facelet/sticker on screen

-2 if the selection is invalid

#### **public static int getQuestionDialogResponse(java.lang.String message)**

Shows a question dialog and gets a decision from a user

Parameters:

message - the text to be shown in the dialog

Returns:

0 if 'Yes' is selected;

1 if 'No' is selected;

-1 if nothing is selected

## Class – OrientationSolver

#### **public void solveOrientation()**

Performs and records the moves required to solve the orientation of cube

#### **private void solveEdgeOrientation()**

Performs and records the moves required to solve the edge orientation of cube

#### **private void solveCornerOrientation()**

Performs the moves required to solve the corner orientation of cube

#### **private int getNumOriented()**

Returns:

the number of edges in the top layer are oriented correctly

#### **private boolean isEdgeOriented(int edgeIndex)**

Determines whether the edge at the specified index is oriented correctly

Parameters:

edgeIndex - the index of the Edge to be analysed

Returns:

true if the edge at index is oriented correctly;

false otherwise

#### **private boolean isCornerOriented(int cornerIndex)**

Determines whether the corner at the specified index is oriented correctly

Parameters:

cornerIndex - the index of the Corner to be analysed

Returns:

true if the Corner at index is oriented correctly;

false otherwise

#### **public boolean isOrientationSolved()**

Returns:

true if all pieces in the top layer are oriented correctly;

false otherwise

#### **public boolean isEdgeOrientationSolved()**

Returns:

true if all Edges are oriented correctly;

false otherwise

#### **public boolean isCornerOrientationSolved()**

Returns:

true if all Corners are oriented correctly;

false otherwise

## Class – PermutationSolver

#### **public void solvePermutation()**

Performs and records the moves required to solve the permutation of cube

#### **private void performAUF()**

After solving the relative permutation of the pieces, this method performs the move "U" until the cube is solved.

#### **private void solveCornerPermutation()**

Performs and records the moves required to solve the corner permutation of cube

#### **private void solveEdgePermutation()**

Performs and records the moves required to solve the edge permutation of cube

#### **public boolean isEdgePermutationSolved()**

Returns:

true if the permutation of the Edges of the last layer is solved;

false otherwise

#### **private boolean edgesAreOpposite(Edge edgeOne, Edge edgeTwo)**

Parameters:

edgeOne - this Edge is compared to the second Edge

edgeTwo - this Edge is compared to the first Edge

Returns:

true if the edges are opposite each other and are on the same face on a solved cube;

false otherwise

#### **public boolean isCornerPermutationSolved()**

Returns:

true if the permutation of the Corners of the last layer is solved;

false otherwise

#### **private boolean headlightsAtBack()**

Returns:

true if there are 'headlights' at the back face of the cube, i.e. two matching Corner stickers;

false otherwise

#### **public boolean permutationSolved()**

Returns:

true if the permutation of all pieces in the top layer is solved;

false otherwise

## Class – Preferences

#### **public int getRealTimeSolvingRate()**

Returns:

the real-time solving speed in milliseconds

#### **public int getInspectionTime()**

Returns:

the time allowed to inspect the cube before a solve starts (seconds)

#### **public boolean solvePieceWarningEnabled()**

Returns:

true if the solvePieceWarningYesItem is selected;

false otherwise

#### **public int getScrambleTextSize()**

Returns: the scramble text size as saved in the 'Preferences' window
<b>public void actionPerformed(java.awt.event.ActionEvent arg0)</b>
This method is called when the enter key is pressed when typing data into one of the text fields.
<b>public void populateFieldsWithValues()</b>
Retrieves the data from preferences.txt and inserts it into the text fields
<b>private void restoreDefaults()</b>
Restores and saves the system's preferences to its default settings
<b>private void savePreferences()</b>
Writes the preferences to preferences.txt

## Class - Scramble

<b>public static java.lang.String generateScramble()</b>
Returns: a randomly generated 25-move scramble
<b>private static java.lang.String getRandomDirection()</b>
Returns: a random element from the directions array
<b>private static java.lang.String getRandomFace(java.lang.String penultimate, java.lang.String last)</b>
Parameters: penultimate - the penultimate move in the scramble last - the last move in the scramble
Returns: a random face whose direction of motion is in the same axis compared with the penultimate and last moves. For example, R L F would be an acceptable series of moves because the directions of motion of these moves is x x z. U D U' would not be acceptable because the directions of motion of these moves is y y y, which are all the same so a new move must be generated.
<b>private static boolean isSameAxis(java.lang.String a, java.lang.String b, java.lang.String c)</b>
Parameters: a - the first move with its associated rotation direction b - the second move with its associated rotation direction c - the third move with its associated rotation direction
Returns: true if the three moves are in the same axis; false otherwise

## Class – ScramblePopUp

<b>private void initListPanel()</b>
Initialises the components associated with the list of scrambles in the window. Fonts, selection modes, and contents are declared here.
<b>public java.lang.String getCurrentScramble() throws java.lang.IndexOutOfBoundsException</b>
Returns: the next scramble in the list
Throws: java.lang.IndexOutOfBoundsException - if there are no elements in the list

## Class – Slice

**public void setCubieIndices(int[] cubieIndices)**

Sets the cubie indices of the pieces

Parameters:

cubieIndices - the cubie indices to be assigned to the pieces. The first four elements should be for the Corners and the next four should be for the Edges.

**public void setEdges(Edge[] edges)**

Sets the Edges of the slice to the specified Edges

Parameters:

edges - the Edges to be assigned to the slice

**public void setCorners(Corner[] corners)**

Sets the Corners of the slice to the specified Corners

Parameters:

corners - the Corners to be assigned to the slice

**public void setEdge(int index, Edge edge)**

Sets the Edge at the specified index to the specified Edge

Parameters:

index - the index of the Edge to be changed

edge - the Edge to be assigned to the specified index

**public void setCorner(int index, Corner corner)**

Sets the Corner at the specified index to the specified Corner

Parameters:

index - the index of the Corner to be changed

corner - the Corner to be assigned to the specified index

**public void setCentre(java.awt.Color centre)**

Sets the centre to the specified Color

Parameters:

centre - the Color to be set as the centre

**public Edge[] getEdges()**

Returns:

an array containing the Edges of the slice

**public Corner[] getCornes()**

Returns:

an array containing the Corners of the slice

**public Edge getEdge(int index)**

Parameters:

index - the index of the Edge to be returned

Returns:

the indexth Edge

**public Corner getCorner(int index)**

Parameters:

index - the index of the Corner to be returned

Returns:

the indexth Corner

**public java.awt.Color getCentre()**

Returns:

the centre Color

**public int[] getCubieIndices()**

Returns:

an array containing the indices for all cubies

**public void performMove(int direction)**

Performs a move on the slice in the specified direction  
 1 - Clockwise 2 - 180 degree -1 - Anticlockwise This method handles an argument of '2'

Parameters:

direction - the direction in which to move the slice

#### **private void pMove(int direction)**

Performs a move on the slice in the specified direction

1 - Clockwise -1 - Anticlockwise

Parameters:

direction - the direction in which to move the slice

#### **public void flipAllEdges()**

Flips all edges so that the stickers and saved orientation changes for each edge

#### **public void flipEdge(int index)**

Flips the stickers and orientation of the Edge at the specified index

Parameters:

index - the index of the Edge to be flipped

#### **public void twistAllCorners(int direction)**

This method should be called after performing a move so that the corners are twisted correctly.

Parameters:

direction - the direction of the move that was performed.

#### **public void twistCorner(int index, int direction)**

Twists (rotates stickers and changes orientation) the Corner at the specified index in the specified direction

Parameters:

index - the index of the Corner to be twisted

direction - the direction in which to rotate the Corner

#### **public void swapEdges(int i, int j)**

Swaps the Edges at the specified indices

Parameters:

i - the index of the first Edge to be swapped.

j - the index of the second Edge to be swapped.

#### **public void swapCorners(int i, int j)**

Swaps the Corners at the specified indices

Parameters:

i - the index of the first Corner to be swapped.

j - the index of the second Corner to be swapped.

#### **public boolean contains(Cubie cubie)**

Parameters:

cubie - the cubie to be searched for

Returns:

true if the slice contains the specified Cubie;

false otherwise

## Class – Solve

#### **public void setStringTime(java.lang.String time)**

Sets the time of the solve.

Parameters:

time - the time of the solve in string format and formatted representation, e.g. the input should be 1:02.00 rather than 62.0

#### **public void setNumericTime(double time)**

Sets the time of the solve

Parameters:

time - the time of the solve as a real number
<b>public void setPenalty(java.lang.String penalty)</b>
Sets the penalty of the solve
Parameters:
penalty - the penalty of the solve
<b>public void setComment(java.lang.String comment)</b>
Sets the comment for the solve
Parameters:
comment - the comment for the solve
<b>public void setScramble(java.lang.String scramble)</b>
Parameters:
scramble - the scramble for the solve
<b>public void setSolution(java.lang.String solution)</b>
Parameters:
solution - the solution for the solve
<b>public void setMoveCount(int moveCount)</b>
Parameters:
moveCount - the move count for the solve
<b>public java.lang.String getStringTime()</b>
Returns:
the time in a string representation
<b>public double getNumericTime()</b>
Returns:
a numeric representation of the solve's time
<b>public java.lang.String getPenalty()</b>
Returns:
the penalty for the solve
<b>public java.lang.String getComment()</b>
Returns:
the comment for the solve
<b>public java.lang.String getScramble()</b>
Returns:
the scramble for the solve
<b>public java.lang.String getSolution()</b>
Returns:
the solution for the solve
<b>public int getMoveCount()</b>
Returns:
the move count for the solve
<b>public static boolean isValidTime(java.lang.String time)</b>
Determines whether the specified time is in a valid. All valid formats are: MM:SS.sss MM:SS. MM:Ssss MM:S. M:SS.sss M:SS. M:Ssss M:S. SS.sss SS. Ssss S. DNF

Parameters:

time - the time to be analysed

Returns:

true if the specified time is valid;

false otherwise

**public static java.lang.String getPaddedTime(java.lang.String time)**

Pads a time-string with leading and trailing zeros where appropriate. The argument must be in the form X:M.ss or similar so that it can be padded correctly.

Parameters:

time - the time-string to be padded

Returns:

the padded time-string

**public static double getFormattedStringToDouble(java.lang.String time)**

Returns the numerical value represented by the formatted time-string

Parameters:

time - the formatted time-string from which the numerical time is to be calculated

Returns:

the numerical value represented by time

**public static java.lang.String getSecondsToFormattedString(double seconds)**

Returns a formatted time-string representing the same number of seconds as the argument.

Parameters:

seconds - the number of seconds of the time

Returns:

a formatted time-string representting the same number of seconds as the argument.

**public static int getMoveCount(java.lang.String moves)**

Parameters:

moves - the sequence of moves whose length is to be calculated.

Returns:

the number of moves represented in moves

**public static boolean isValidDate(java.lang.String dateString)**

Parameters:

dateString - the date to be analysed

Returns:

true if the date is valid and in the form yyyy-MM-dd HH:mm:ss

false otherwise

**private static boolean isValidDateCheck(java.lang.String dateString)**

Parameters:

dateString - the date to be analysed

Returns:

true if the date is in the form yyyy-MM-dd and is valid;

false otherwise

**private static int getNumDaysInMonth(int month, int year)**

Parameters:

month - the month whose number of days is to be found

year - the year of the date

Returns:

the number of days in the month

**public static boolean isValidTimeCheck(java.lang.String timeString)**

Parameters:

timeString - the time to be analysed

Returns:

true if the time is valid;

false otherwise
-----------------

## Class – SolveDatabaseConnection

```
public static SolveDBType[] executeQuery(java.lang.String query)
throws java.sql.SQLException,
java.lang.ClassNotFoundException
```

The try/catch block is invoked if the table does not exist or the query is invalid

Parameters:

query - the SQLite query to be performed on 'solve' table

Returns:

an array of Solves representing the result of the specified query

Throws:

java.sql.SQLException - if the query is invalid

java.lang.ClassNotFoundException - if SQLite classes are missing

```
private static SolveDBType[] executeSafeQuery(java.lang.String query)
throws java.lang.ClassNotFoundException,
java.sql.SQLException
```

This method will only be called once the table exists

Parameters:

query - the SQLite query to be performed on 'solve' table

Returns:

an array of Solves representing the result of the specified query

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the table does not exist or the query is invalid

```
public static void executeUpdate(java.lang.String update)
throws java.lang.ClassNotFoundException,
java.sql.SQLException
```

Executes the specified update on the 'solve' table

Parameters:

update - the update to be performed on the table

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

java.sql.SQLException - if the query is invalid

```
private static void initTable()
throws java.lang.ClassNotFoundException
```

Initialises the table in the database. This method will be called if the executeQuery(...) method cannot find the table in the database.

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing

```
public static java.lang.String getCurrentTimeInSQLFormat()
```

Returns:

the current time in the format yyyy-MM-dd HH:mm:ss

## Class – SolveDatabasePopUp

```
private void editSolveButtonFunction()
```

Sets up solveInputForm so that the data from the selected row in the table is placed in its fields. It is also indicated that a record is being edited (rather than added).

```
public void populateCellsWithDatabaseData()
```

Retrieves the data from the 'solve' table and adds it to the table in the window

```
public void populateCellsWithDatabaseData(java.lang.String query)
```

Retrieves the data matching the specified query from the 'solve' table and adds it to the table in the window

Parameters:

query - the query to be used on the database

```
private void addRow()
throws java.lang.ClassNotFoundException,
java.sql.SQLException
```

Adds a row to the table and to the database

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing  
java.sql.SQLException - if the table does not exist etc.

```
private void deleteRow()
throws java.lang.ClassNotFoundException,
java.sql.SQLException
```

Deletes the selected rows from the table, and deletes the corresponding records from the database

Throws:

java.lang.ClassNotFoundException - if SQLite classes are missing  
java.sql.SQLException - if the table does not exist or the updates fail

```
private void resizeColumnWidths(javax.swing.JTable table)
```

Resizes the widths of the columns in the specified table so that there is maximum visibility in each column

Parameters:

table - the table whose columns need resized

```
private void resetTimeBoundariesAndShowErrorMessage(java.lang.String
originalLowerBoundary,
java.lang.String originalUpperBoundary)
```

This method is invoked if the user is trying to filter the data in the table, but enters invalid data. The data will be reset and an error message will be shown.

Parameters:

originalLowerBoundary - the value that will be assigned to the lowerTimeBoundary  
originalUpperBoundary - the value that will be assigned to the upperTimeBoundary

```
private void setUpSolveInputForm()
```

Sets up the Solve Form

```
public void keyPressed(java.awt.event.KeyEvent arg0)
```

Specified by:

keyPressed in interface java.awt.event.KeyListener

```
public void keyReleased(java.awt.event.KeyEvent arg0)
```

Specified by:

keyReleased in interface java.awt.event.KeyListener

```
public void keyTyped(java.awt.event.KeyEvent arg0)
```

Specified by:

keyTyped in interface java.awt.event.KeyListener

## Class – SolveDBType

```
public int getID()
```

Returns:

ID of the solve

```
public java.lang.String getDateAdded()
```

Returns:

the date the solve was added

## Class - SolveMaster

**public java.lang.String getSolutionExplanation()**

Returns:

the explanation of the solution (prose)

**public int getIndexOfCubie(cubie cubie)**

Returns the index of the specified cubie on the cube

Parameters:

cubie - the cubie to be found

Returns:

the index of the cubie on the cube

**public void rotateToTop(java.awt.Color color)**

Performs the rotations required so that the cube shows the specified centre colour on top.

The rotations performed are recorded.

Parameters:

color - the colour which should end up on top

**public void rotateToTopDoNotRecord(java.awt.Color color)**

Performs the rotations required so that the cube shows the specified centre colour on top.

The rotations performed are not recorded.

Parameters:

color - the colour which should end up on top

**public void rotateToTopFront(java.awt.Color top, java.awt.Color front)**

Performs the rotations required so that the specified centre colour end up on top and front.

This will get stuck in an infinite loop if the centre are opposite, e.g. white and yellow. These parameters cannot be given during runtime, so if this gets stuck, there is a problem in code.

The rotations are recorded.

Parameters:

top - the centre to end up on top.

front - the centre to end up on front.

**public void rotateToTopFrontDoNotRecord(java.awt.Color top, java.awt.Color front)**

Performs the rotations required so that the specified centre colour end up on top and front.

This will get stuck in an infinite loop if the centre are opposite, e.g. white and yellow. These parameters cannot be given during runtime, so if this gets stuck, there is a problem in code.

The rotations are not recorded.

Parameters:

top - the centre to end up on top.

front - the centre to end up on front.

**public static void simplifyMoves(java.util.LinkedList<java.lang.String> originalMoves, int type)**

Simplifies the specified moves in the specified way.

Parameters:

originalMoves - the moves to be simplified

type - the type of simplification to be applied

**private static void simplifyCornerEdgeSolution(java.util.LinkedList<java.lang.String> originalMoves)**

Simplifies the specified moves so that cancellations occur,

e.g. R R R = R'

and so that split rotations are grouped,

e.g. y U y U = y2 U2

Parameters:

originalMoves - the moves to be simplified

**private static void simplifyCross(java.util.LinkedList<java.lang.String> originalMoves)**

Simplifies the cross solution so that there are no rotations,

e.g. R2 U y2 R = R2 U L

Parameters:

originalMoves - the moves to be simplified

```
private static java.lang.String applyRotationToMove(java.lang.String rotation,
java.lang.String move)
```

Returns the move that performs the same action on the cube if the cube was not rotated before the move.

Parameters:

rotation - the rotation that would have been made.

move - the move that would have been made after the rotation

Returns:

the move that performs the same action as the specified move without a rotation

```
private static void cancelMoves(java.util.LinkedList<java.lang.String> originalMoves)
```

Cancels moves such as L2 L' -> L.

Parameters:

originalMoves - the moves to simplify

```
private static java.lang.String getCombination(java.lang.String one, java.lang.String two)
```

Parameters:

one - the first direction

two - the second direction

Returns:

the resultant direction, e.g. ("2", "") would return "

```
protected boolean isPieceSolved(Corner corner)
```

Determines whether or not the specified corner is solved. This method uses the actual argument in the further method calls.

Parameters:

corner - the corner to be analysed

Returns:

true if the specified corner is solved;

false otherwise

```
protected boolean newPieceSolved(Cubie cubie)
```

Determines whether or not the specified cubie is solved. This method uses a retrieved cubie in the further method calls rather than the original argument. This is useful if the argument does not actually belong to the same instance of Cube as the field in this class.

Parameters:

cubie - the cubie to be analysed

Returns:

true if the specified cubie is solved;

false otherwise

```
protected boolean isPieceSolved(Edge edge)
```

Determines whether or not the specified edge is solved. This method uses the actual argument in the further method calls.

Parameters:

edge - the edge to be analysed

Returns:

true if the specified edge is solved;

false otherwise

```
private int getSliceIndexOfCentre(java.awt.Color centre)
```

Returns the index of the specified centre colour on the cube.

0 = Top

1 = Bottom

2 = Right

3 = Left 4 = Front 5 = Back Parameters: centre - the centre colour to be found Returns: the index of the slice whose centre is the specified colour
<b>private void catalogMove(java.lang.String move)</b>
Records the specified move Parameters: move - the move to be recorded
<b>public void catalogMoves(java.lang.String moves)</b>
Checks that the argument is not null, then records the moves. Parameters: moves - the moves to be recorded
<b>private void catMoves(java.lang.String moves, int index)</b>
Recursively records the specified moves, one by one. E.g. "R U R' F" will be recorded as "R", "U", "R'", "F" Parameters: moves - the moves to be recorded index - the last index of a space
<b>public java.util.LinkedList&lt;java.lang.String&gt; getCatalogMoves()</b>
Returns: the recorded moves in a linked list of strings
<b>public int getNumMoves(java.lang.String[] moves)</b>
Parameters: moves - the moves to be analysed Returns: the number of moves contained within moves
<b>public void clearMoves()</b>
Clears all moves and the associated explanation
<b>public boolean isPieceInCorrectPosition(Cubie cubie)</b>
Parameters: cubie - the cubie to be analysed Returns: true if the specified cubie is in the correct position on the cube (regardless of orientation); false otherwise
<b>protected int getIndexOfDestination(Cubie cubie)</b>
Parameters: cubie - the cubie to be analysed Returns: the destination of the cubie on a solved cube according the current permutation of the centres.
<b>private int getShortestOffset(int start, int end)</b>
Parameters: start - the starting position end - the ending position Returns: the shortest offset between the start and end, e.g. 3 -> -1
<b>public java.lang.String getStringMoves()</b>
Returns:

the recorded moves as a string with a space between each move
<b>public static java.lang.String getStringMoves(java.util.LinkedList&lt;java.lang.String&gt; moves)</b>
Parameters:
moves - the moves to be returned as a string
Returns:
the specified moves as a string with a space between each move
<b>public int getNumTrailingU()</b>
Returns the number of trailing U moves in the recorded moves and removes them simultaneously.
Returns:
the number of trailing U moves in the recorded moves
<b>public static java.lang.String getReverseStringMoves(java.util.LinkedList&lt;java.lang.String&gt; originalMoves)</b>
Parameters:
originalMoves - the moves to be reversed
Returns:
the original moves in reverse order as a string with a space between each move
<b>public static java.lang.String getReverseStringMoves(java.lang.String moves)</b>
Parameters:
moves - the moves to be reversed
Returns:
the moves in reverse order as a string with a space between each move
<b>public static java.lang.String getKeyToMove(java.lang.String key)</b>
Parameters:
key - the key that was pressed on the keyboard
Returns:
the move associated with that key press
<b>public static boolean isValidMove(java.lang.String move)</b>
Parameters:
move - the move to be checked
Returns:
true if the move is valid; false otherwise
<b>public java.lang.String getStateString()</b>
This returns a string representation of the current state of the cube so that it can be saved/loaded to/from file. The first two colours are top and front centres, and other colours are each sticker of each corner then each edge.
Returns:
the current state of the cube represented as a string
<b>public void applyStateString(java.lang.String stateString)</b>
Extracts the state stored in stateString and applies the corresponding data to the cube so that the cube is in the state specified in stateString
Parameters:
stateString - the state to be given to the cube

## Class - Sorter

<b>public static void sortByTime(SolveDBType[] list, int start, int end)</b>
Checks that the list is not null or empty then sorts the list with the fastest time first and the slowest time last.
Parameters:
list - the list to be sorted

start - the index of the first element to be sorted  
end - the index of the last element to be sorted

**public static void sBT(SolveDBType[] list, int start, int end)**

Sorts a list with the fastest time first and the slowest time last.

Parameters:

list - the list to be sorted

start - the index of the first element to be sorted

end - the index of the last element to be sorted

**public static void sortByDateAdded(SolveDBType[] list, int start, int end)****throws java.text.ParseException**

Sorts the list with the earliest date first and the latest date last.

Parameters:

list - the list to be sorted

start - the index of the first element to be sorted

end - the index of the last element to be sorted

Throws:

java.text.ParseException - if one of the dates cannot be parsed, i.e. the format is wrong

**public static void sortBySolveID(SolveDBType[] list, int start, int end)****throws java.lang.Exception**

Parameters:

list - the list to be sorted

start - the index of the first element to be sorted

end - the index of the last element to be sorted

Throws:

java.lang.Exception - if one of the elements has not been instantiated etc

**public static void swap(SolveDBType[] list, int i, int j)**

Parameters:

list - the list in which elements will be swapped

i - the index of the first element to be swapped

j - the index of the second element to be swapped

**public static void reverseArray(SolveDBType[] list)**

Reverses the order of the specified array,

e.g. {1, 7, 4} would be come {4, 7, 1}

Parameters:

list - the array to be reversed

**public static void sortByAverageThenTime(MemberCompetition[] list, int start, int end)**

Checks that the list is not null or empty then sorts the list with the best average first and the worst average last.

Parameters:

list - the list to be sorted

start - the index of the first element to be sorted

end - the index of the last element to be sorted

**public static void sBA(MemberCompetition[] list, int start, int end)**

Checks that the list is not null or empty then sorts the list with the best average first and the worst average last.

Parameters:

list - the list to be sorted

start - the index of the first element to be sorted

end - the index of the last element to be sorted

**public static void sBA(MemberCompetition[] list, int start, int end)**

Sorts the list with the best average first and the worst average last.

Parameters:

list - the list to be sorted

start - the index of the first element to be sorted

end - the index of the last element to be sorted

**public static void swap(MemberCompetition[] list, int i, int j)**

Parameters:

list - the list in which the elements will be swapped

i - the index of the first element to be swapped

j - the index of the second element to be swapped

**public static void quickSort(double[] list)**

Sorts the specified list into ascending order. The argument is checked to see if it is null or has fewer than 2 elements.

Parameters:

list - the list to sort

**private static void qSort(double[] list, int start, int end)**

Sorts the specified list in to ascending order.

Parameters:

list - the list to be sorted

start - the index of the first element to be sorted

end - the index of the last element to be sorted

**private static void swap(double[] list, int i, int j)**

Swaps the elements at the specified indices in the specified list

Parameters:

list - the list in which elements will be swapped

i - the index of the first element to be swapped

j - the index of the last element to be swapped

## Class – Statistics

**public void setTimes(java.util.LinkedList<Solve> times)**

Sets the times to be used for averages

Parameters:

times - the times to be used for averages

**public double getRecentAverageOf(int sizeOfAverage)**

Returns an average of the specified size. The times used are the most recent ones.

Parameters:

sizeOfAverage - the size of the average

Returns:

the average of the specified size

**public double getOverallMean()**

Returns:

the overall mean of all times, ignoring DNFs

**public java.lang.String getRecentFormattedStandardStatistics()**

Returns:

a string representing all calculated statistics in a formatted fashion

**public java.lang.String[] getRecentFormattedStatisticsArray()**

Returns:

a array containing all calculated statistics in a formatted fashion

**public static double getAverageOf(int sizeOfAverage,  
java.util.LinkedList<Solve> times)**

Parameters:

sizeOfAverage - must be 5 or over

times - the times from which the average is calculated

<p>Returns: the average of the specified size.</p> <p><b>public static double getAverageOf(int sizeOfAverage, java.lang.String[] stringTimes)</b></p>
<p>Parameters:</p> <p>sizeOfAverage - the size of the average (must be greater than or equal to 5) stringTimes - the times from which the average will be calculated</p>
<p>Returns: the average of the specified times</p> <p><b>public static double getOverallMean(java.util.LinkedList&lt;Solve&gt; times)</b></p>
<p>Parameters:</p> <p>times - the times from which the overall mean will be calculated</p>
<p>Returns: the overall mean of all specified times, ignoring DNFs</p> <p><b>public static java.lang.String getFormattedAverage(int sizeOfAverage, java.util.LinkedList&lt;Solve&gt; times)</b></p>
<p>Returns a formatted average (i.e. all times are separated by commas and the fastest and slowest times are surrounded with brackets) of the specified size as the string</p> <p>Parameters:</p> <p>sizeOfAverage - the size of the average to be returned times - the times from which the average is calculated</p>
<p>Returns: a formatted string of the average in the form "[average] [time_1], [time_2], ..., [time_n]"</p> <p><b>private static void sortByDNF(double[] list)</b></p>
<p>Sorts the specified list so that '-1' elements are at the end</p> <p>Parameters:</p> <p>list - the list to be sorted</p> <p><b>public static double getBestAverageOf(int sizeOfAverage, int averageIndex, java.util.LinkedList&lt;Solve&gt; times)</b></p>
<p>Parameters:</p> <p>sizeOfAverage - the size of the average to be calculated averageIndex - the index of the element in averages that contains the properties for the average. times - the times from which the average will be calculated</p>
<p>Returns: the best average of the specified size.</p> <p><b>public static java.lang.String getFormattedStandardStatisticsString(java.util.LinkedList&lt;Solve&gt; times)</b></p>
<p>Parameters:</p> <p>times - the times from which the formatted statistics will be generated.</p>
<p>Returns: a string containing the all calculated statistics in a formatted fashion.</p> <p><b>public static java.lang.String[] getFormattedStatisticsArray(java.util.LinkedList&lt;Solve&gt; times)</b></p>
<p>Returns each the formatted string for each average in an array</p> <p>Parameters:</p> <p>times - the times from which the statistics will be generated</p>
<p>Returns: an array containing strings which represent the statistics in a formatted fashion</p> <p><b>public static int getNumDNF(int sizeOfAverage, java.util.LinkedList&lt;Solve&gt; times)</b></p>
<p>Parameters:</p>

sizeOfAverage - the size of the average currently being calculated  
 times - the times from which the number of DNFs will be determined

Returns:  
 the number of 'DNF'-1 times in the past sizeOfAverage times

**public static double getFastestTimeInPrevious(int sizeOfAverage,  
 java.util.LinkedList<Solve> times)**

Returns the fastest time in the previous sizeOfAverage times

Parameters:

sizeOfAverage - the size of the average being calculated  
 times - the times from which the fastest time will be determined

Returns:  
 the fastest time in the previous sizeOfAverage times

**public static double getSlowestTimeInPrevious(int sizeOfAverage,  
 java.util.LinkedList<Solve> times)**

Returns the slowest time in the previous sizeOfAverage times

Parameters:

sizeOfAverage - the size of the average being calculated  
 times - the times from which the slowest time will be determined

Returns:  
 the slowest time in the previous sizeOfAverage times

## Class – TextFile

**public void setFilePath(java.lang.String filePath)**

Sets the file path of the text file

Parameters:

filePath - the file path of the file to be accessed

**public void close()**

Closes the buffers accessing the file

**public void setIO(int io)**

**throws java.lang.Exception**

Changes the I/O method to the specified I/O method, it can be either read, write, or writing and appending.

Parameters:

io - the IO method

Throws:

java.lang.Exception - if the file cannot be found, opened etc.

**public int getNumLines()**

**throws java.lang.Exception**

Returns:

the number of lines in the text file

Throws:

java.lang.Exception - if the file cannot be accessed properly.

**public static int getNumLines(java.lang.String filePath)**

**throws java.lang.Exception**

Parameters:

filePath - the path of the file whose number of lines will be counted.

Returns:

the number of lines in the specified file

Throws:

java.lang.Exception - if the file cannot be accessed properly

**public java.lang.String readLetter()**

**throws java.lang.Exception**

Returns: the next letter read from the file. Throws: java.lang.Exception - if the file cannot be accessed properly
<b>public java.lang.String readLine() throws java.lang.Exception</b>
Returns: the next line read in the file Throws: java.lang.Exception - if the file cannot be accessed properly
<b>public java.lang.String[] readAllLines() throws java.lang.Exception</b>
Returns: an array of strings containing the data of each line in the file Throws: java.lang.Exception - if the file cannot be accessed properly
<b>public void writeLine(java.lang.String data) throws java.lang.Exception</b>
Parameters: data - writes the specified data to the file Throws: java.lang.Exception - if the file cannot be accessed properly
<b>public void writeLine(char data) throws java.lang.Exception</b>
Parameters: data - writes the specified data to the file Throws: java.lang.Exception - if the file cannot be accessed properly
<b>public void writeLine(double data) throws java.lang.Exception</b>
Parameters: data - writes the specified data to the file Throws: java.lang.Exception - if the file cannot be accessed properly
<b>public void writeLine(long data) throws java.lang.Exception</b>
Parameters: data - writes the specified data to the file Throws: java.lang.Exception - if the file cannot be accessed properly
<b>private void writeLineGeneral(java.lang.String data) throws java.lang.Exception</b>
Parameters: data - writes the specified data to the file and adds a new line character. Throws: java.lang.Exception - if the file cannot be accessed properly
<b>public void write(java.lang.String data) throws java.lang.Exception</b>
Parameters: data - writes the specified data to the file Throws: java.lang.Exception - if the file cannot be accessed properly
<b>public void write(char data)</b>

**throws java.lang.Exception**

Parameters:

data - writes the specified data to the file

Throws:

java.lang.Exception - if the file cannot be accessed properly

**public void write(double data)**

**throws java.lang.Exception**

Parameters:

data - writes the specified data to the file

Throws:

java.lang.Exception - if the file cannot be accessed properly

**public void write(long data)**

**throws java.lang.Exception**

Parameters:

data - writes the specified data to the file

Throws:

java.lang.Exception - if the file cannot be accessed properly

**private void writeGeneral(java.lang.String data)**

**throws java.lang.Exception**

Parameters:

data - writes the specified data to the file.

Throws:

java.lang.Exception - if the file cannot be accessed properly

## Class – TimeGraph

**public void draw()**

Draws the graph in the window

**public void setGraphTitle(java.lang.String chartTitle)**

Sets the title of the chart

Parameters:

chartTitle - the title of the chart

**public void resetGraphView()**

Resets the graph to its default zoom.

**public void setDataset(org.jfree.data.category.DefaultCategoryDataset dataset)**

Sets the dataset for the graph

Parameters:

dataset - the dataset to be used to draw the graph

## Class – TimeListPopUp

**public void selectAllTimeText()**

Selects the text in the time field so that the user can edit it without having to use backspace.

**public void setSolve(Solve currentSolve)**

Sets the current solve to work with in the Solve Editor window.

Parameters:

currentSolve - the solve to work with.

**public void keyPressed(java.awt.event.KeyEvent arg0)**

Specified by:

keyPressed in interface java.awt.event.KeyListener

**public void keyReleased(java.awt.event.KeyEvent arg0)**

Specified by:  
keyReleased in interface java.awt.event.KeyListener  
**public void keyTyped(java.awt.event.KeyEvent arg0)**  
Specified by:  
keyTyped in interface java.awt.event.KeyListener

## Class – Tutorial

<b>public void loadTutorial(java.lang.String filePath) throws java.lang.Exception</b>	Loads the tutorial stored in the file at the specified file path. The fields store the data from the file.  Parameters: filePath - the path of the file which contains the tutorial Throws: java.lang.Exception - if the file cannot be accessed properly or is not formatted correctly
<b>private int getNumSubTutorials()</b>	Returns: the number of sub-tutorials in the tutorial
<b>public void loadNextSubTutorial()</b>	Loads the next sub-tutorial, i.e. the sub-tutorial index is incremented and other indices are reset.
<b>public void loadPreviousSubTutorial()</b>	Loads the previous sub-tutorial, i.e. the sub-tutorial index is decremented and other indices are reset.
<b>public java.lang.String getScramble()</b>	Returns: the scramble for the current sub-tutorial
<b>public java.lang.String getDescription()</b>	Returns: the description for the current sub-tutorial
<b>public void loadNextHint()</b>	Loads the next hint for the current sub-tutorial, i.e. the hint index is incremented. The hint index will follow 0, 1, 2, ..., n, 0, 1, 2, ...
<b>public void loadPreviousHint()</b>	Loads the previous hint for the current sub-tutorial, i.e. the hint index is decremented. The hint index will follow ..., 2, 1, 0, n, n - 1, ...,
<b>public java.lang.String getHint()</b>	Returns: the current hint for the current sub-tutorial
<b>public java.lang.String getExplanation()</b>	Returns: the explanation for the current sub-tutorial
<b>public java.lang.String[] getOptimalSolutions()</b>	Returns: the optimal solutions for the current sub-tutorial
<b>public boolean criteriaFilled()</b>	Returns: true if the criteria for the current sub-tutorial has been filled by the user; false otherwise
<b>private java.awt.Color[] getStickers(java.lang.String stickerFormat)</b>	This returns an array of Colors containing all colours denoted in the stickerFormat

parameter.

For example, Input: "r-g" will return {Color.red, Color.green}

Parameters:

stickerFormat - denotes the stickers with letters separated by '-' characters.

Returns:

an array of Colors containing all colours denoted in the stickerFormat parameter.

**public boolean requiresUserAction()**

Returns:

true if the current sub-tutorial requires/allows the user to perform moves, i.e. moves are not locked;

false otherwise

**public boolean requiresUserSolution()**

Returns:

true if the current sub-tutorial requires the user to solve a problem;

false otherwise

**public int getOptimalSolutionLength()**

Returns:

the number of moves in the first optimal solution.

**public static int getNumMovesWithoutRotations(java.lang.String moves)**

Parameters:

moves - the moves to be analysed

Returns:

the number of moves in moves but not counting rotations

**public static int getNumMovesWithoutRotations(java.util.LinkedList<java.lang.String> moves)**

Parameters:

moves - the moves to be analysed

Returns:

the number of moves in moves but not counting rotations

**public boolean isLoaded()**

Returns:

true if the tutorial is loaded;

false otherwise

**private boolean isCrossSolved()**

Returns:

true if the cross is solved;

false otherwise

**private boolean isCornerPermutationSolved()**

Returns:

true if the permutation of the corners is solved;

false otherwise

**private boolean isEdgePermutationSolved()**

Returns:

true if the permutation of the edges is solved;

false otherwise

**public int getSubTutorialIndex()**

Returns:

the index of the current sub-tutorial

**public boolean isFirstSubTutorial()**

Returns:

true if the current sub-tutorial is the first sub-tutorial;

false otherwise

**public boolean isLastSubTutorial()**

Returns:  
true if the current sub-tutorial is the last sub-tutorial;  
false otherwise