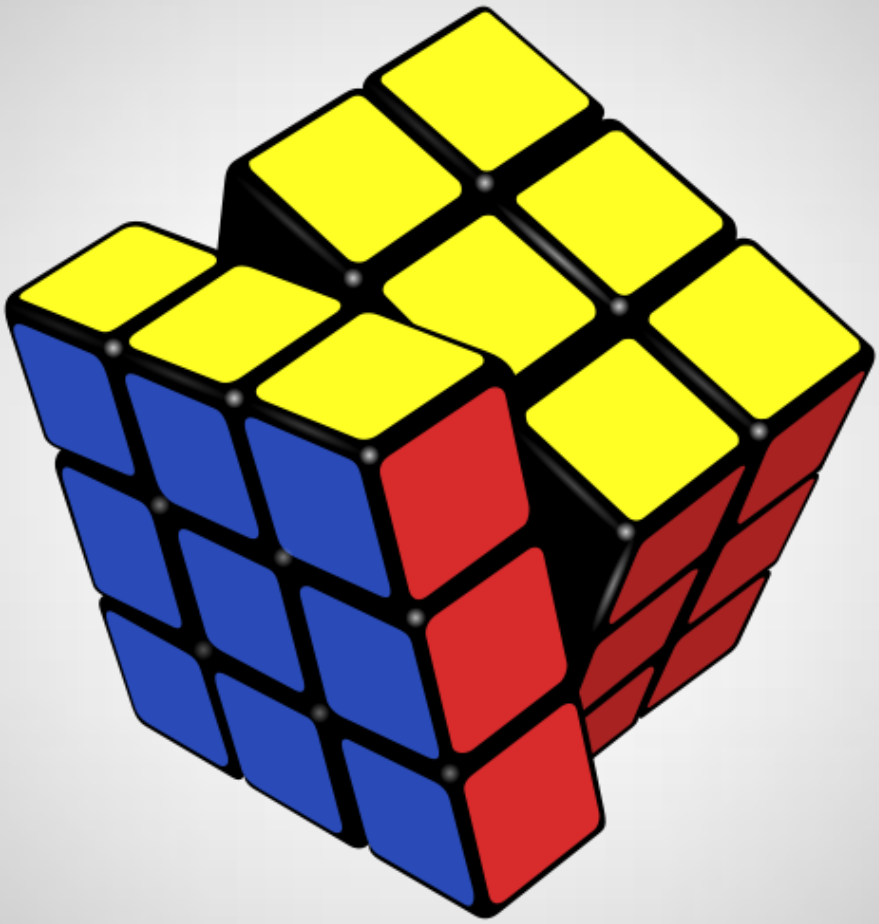
SECTION 1

ANALYSIS



Contents

Problem Identification and Background

My client, Dr McIvor, is the founder of my school's mathematics club. The club was started so that students interested in mathematics could give presentations and talks relating to different areas of mathematics that they found interesting, such as group theory, the history of mathematics etc. The club embarked on a task to learn how to solve the Rubik's cube, but soon encountered difficulties due to the complexity of the puzzle. This means that Dr McIvor does not have enough time to both teach new members the method and teach existing members advanced concepts. I intend to design a system that will solve this problem, allowing the ~ 30 members to progress at the same rate, help newcomers learn faster, and allow Dr McIvor to manage the information more effectively. I am a member of the club, and I frequently help Dr McIvor in teaching the new members, so I understand the concepts and difficulties that need to be addressed.

There are also problems for learners outside the club. For example, after one learns how to solve the Rubik's cube, they may wish to improve their solve-time. Their progress can be dilatory due to lack of understanding or inadequate resources available to them. Furthermore, if someone needs assistance with a particular state/case, he or she must either tediously replicate the state in a custom image editor, or take a picture of the cube and ask for help on a forum. One can track their progress by saving their times in some form of digital timer, but this saves very little information and can become frustrating when saving many times. A very important aspect of progression in any task is learning from your mistakes, but this cannot be achieved unless one records many solves on a camera, or they write down their solution, which is almost impossible to remember after the cube is solved.

I would like to design a system, called *Kuubik*, which will provide a graphical interface for beginners to learn and practise, and provide features for Dr McIvor in order to store details about the members and the competitions in which the participate.. The program should have interactive features/tutorials so that problems can be solved by the user during their learning. To remove the need for learners to seek help on the Internet, I will include an auto-solving feature, which can either solve the entire cube, or solve a selected piece. Every solve (with corresponding moves, time etc.) can be recorded in a database so that users can review their solves over their entire learning-time. I also plan to include features that track the current average of 5, 12, 50, and 100, and will plot a graph of the previous 12 times. For Dr McIvor, there will be features to record the details of the members, such as name, date of birth etc., and the times achieved by the members in each of the competitions.

I will try to meet with Dr McIvor once a month to review the progress on the system and to ascertain any changes that should be made.

Description of the Current System

Currently, printed hand-outs are given to new members to help them learn. These hand-outs include a mostly text-based description of how to solve the cube, with a few images corresponding to the current step. Some essential algorithms (sequences of moves) are included.

Member details (name, age, form class, etc.) and their competition details (ranking, average time, etc.) are kept in a word document by Dr McIvor. The competitions are held once every two weeks. After students compete in a competition, Dr McIvor writes out a Competition-Rank table for that competition. The person with the best (fastest) average comes first and the person with the worst (slowest) average comes last. If two people have the same average, then the individual times in the average are compared to determine the order.

Several problems arise from the current system that will be improved by the final system:

- After learning a beginner's method this way, members may have acquired some bad habits and this will affect their progress.
- The instructions provided by Dr McIvor can be lost easily, and if the instructions are updated, they need to be printed for every member.
- Paper-based instructions are very difficult to follow, and are not interactive which makes the learning process even more difficult.
- It is very difficult to record the solution to a particular cube-state; the user must record themselves on video and try to reconstruct their moves during the solve if they want to obtain their original solution.
- There are pieces of software designed to record the time taken to solve a Rubik's cube, but recording the times over a lifetime is frustrating in these programs because of the way in which they calculate the statistics. Users might become very fast, but will wish to save their slower times for reference; the current timing programs do not offer this sort of functionality.
- Storing member details and competition details is error-prone and tedious with the current paper-based system. In addition, calculating the rankings for each competition is time consuming and even more error-prone.

Identification Prospective Users

The main users of the system will be the ~ 30 students in the club and those who wish to join the club. Due to the varying competency of the users, an easy-to-use GUI interface will be needed to allow all users to access the system's features, and will need to be accessible enough in order to be a viable alternative to using a real Rubik's cube. These users will interact with the 'learning suite' in order to study a method for solving the cube. An introduction and/or manual will be needed to teach the users how to perform moves and rotations (moves will be mapped to keys on a keyboard).

For Dr McIvor, I will include database features so that member details and competition details can be stored. Times and averages for individual members will be stored, and 'Competition' rankings will be calculated automatically. During a competition, Dr McIvor will input the times achieved by each member and the calculations will be performed on this data.

Identification of User Needs and Acceptable Limitations

To identify the aspects of the current system, and the aspects that need to be incorporated in the new system, I issued questionnaires to two of the members and had an interview with Dr McIvor. By examining the existing documentation (see page 26), and gathering information from these questionnaires (see pages 2-5 of the appendix section) and the interview (see page 22), I was able to determine the requirements of the system.

The system must…

- Offer a 'timing suite' in which users can solve the virtual cube under timed conditions. This suite will include several features:
    - Inspection time
    - Solve cancellation
    - Time-tracker (lists the times, and calculates current average of 5, average of 12 etc.)
    - Visible time-display
    - Ability to add penalties and comments to times (for example, if something interesting happened during the solve, such as a last-layer skip)
    - Ability to save a copy of the statistics to pdf
    - Ability to save an individual solve's information (time, penalty, comment, scramble solution)
    - Ability to edit, delete, and add times manually
- Offer a 'learning suite' in which users can learn how to solve the cube using interactive tutorials and real-time tips. The suit will include several features including:
    - Detailed walk-through of these sub-steps:
        - Cross
        - Corners
        - Edges
        - Orientation
        - Permutation
    - Access to algorithm list/table
    - Solve-this-piece: the program will show how to solve the selected piece.
    - Save the current state of the cube.
    - Solve the cube – the system should be able to solve a cube from any state and display the moves required to solve the cube.
- Ability to view solves executed automatically after loading them from file.
- History features, which allow the user to sort solves by date and time duration, and view a solve executed automatically by scrambling it using the provided scramble and then performing the moves of the solution.
- Manage a database holding member details, competition details, and competition times.
- Offer simple means of displaying competition rankings.
- Store scrambles, e.g. R U R' U', F2 R2 F2 etc.

The students must be able to…

- Learn how to solve the cube using the learning suite.
- Share their progress with others easily using the save features and statistics calculator/generator.
- View statistics about the current session and save this session to the database.
- Save algorithms in the algorithm database.
- Generate solutions to difficult states of the cube.
- Solve the cube while simultaneously recording the moves performed and the time taken.
- Paint the stickers to represent a custom state and generate a solution to this state.
- Change preferences for the system.
- Input scrambles and use these to scramble the cube automatically.

Dr McIvor/the administrator must be able to…

- Enter, update, and delete member details, competition details, and member-competition times.
- View rankings for a selected competition.

Limitations and Areas Considered for Future Improvement

- The system must be complete by April 2014.
- The amount of data can increase very quickly, so there must be warnings about backing up data to help prevent loss of this data.
- Since the system will not be operating online, some data will still be entered manually, such as individual times during a competition. If the system were online, then competitions times could be added automatically.
- A race mode, which would allow several users across the Internet to receive the same scramble and race against each other.
- Tutorials for other methods. My program will focus on a modified beginner's method, but advanced and alternate methods could be added (such as CFOP, Roux, Petrus, ZZ etc.).
- The system could be developed for mobile devices.
- Using the Facebook API, the user could update their Facebook status from within the program and include the statistics, solution etc.

Constraints

- The users will have varying levels of competency, so the user interface must be very straightforward.
- To view the PDFs, the user must have the appropriate software installed.
- The client computer must have Java installed.

Data Sources and Destinations

Current System

Students can store their times in timing software or in a database. Algorithms can be found on the Internet. Dr McIvor stores all details in a word document, and creates competition-ranking tables manually using the same software. During a competition, competitors complete five solves; the fastest and slowest times are ignored, and a mean of the remaining three is calculated. This is recorded as their 'Average of 5' for that competition. The dates for competitions and the details of the members are recorded.

Proposed System

Data	Source	Input Method	Destination	Output Method
Moves to apply (real-time)	User	User presses keys on keyboard corresponding to moves on the cube.	Program will handle keyboard events.	Displayed on screen: the stickers on the cube will change according to the move applied.
Cube-Algorithms	User	The system will display existing algorithms in a table and will have a button that will add a row to the table into which the user can enter information about a new algorithm.	Stored in 'Algorithm' table	Displayed on screen: when accessing the algorithm database, a table will be presented showing the information for each of the algorithms stored.
Solves	User	There will be a menu item which will trigger a form to be displayed into which the user can enter information about a solve (time, penalty, comment, scramble, and solution). If the cube is solved using the built-in timing features, the solve information will be recorded automatically. The information of each solve can be stored in a text file and be loaded from a text file.	Initially stored temporarily, but can then be submitted to the 'Solve' table in the database, or an individual solve's information can be saved to/loaded from a text file.	Displayed on screen: the times will be appended to the list of recorded times in the window.
Tutorials	Pre-written text files.	The user can choose a tutorial from the menu bar, or can load a custom tutorial from file.	The system will extract the information from the text files and use this to generate the interactive tutorial.	Displayed on screen: manifests itself as a 'tutorial' mode with which the user can interact in order to learn.

Scrambles	User	An input dialog will be displayed, allowing the user to enter the scramble and this will be listed along with other scrambles.	Stored temporarily, but can be saved to a text file from within the program.	Displayed on screen in a list.

Data	Source	Input Method	Destination	Output Method
Member Details	User	A form will be provided to enter details about a new member, or to update details about an existing member.	Stored in member table in database.	Displayed on screen: the details of all members will be shown in a table.
Competition Details	User	The system will have an appropriate form to enter the data.	Stored in competition table in database.	Displayed on screen: a form will be shown when competition details are requested.
Member-Competition Times	User	A member will perform 5 solves during a competition. After each solve, Dr McIvor will enter the time into a form. When all 5 solves have been completed, the information will be added to member-competition table.	Stored in member-competition table in database.	Displayed on screen: times will be stored for a particular member and competition. There will be an option to view the times for a competition (ordered by average). A new window will open to show the selected times.

Data Volumes

Members

The club has approximately 30 members. Member ID, Forenames, Surname, Gender, Date of Birth, Email, and Form Class will be stored for each member. There are only 30 members, so the amount of data is expected to be reasonable small (≈ 50 KB). Some members' details will need to be deleted annually (if pupils change school, or when pupils finish secondary school education).

Competitions

A competition takes place once every two weeks. Competition ID, and Date will need to be stored for each competition. The amount of data stored for each competition record is very small, so, even after ten years, the overall amount of data is not a problem in terms of storage space (≈ 2 KB).

Member-Competition Times

In a competition, each of the 30 members will perform 5 solves, resulting in 150 times being stored per competition. Member ID, Competition ID, and the times will be stored in this table. This table should not exceed approximately 100 KB per year.

Solves

Solves will be recorded very regularly - approximately 20 solves per day. If this information is saved to the database, then the size of the solve table in the database will increase by about 10 KB per day.

Analysis Data Dictionary

Current System

Member Details

Variable	Data Type
Name	Text
Date of Birth	DD/MM/YYYY
Form Class	2 digits and 1 letter: e.g. 11M

Competition Details

Variable	Data Type
Competition Number	Integer
Date	DD/MM/YYYY

Algorithm

Variable	Data Type
Algorithm Name	Text
Moves	Text
Comments	Text

Competition-Time

Variable	Data Type
Member Name	Text
Competition Number	Integer
Solve Number	Integer
Duration	Time format, i.e. XX:XX.XX
Average of 5	Time format, i.e. XX:XX.XX

Proposed Data Dictionary

Member

Variable	Data Type
(Primary Key) MemberID	AutoNumber
Forenames	Text
Surname	Text
Gender	Text
DateOfBirth	DD/MM/YYYY – must be earlier than the current date.
Email	Text: Regex match.
FormClass	Text: Regex Match: "0*(8|9|10|11|12|13|14)[MRSTW]"

Competition

- If the date is not in the valid format, or is a non-existent date, then an error message should be shown to inform the user.

Variable	Data Type
(Primary Key) CompetitionID	AutoNumber
Date	DD/MM/YYYY – must be earlier than the current date.

Member-Competition

Variable	Data Type
(Foreign Key) CompetitionID	Integer
(Foreign Key) MemberID	Integer
Time1	Text – between 00:00.00 and 59:59.59 inclusive.
Time2	Text – between 00:00.00 and 59:59.59 inclusive.
Time3	Text – between 00:00.00 and 59:59.59 inclusive.
Time4	Text – between 00:00.00 and 59:59.59 inclusive.
Time5	Text – between 00:00.00 and 59:59.59 inclusive.

Algorithm

Variable	Data Type
AlgorithmID	AutoNumber
MoveSequence	Text
Comment	Text

Solve

- If the time is entered incorrectly, then an error message listing all the valid types of input should be shown.

Variable	Data Type
SolveID	AutoNumber
Time	Text – between 00:00.00 and 59:59.59 inclusive.
Penalty	Text
Comment	Text
Scramble	Text
Solution	Text
DateAdded	Date

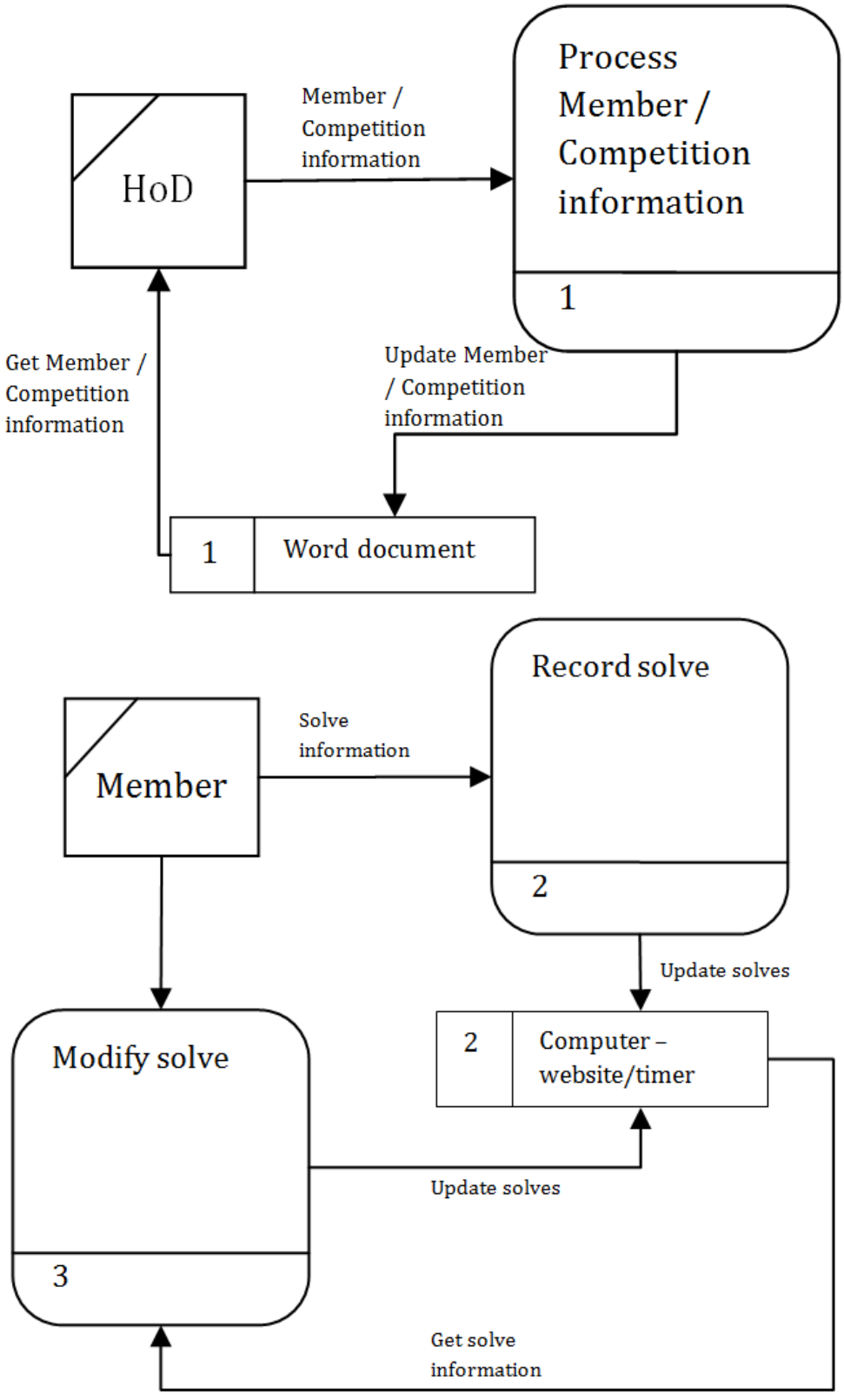
Terminology

Term	Meaning
Solve	Solve can refer either to the act of solving a Rubik's cube, or can refer to the process of solving, e.g. "I just completed a solve", as a noun. The noun form is actually more common than the verb.
Cross	The cross is the most common first step of solving methods. The cross consists of four solved edges on the same face. The most common cross colour is white.
Corner	A corner is any piece on a Rubik's cube that has three stickers.
Edge	An edge is any piece on a Rubik's cube that has two stickers.
Centre	There are six centres on the Rubik's cube – White, Yellow, Red, Orange, Green, and Blue. Each face consists of 9 stickers, and the centre sticker of each face is known as the 'centre'.
Cubie/Piece	The general term for a corner or an edge.
Algorithm	Any sequence of moves, e.g. R U R' U'
Layer	A layer refers to a group of 8 pieces that can all be affected by a single move, e.g. the w-r-g, w-g, w-g-o, w-o, w-o-b, w-b, w-b-r, w-r pieces constitute a layer.
Orientation	The orientation of a piece refers to how a corner is twisted/how an edge is flipped in relation to the centres. An edge is oriented if it can be solved using only U, D, R, L, F and B moves. A corner is oriented if the top sticker of the corner is the same as the top or bottom centre.
Permutation	Permutation refers to the position of the pieces relative to each other.
OLL	Orientation of the Last Layer: This refers to the process of flipping/twisting all pieces so that all stickers on the top face are the same colour.
PLL	Permutation of the Last Layer: This refers to the process of solving the permutation of all pieces in the last layer.
DNF	Did Not Finish: This denotes a disqualified time.
Scramble	The 'scramble' is the set sequence of moves used to 'mix up' the cube.
Solution	The moves used to solve a cube after being scrambled.

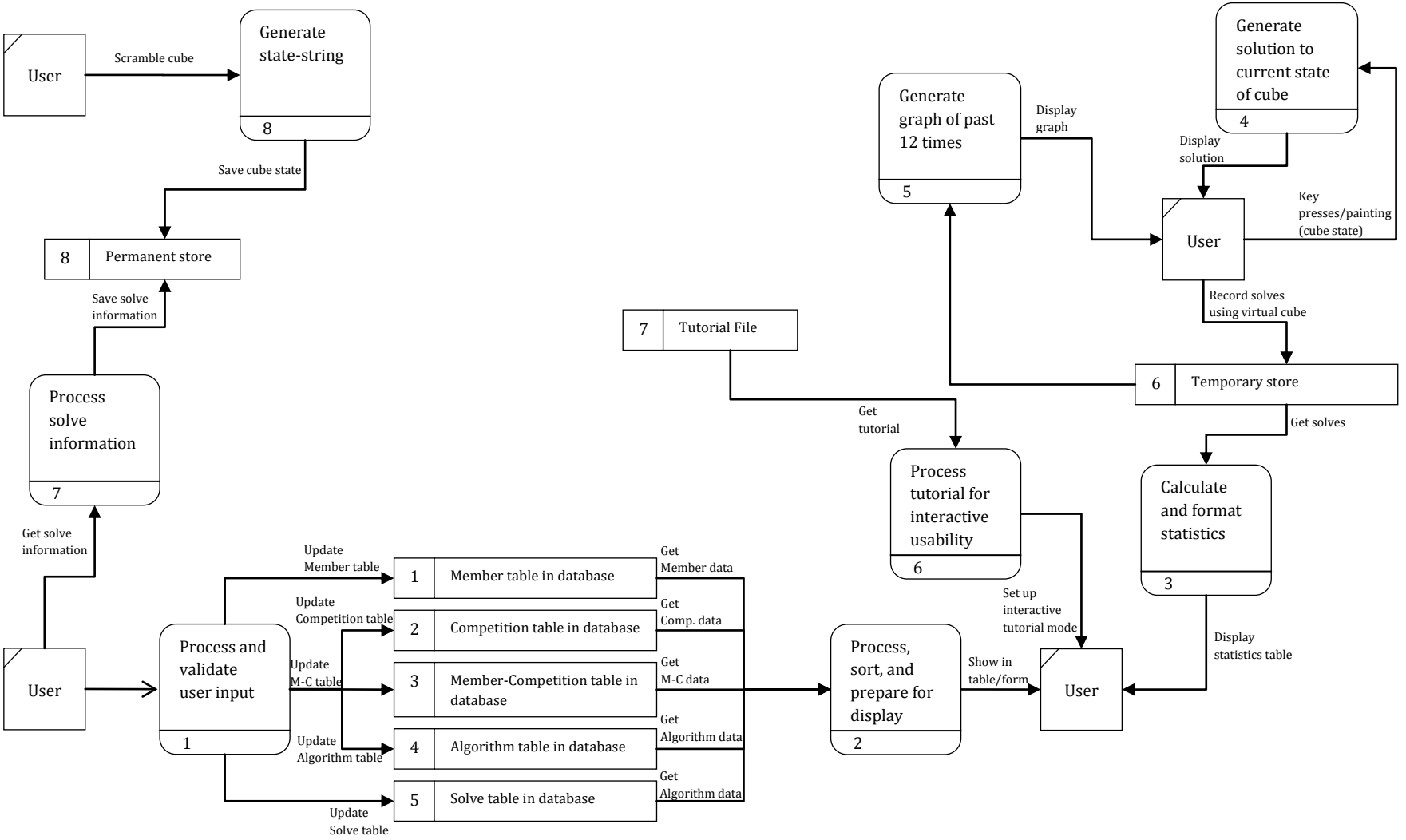
Data Flow Diagrams

Existing System

Level 1



Proposed System



Objectives for the Proposed System

1. The system should be able to run Windows 7.

2. The system should allow the user to record solves with ease and should store the information for the solve automatically.

3. The system should provide an interactive learning process.

4. The system should allow the user to load data into the program, such as solves shared over the Internet.

5.
    a. The system should calculate statistics automatically and display them to the user.
    b. The statistics should be able to be saved to a pdf file.

6. The system should load within 10 seconds.

7. The system must allow the user to store pertinent data, such as solve information, times, algorithms etc.

8.
    a. The system should present a graph of the previous 12 times in a separate window.
    b. There should be an option to save the time graph as an image on the user's computer.

9. The system should allow the execution of previous solves to be performed automatically.

10. The system should be able to generate solutions to any (valid) given state and show this solution to the user.

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 automatically.

12. The system should be able to generate a solution for a selected piece.

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.

14.
    a. The system must allow the user to save the solves in the current session to the database.
    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.

15.
    a. The system must provide easy means of navigation.
    b. There must be clearly labelled menu items which take the user to the relevant part of the system.
    c. There must be understandable buttons throughout to help the user access all features of the system.
    d. Most of the database forms should have similar Add, Edit, and Delete buttons so that the user can manipulate data easily.

16.
    a. The system should present error messages when the user performs an incorrect action, such as submitting invalid data.
    b. These error messages should explain what the error is, and should give some indication of how to fix the problem.

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.

18.
    a. The system must have the ability to add, delete, and update member details.
    b. The following details about members should be stored: forenames, surname, gender, date of birth, email, and form class.
    c. There should be labelled text field boxes to enter this information with text indicating what the field is looking for.

19. The system must be able to create a table to display all details stored on the members in the club.

20. The system must have the ability to add, delete, and update competition details, e.g. the date of the competition.

21. The system must be able to create a table showing the ID and date for each competition.

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).
    b. There should be labelled text field boxes to enter this information with text indicating what the field is looking for.

23.
    a. The system must be able to create a table to display member-competition information showing all records saved for a selected competition.
    b. The 'average' of each record should be calculated as displayed automatically.
    c. The records should be sorted according to average.

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.
    b. The results should be returned within 0.5 seconds.

25. The system should be able to store up to 100 members.

26. The system must be completed by April 2014.

27. The system must be relatively bug-free.

28.
    a. In order to realistically create the system I shall use high-level programming languages.
    b. I shall use Java to design and program the main system.
    c. I shall use SQLite to manage, retrieve and store information from/to the database.

Realistic Appraisal of the Feasibility of Potential Solutions

1. Non-Computerised Solution

- A detailed document could be devised which would describe a method for solving the Rubik's cube. This document could then be distributed to members of the club.

- At home, members could record their times using a stopwatch and then write the time on a piece of paper.

- Dr McIvor could store and find member details easier with a structured system for storing the data. Information could be written in several books to separate the data accordingly.

Advantages

- Users do not need to know how to use computers.
- Finding and storing information in very simple and intuitive.

Disadvantages

- Calculating averages (e.g. average of 5) requires the user to perform a calculation after *every solve*. This is extremely tedious.
- Locating member information is slow.
- Damage/loss of books could result is loss of data.
- In visual situations, interactive and visual solutions are much more effective than text solutions.

This method would not improve the current situation by much. The main problem that the proposed system will fix is the difficult of learning; it is almost impossible to do this with a non-computerised solution. The adjustments to Dr McIvor's system would be minimal, and probably would not improve the effectiveness or speed of the system.

2. Web-Based Solution

- The functionality of the current system could be improved using web-based tutorials implemented in the appropriate languages (such as Javascript and jQuery).
- The solving system would be feasible with the aforementioned languages, and the automated solver would be implemented behind the user interface.

Advantages

- Users could access the system easily over the Internet.
- Installation would not be required.
- Multiplayer extensions would be simpler to accommodate since the system would already be implemented to be interpreted by a web browser.
- The use of paper would be eliminated.
- Dr McIvor could access the data from any workstation that had an Internet connection.

Disadvantages

- I do not have enough experience with Javascript, jQuery, or PHP.
- Users would have to be quite competent with computers in order to identify errors caused by certain web browsers, or if Java was not installed etc.
- Dr McIvor would need to further his competency with computers in order to manage the data effectively.
- If a user did not have an Internet connection available, then they would not have access to the system.

This solution would be appropriate for solving the problem of teaching the members, and improving the way in which data is stored. By having the system available over the Internet, networking features could be added easily. Unfortunately, due to the deadline for this system, I do not have enough time to learn the required languages sufficiently in order to produce an adequate solution.

3. Existing Software

- Some solving simulators exist that allow the user to record their times digitally.
- There are some programs designed to aid users in the learning/practising of a certain sub-step or technique.
- General-purpose software, such as Microsoft Access, could be used to manage the database.

Advantages

- The functionality of different programs could be integrated into a general system in order to provide users with many resources.
- By not having to implement a unique solution, more time could be spent on usability of the system, making it an easier experience for the user.

Disadvantages

- The current programs do not offer an easy means of recording a large database of times.
- The current software is free, which means that it is bug-prone since the developers have no obligation to design (almost) perfect code.
- It would be difficult to integrate the functionality of these programs effectively into one system. It could be more difficult to combine these systems than to design a new one.
- Dr McIvor is not familiar with any database applications, so the transition to this system would be difficult. He wants to use the database for a specific reason, so it would be better to have a system that is tailored to the specific needs of the situation.

Some existing software could be used to help certain areas of the problem, but none of these solve the entire problem of giving the members an easy learning experience and providing an easy method of storing and processing data.

4. New System Implemented using Java and SQLite

Advantages

- I have experience of these languages.
- Java is object-orientated, which means that I have the opportunity to work with classes and advanced ideas, such as polymorphism.
- The Java Virtual Machine (JVM) allows Java programs to be run on any operating system. This is important since the members may have varying operating systems: Windows, Mac, Linux etc.
- The Java Swing library has many features which help in the design of GUIs. This means more time can be spent on improving the usability and features of the program.
- There are many free resources available for these languages.
- Eliminates requirement for Internet access once the program is installed.
- SQLite requires no installation (server-less); many people will be using the system on many different machines, so the absence of installation is desirable.

Disadvantages

- Java is a vast language, so I won't have enough time to learn some of the intricate features such as generics and security.
- Developing the entire system from scratch is more time consuming than enhancing existing software.
- The database would not be accessible over the Internet unless the project was developed further.
- Computer literacy will be more of an issue with a system like this. The system will require a detailed manual to explain all features.

Justification of Chosen Solution

Interview 1

What are the problems with the current system?

Dr McIvor: "The main problem is trying to teach simultaneously new members and existing members. Preferably, I want to spend more time teaching the existing members advanced concepts, but I also want everyone to be involved, so at the moment, I'm having to compromise between the two groups. There are resources available for learning, but they are usually not adequate to teach total beginners. I need a resource that essentially replicates my teaching process. I have tried recording videos for the members, but lack of interactivity discouraged potential members.
In addition, there are problems with my system for storing member and competition details. Storing the information in my word document is starting to become cumbersome, and I would like a digital alternative."

Exactly what will the new system achieve?

Dr McIvor: "The new system should give the members an automatic method of recording solves and the details associated with them. Statistics should be calculated from these times. To help beginners, the system will need to have some sort of 'tutorial'-based learning process – preferably interactive. In addition to the tutorials, it would be advantageous if the user could ask for a solution to a given cube so that the user can get the solution to difficult problems. Most of the information in the system, like algorithms, scrambles, solves and so on, should be able to be saved to the computer. I will need some way of storing and displaying member and competition details, including the times achieved during each competition, which are sorted according to the best average of 5."

Which parts of the current system would you like to retain?

Dr McIvor: "Obviously all current functionality must be retained, but there aren't parts of the current system that I particularly want to retain; I want most elements of the current system to be redesigned to work on computers. Storing member details, competition details, calculations, and teaching new members should be done by the new system."

What type of input is there?

Dr McIvor: "The members of the club will need to enter the time, comment, penalty, scramble, and solution for a solve. Scrambles need to be stored so that they can be used in competitions. The user should be able to enter algorithms so that they can list all the algorithms they need to solve the cube. The stickers on the cube should be able to be changed manually so that if the user has a physical cube, they can replicate the state of the cube in the program. I should be able to enter the existing details for the members and any other information you think is necessary. For competitions, I should be able to enter the date and the times achieved by each of the members in that competition."

What form will the outputs take?

Dr McIvor: “I would like the member and competitions details to be displayed in a similar style – preferably in a table. The times achieved by the members must be sorted with the fastest average at the top of the table and the slowest average at the bottom. When the members are recording solves, I think it would be a good idea to show these times in a graph so that they can view their progress visually. ”

How much data is there?

Dr McIvor: “There are currently 30 members in the club, and the details of each member needs to be calculated. When there is a competition, most of the members compete, and the five times achieved by each member need to be recorded. When the members are using the system, they will regularly be recording the time, penalty, comment, scramble, and solution for solves. Algorithms and scrambles are entered quite rarely, probably a maximum of five of each per week.”

Are there any calculations that need to be made?

Dr McIvor: “Members complete five solves during a competition. An *average* is calculated from these times by removing the fastest and slowest times then calculating the mean of the remaining three times. The results for the competition need to be ranked from fastest *average* to slowest *average*. I would like comprehensive statistics to be calculated, including current average of 5, 12, and 50 and best average of 5, 12, and 50. Each type of average is calculated in the same way: remove the fastest and slowest times and calculate the mean of the remaining times. If there is more than one disqualified time (DNF) then the average is DNF (Did Not Finish).”

Will any personal data be entered into the system?

Dr McIvor: “The only personal data entered in the system will be the members’ details.”

Is the system to be secure?

Dr McIvor: “I will be the only person with access to the member details. If I had an offline digital system then there would be no need for extra security since my computer is password-protected. The only personal data stored is the member details; no personal data is stored regarding the learning process.”

Should there be limits to areas of the system for different users?

Dr McIvor: “No, I would like every user to have access to every part of the system. Only I can access personal information, but the features should be available to the members so that they can organise their own competitions if they want to.”

Should the users have passwords?

Dr McIvor: “Passwords are unnecessary since every member can access all parts of the system.”

Mention some of the specific information on the tutorials.

Dr McIvor: “When I am teaching, I try to have one or two examples of the concept in question, so the new system should be able to provide clearly-explained examples. I then have one or two exercises, which are based on the previously-taught concept, for the members. The system must have interactive features like this which can check whether the user understands the concept, and can gives hints. The method which I teach to the pupils involves four main steps: cross, corners, edges, last layer; teaching users how to solve the cross is difficult since this is the most intuitive part of the process. The system should provide many examples for solving the cross, and explanations of these examples. Solving the corners and edges is fairly mechanical and shouldn’t be difficult for the user to understand, but there are some algorithms that need to be learned here, so the system should provide these algorithms in a simple and readable format. The last layer is also mechanical and fairly easy to understand, but there are even more algorithms needed for this step, so the user should have some facility to view and memorise the algorithms.”

What type of searching needs to be performed?

Dr McIvor: “The students should be able to search for times between lower and upper boundaries.”

Do you require any validation to be performed throughout the system?

Dr McIvor: “It would be helpful if the input was checked against certain rules so that trivial mistakes are avoided. Some data can be checked easily, such as whether I enter the form class correctly or not because there are only 35 possible form classes. For both students and myself, I think that the data should be validated when entering times, since it can be tricky to enter these correctly. Also, dates are easy to enter incorrectly, so these should be validated.”

What hardware/software does the user have?

Dr McIvor: “Most students will have a standard laptop at home with Windows 7. In school, I have an Acer Aspire laptop. The system would need to be able to run on computers with low specifications since I want all members to use the system. I would also prefer if the system is mostly self-contained so that students don’t have to download/install additional software.”

How often will the data change?

Dr McIvor: “The data that I store on my laptop will only change when a member’s information needs updated or when times are being recorded during a competition. The competitions happen once every two weeks. The data on students’ computers will change very regularly, since they will be solving the cube many times and will be recording the data associated with each solve.”

What data needs to be imported/exported from the system?

Dr McIvor: “The students should be able to save/load information about a solve to/from the system. This would include the time, penalty, comment, scramble, and solution for the solve. The members should have the option to save or load states of the cube to their computer.

The members should be able to save the graph of their times as an image. It would be useful if the statistics could be exported in a formatted fashion, such as in a pdf file.”

Existing Documentation

These are extracts from the tables in Dr McIvor's word document in which he stores details pertaining to the club. These extracts of the non-automated system demonstrate the travail required to record the information for just one competition, especially since averages and ranks must be calculated for every single member.

Member Details

Name	Date of Birth	Form Class
Jack Cooper	22/05/2001	9S
Sam Middleton	10/09/1999	10W

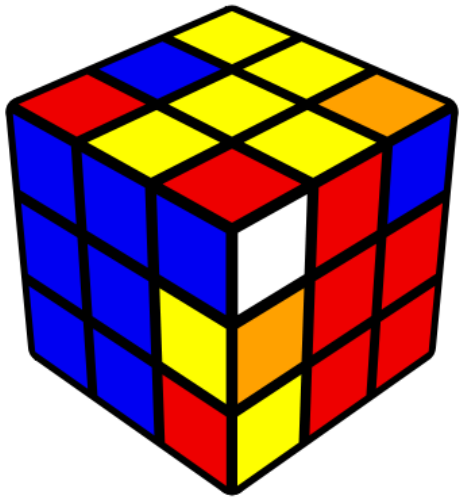
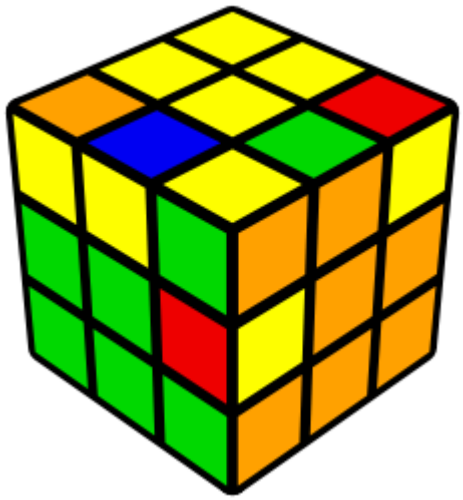
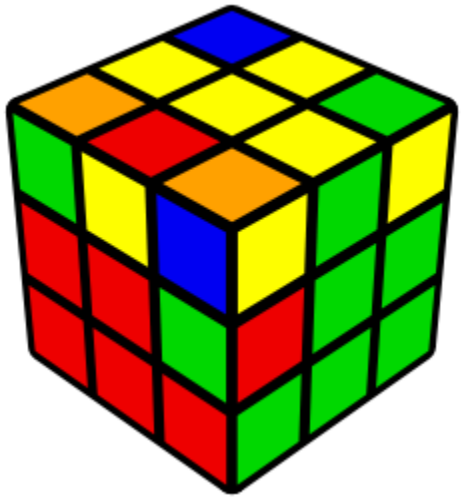
Competition Details

Competition Number	Date	Member Name	Rank
5	18/01/2010	Sarah Rutherford	1
		William Johnston	2
		Joy Montgomery	3
6	01/02/2010	Simon Lane	1
		William Johnston	2
		Sarah Rutherford	3

Competition-Time

Name	Lee Brown
Competition	3
Average of 5	14.49
Solve Number	**Duration**
1	14.56
2	15.32
3	14.24
4	14.66
5	12.30

This is an example of one of the ‘algorithm sheets’ which Dr McIvor gives to new members.

Case	Algorithm
	To solve the white-blue-red corner: R U R’
	To solve the green-orange edge: U’ F’ U F U R U’ R’
	To solve the green-red edge: F’ U F U’ R U2 R’ U2 R U’ R’

Chosen Solution

After considering the potential solutions, I believe that a system developed from scratch using Java and SQLite is the best option. After discussing these ideas with Dr McIvor, he corroborated my decision:

> **Dr McIvor**: "I think a manual system would be insufficient to cover all problems in the current system, so I would prefer a computerised solution. I think a web-based solution is a good idea, but it really isn't necessary since I will be the only person accessing the data, and I will be doing so from a single laptop. I would prefer to have a new system which has a full user-manual and all of the features which I require since I'm not very familiar with the existing software used for database management."

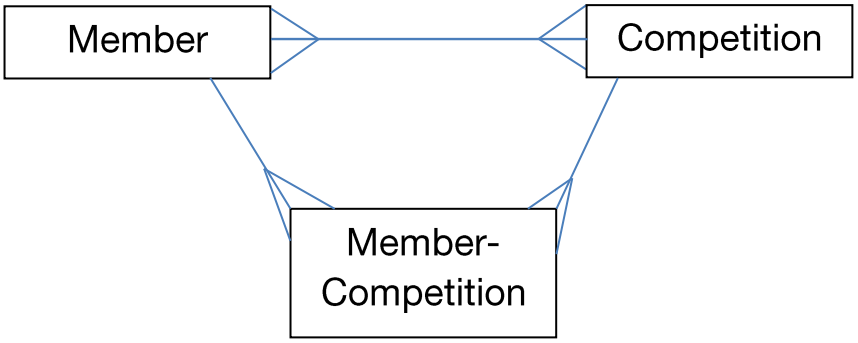
For the main system, I believe that Java is the best choice. It allows me to create easy-to-use GUIs (Graphical User Interfaces) with icons, buttons, menus, images etc. Java is an object-oriented language so I will be able to include features that might not be feasible in other languages. In addition, my knowledge of Java surpasses my knowledge of any other language, which means that my progress will not be hindered by lack of knowledge in another language.

Entity-Relationship Model and Diagrams

Existing System

- A member can compete in many competitions
- A competition can have many competitors

Proposed System



- A member can compete in many competitions
- A competition can have many competitors
- A member can have many member-competitions (which consist of their details/performance for that competition)
- A competition can have many member-competitions (which, in this context, are essentially registrations for that competition)

Identification of Objects and Object-Analysis Diagrams for Object-Oriented Programmed Solutions

Cubie

Fields	Methods
orientation	setOrientation(int orientation) : void getOrientation() : int
stickers	setStickers(Color[] stickers) : void getStickers() : Color[]
cubieIndex	setCubieIndex(int cubieIndex) : void getCubieIndex() : int

Corner(Cubie)

Fields	Methods
INITIAL_CORNERS	getAllInitialStickers() : Color[][] getInitialStickers(int cornerIndex) : Color[]
	setStickers(Color i, Color j, Color k) Overrides(Cubie) : void twist(int direction) : void

Edge(Cubie)

Fields	Methods
INITIAL_EDGES	getAllInitialStickers() : Color[][] getInitialStickers(int edgeIndex) : Color[]
	setStickers(Color i, Color j) Overrides(Cubie) : void flip() : void

Slice

Fields	Methods
edges	setEdges(Edge[] edges) : void getEdges() : Edge[] setEdge(int index, Edge edge) : void getEdge(int index) : Edge flipAllEdges() : void flipEdge(int index) : void swapEdges(int i, int j) : void
corners	setCorners(Corner[] corners) : void getCorners() : Corner[] setCorner(int index, Corner corner) : void getCorner(int index) : Corner twistAllCorners(int direction) : void twistCorner(int index, int direction) : void swapCorners(int i, int j) : void
centre	setCentre(Color centre) : void getCentre() : Color
	performMove(int direction) : void

Cube

Fields	Methods
edges corners	
slices	getSlice(int index) : Slice
	performMove(int direction) : void rotate(String direction) : void updateCubies(int sliceIndex) : void updateAll() : void

Member

Fields	Methods
(Constructor) Member (int memberID, String forenames, String surname, String gender, String dateOfBirth, String email, String formClass)	
memberID	getMemberID() : int setMemberID(int memberID) : void
forenames	getForenames() : String setForenames(String forenames) : void
surname	getSurname() : String setSurname(String surname) : void
gender	getGender() : String setGender(String gender) : void
dateOfBirth	getDateOfBirth() : String setDateOfBirth
email	getEmail() : String setEmail(String email) : void
formClass	getFormClass() : String setFormClass() : void
	isValidEmail(String email) : boolean isValidFormClass(String formClass) : boolean

Competition

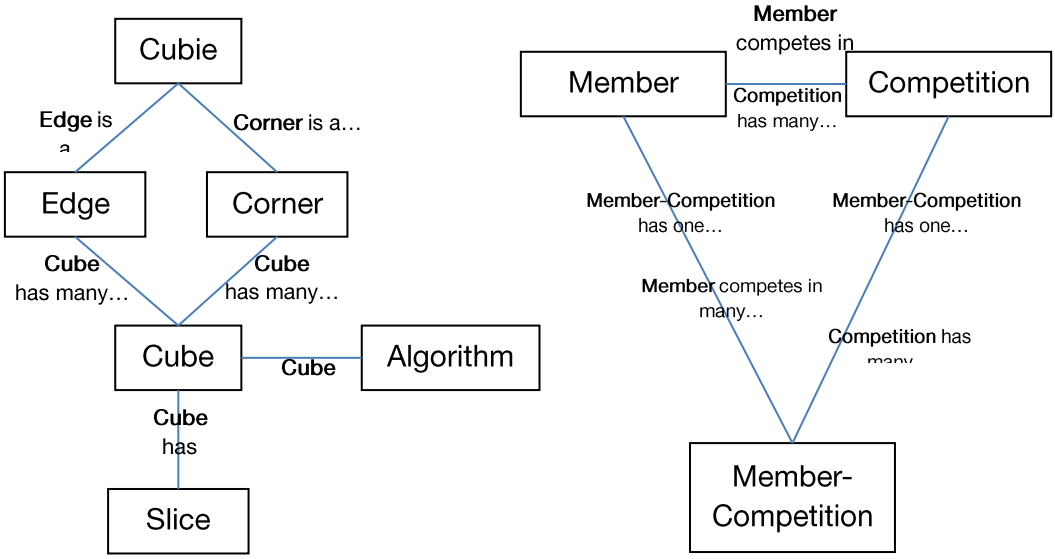
Fields	Methods
(Constructor) Competition (int competitionID, String date)	
competitionID	getID() : int setID(int competitionID) : void
date	getDate() : String setDate(String date) : void

MemberCompetition

Fields	Methods
(Constructor) MemberCompetition (int competition, int memberID, String time1, String time2, String time3, String time4, String time5)	
competitionID	getCompetitionID() : int setCompetitionID(int competitionID) : void
memberID	getMemberID() : int setMemberID(int memberID) : void
times	getTimes() : String[] setTimes(String[] times) : void getAverage() : double isBetterThan(MemberCompetition other) : boolean

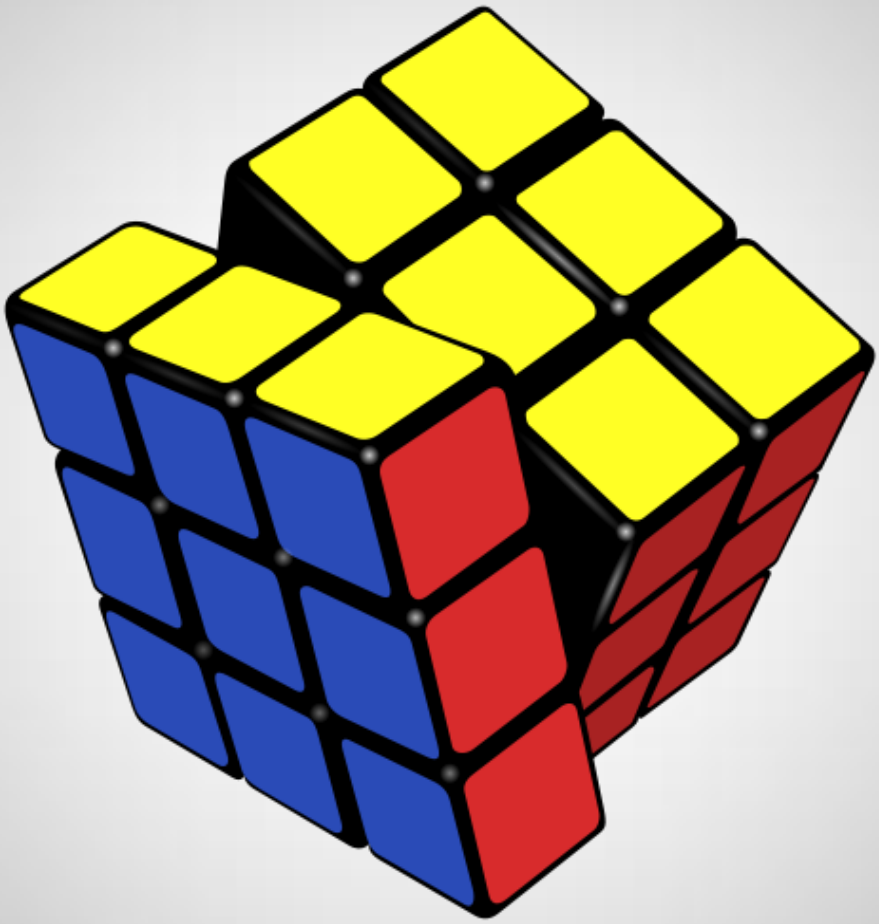
Algorithm

Fields	Methods
(Constructor) Algorithm (int algorithmID, String moveSequence, String comment)	
algorithmID	getAlgorithmID() : int setAlgorithmID(int algorithmID) : void
moveSequence	getMoveSequence() : String setMoveSequence(int moveSequence) : void
comment	getComment() : String setComment(String comment) : void



SECTION 2

DESIGN

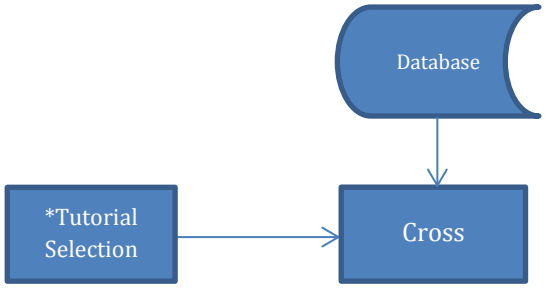


Contents

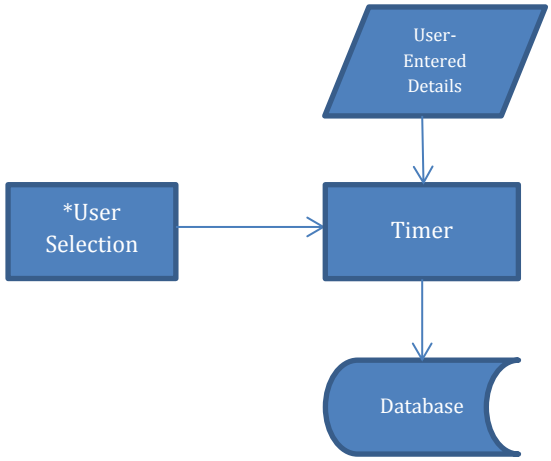
System Flow Charts

Select Tutorial

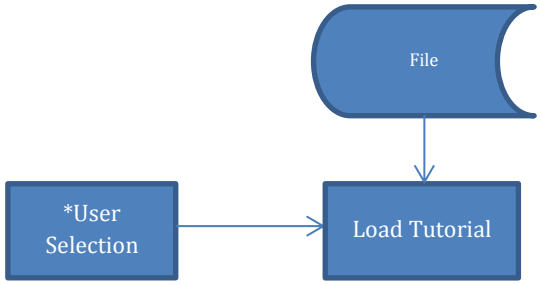
View Sub-Step Tutorial (*other tutorials[1] will follow same structure*)



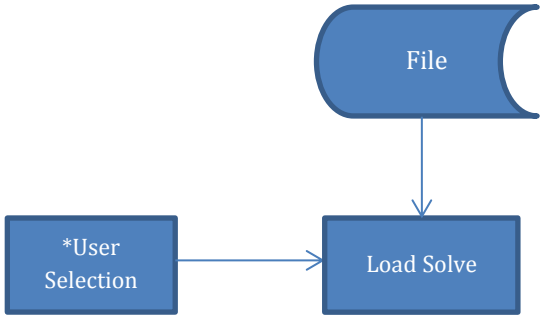
Add New Time



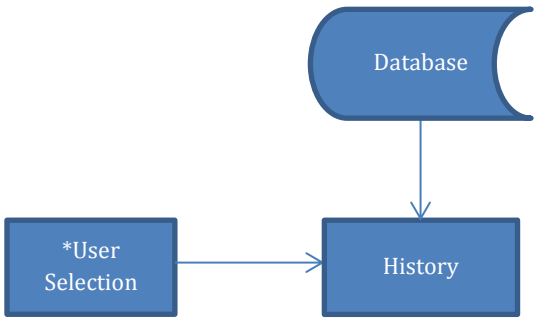
Load Tutorial from File



Load Solve

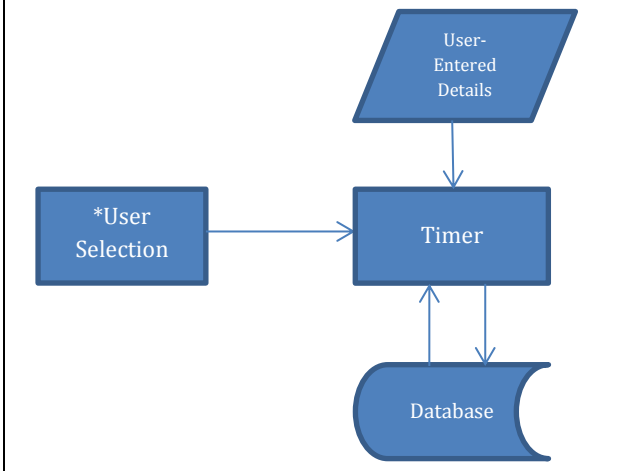


View History of Solves

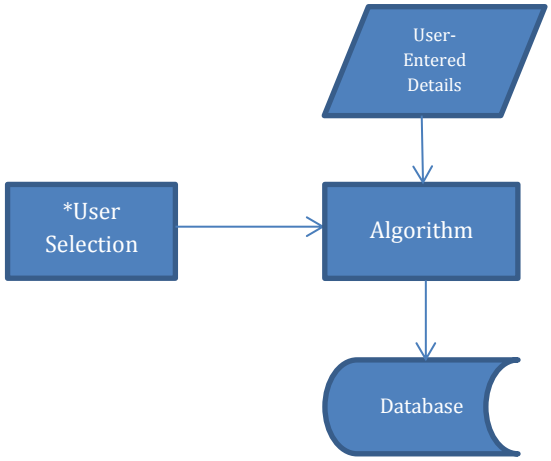


[1] cross, corners, edges, orientation of last layer, permutation of last layer

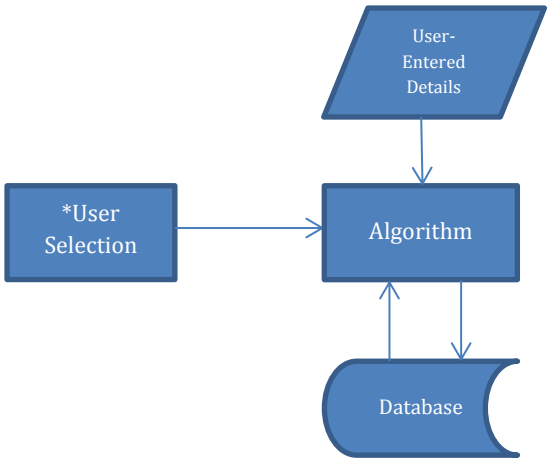
Edit Time



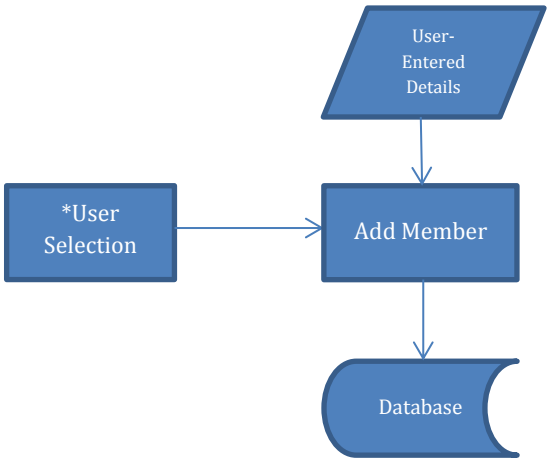
Add Algorithm



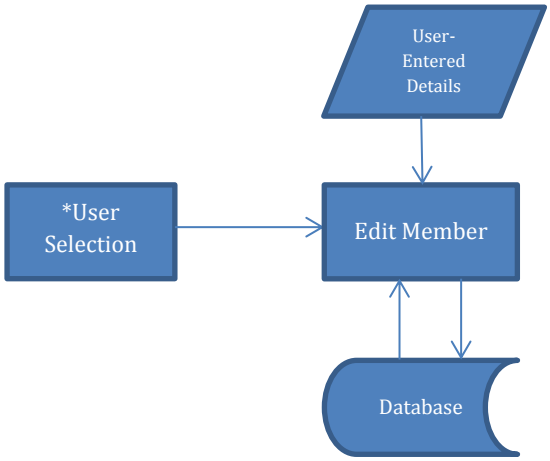
Edit Algorithm



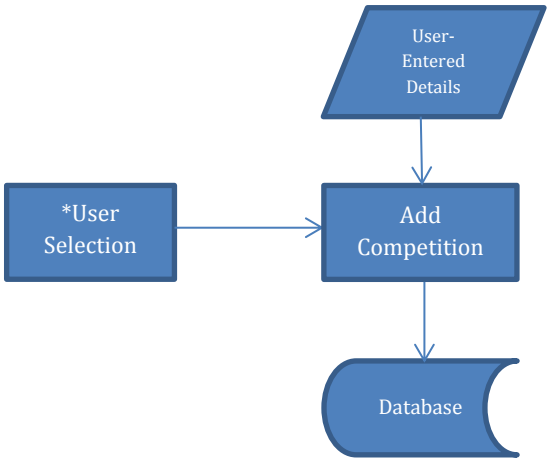
Add Member



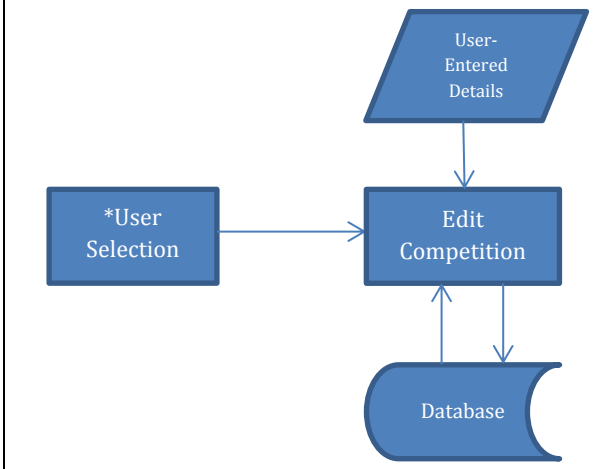
Edit Member



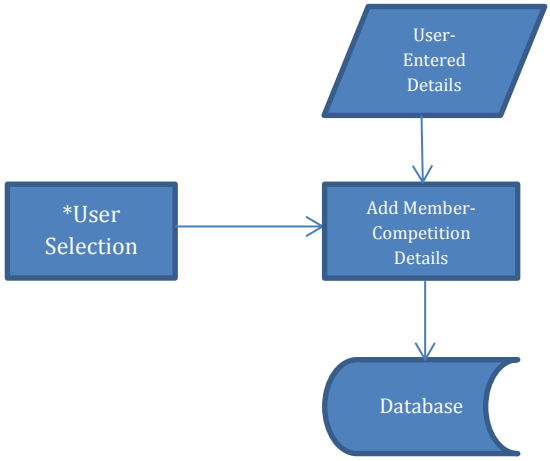
Add Competition



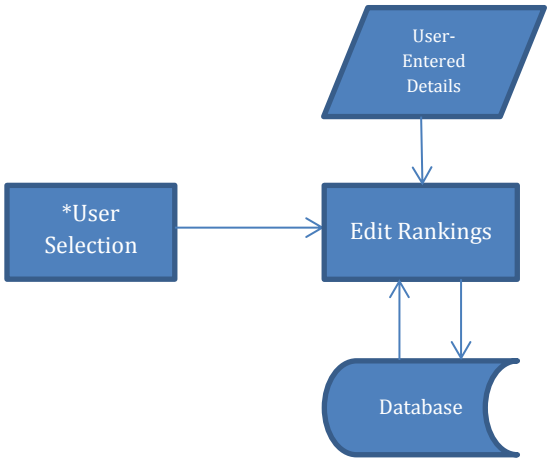
Edit Competition



Add Competition Times

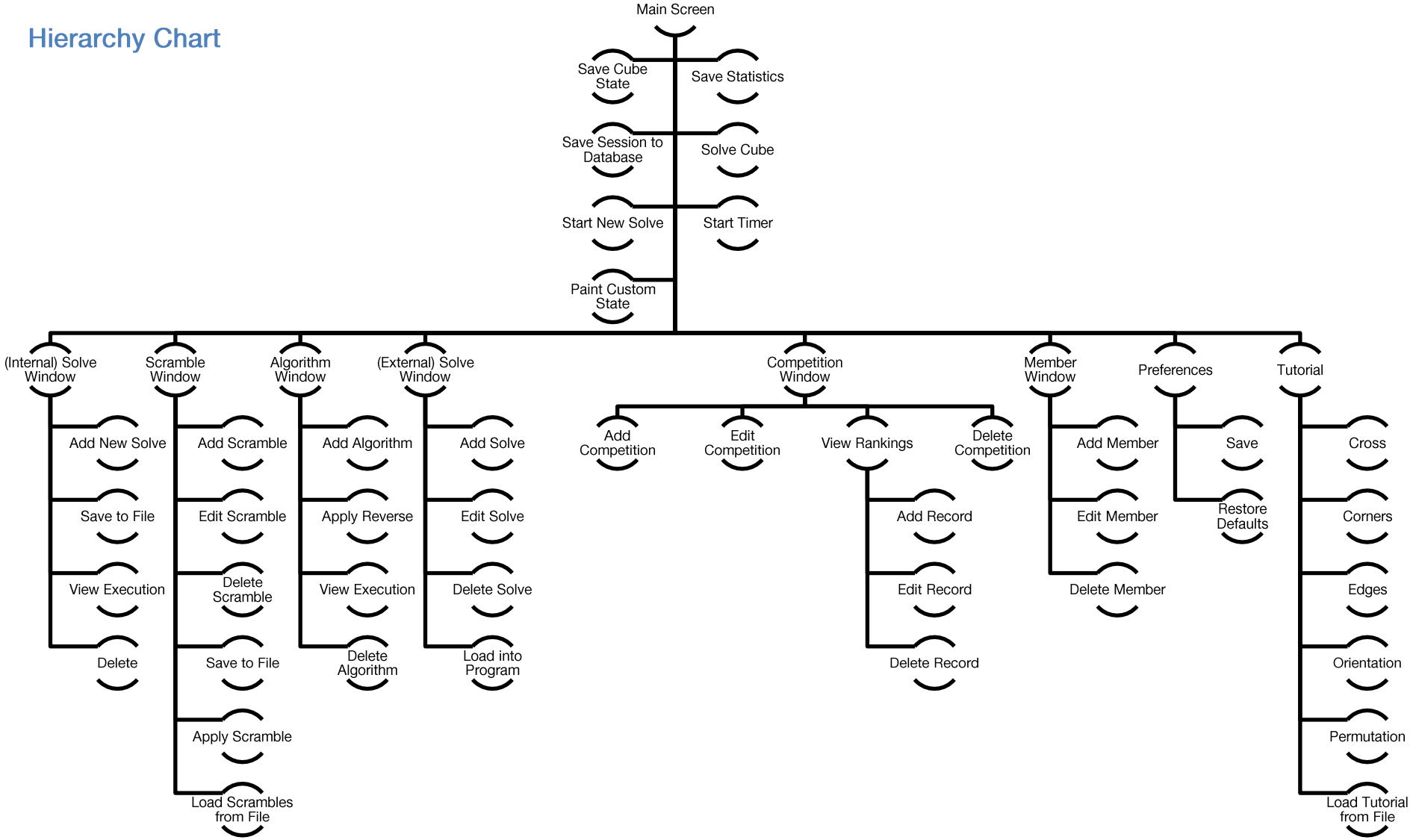


Edit Competition Times

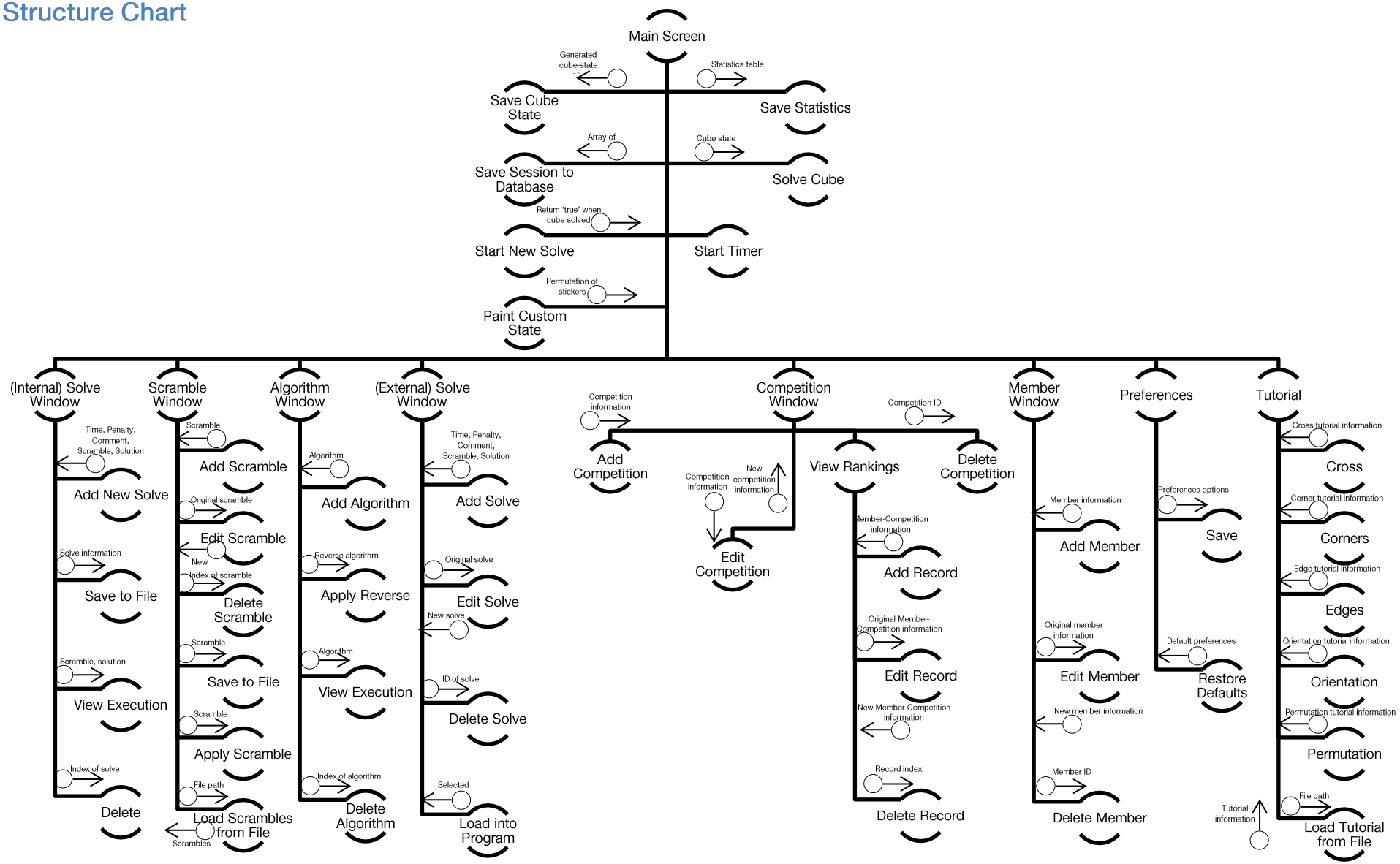


Description of Modular Structure of System

Hierarchy Chart



Structure Chart



Algorithms using Pseudo Code

1. QuickSort Algorithm

This algorithm sorts the elements between the specified indices in the specified array into order so that the least value is at the 'left' of the array and the greatest element is at the 'right'.

Variable	Type	Description
list	Array	Holds the elements to be sorted. The array can be of any type that supports magnitude comparison, e.g. numbers, strings etc.
start	Integer	Holds the index of the left-most element in the range of elements to be sorted.
end	Integer	Holds the index of the right-most element in the range of elements to be sorted.
pivot	Integer	The pivot stores the value of the middle element in the list to be sorted.
i	Integer	Left-Cursor - traverses each element until it reaches the right cursor.
j	Integer	Right-Cursor – traverse each element until it reaches the left cursor.

```
procedure quickSort(list, start, end)
   pivot <- list[(start + end)/2]  //pivot is the middle element in list
   i ← start
   j ← end

   while (i <= j)
      while (list[i] < pivot)
         i ← i + 1
      end while

      while (list[j] > pivot)
         j ← j - 1
      end while

      if (i <= j) then
         swap(list, i, j)   //elements i and j are swapped in list
         i = i ← 1
         j = j ← 1
      end if
   end while

   if (i < end) then
      quickSort(list, i, end)   //recursively sort the right side of list

   if (start < j) then
```

```
      quickSort(list, start, j)//recursively sort left side of list
end procedure
```

2. Perform Move Algorithm

Algorithm will be implemented in 'Slice' class in order to move the cubies in the slice.

Variable	Type	Description
direction	Integer	If a clockwise move is to be performed then the direction should be 1, and -1 if it is an anticlockwise move.
tempEdgeStickers	Array of Color	Temporary holder of the stickers of the first edge in the slice.
tempCornerStickers	Array of Color	Temporary holder of the stickers of the first corner in the slice.
nextIndex	Integer	Holds index of next edge.
end	Integer	Holds the index of the last element to assign a value to.
i	Integer	Traverses each cubie in the slice's list.

```
procedure performMove(direction)
   tempEdgeStickers ← edges[0].getStickers()
   tempCornerStickers ← corners[0].getStickers()
   end ← (4 + direction) MOD 4

   for (i ← 0; i != end; i ← (i - direction + 4) MOD 4)
      nextIndex ← (i - direction + 4) MOD 4
      edges[i].setStickers(edges[nextIndex].getStickers())
      corners[i].setStickers(corners[nextIndex].getStickers())
   end for

   edges[end].setStickers(tempEdgeStickers)
   corners[end].setStickers(tempCornerStickers)
end procedure
```

3. Rotate Cube Algorithm

Algorithm will be implemented in the 'Cube' class in order to rotate the cube, moving the corresponding pieces.

Variable	Type	Description
move	String	"x" – a rotation in the direction of an "R" move. "y" – a rotation in the direction of a "U" move. "z" – a rotation in the direction of an "F" move.
slicesIndex	Array of Integer	Holds the indices of the slices to manipulate. For example, if slicesIndex holds (5, 2, 4, 3), then the permutations will result as follows: 5 → 2 2 → 4 4 → 3 3 → 5
tempSlice	Slice	Holds a Slice object to facilitate the adjustment of permutations.
nextIndex	Integer	Holds the index of the next index in the slicesIndex array.
direction	Integer	Each of the "x", "y", and "z" moves can be inverted. If direction = 1, this represents a clockwise rotation, and direction = -1 represents an anticlockwise rotation.
end	Integer	Holds the last index of the slicesIndex array to use. This variable depends on the direction.
i	Integer	Counter variable.

```
procedure rotate(move)
    if (move = "z") then
        rot("y")
        rot("x'")
        rot("y'")
    else if (move = "z'") then
        rot("y'")
        rot("x'")
        rot("y")
    else
        if (move is a 180 degrees move) then
            rotate(move.substring(0,1))
            rotate(move.substring(0,1))
        else
            rot(move)
    end if
end procedure

procedure rot(move)
    slicesIndex ← array of integer containing indices of slices to manipulate
    tempSlice ← slices[slicesIndex[0]]
```

```
    direction ← (move.contains("'") ? -1 : 1)
    end ← (4 + direction) MOD 4

    for (i ← 0; i != end; i ← (i - direction + 4) MOD 4)
        nextIndex ← (i - direction + 4) MOD 4
        slices[slicesIndex[i]] ← slices[slicesIndex[nextIndex]]
        performRotationMaintenance(move, slicesIndex[i])
        updateCubies(slicesIndex[i])
    end for

    slices[slicesIndex[end]] ← tempSlice
    performRotationMaintenance(move, slicesIndex[end])
    updateCubies(slicesIndex[end])

    switch(move)
        case "x'":
        case "x":
            slices[0].twistAllCorners(1)
            updateCubies(0)
            slices[1].twistAllCorners(-1)
            updateCubies(1)
        end case
        case "y":
        case "y'":
            for (i ← 0; i < 4; i = i + 1)
                slices[slicesIndex[i]].flipEdge(1)
                updateCubies(slicesIndex[i])
            end for
        end case
    end switch

    if (move = "x'") then
        slices[0].performMove(2)
        updateCubies(0)
        slices[1].performMove(2)
        updateCubies(1)
    end if

        updateAll()

end procedure
```

4. Compare Cubies Algorithm

Algorithm will be implemented in the 'Cubie' class in order to compare two cubies to see if they have the same stickers.

Variable	Type	Description
otherCubie	Cubie	This is the cubie which the the current cubie is compared to.
found	boolean	True if a matching sticker is found in otherCubie.
i	Integer	Counter variable.

```
function compareTo(otherCubie)
   if (otherCubie.getStickers().length != this.stickers.length) then
      return -1

   for (i ← 0; i < stickers.length; i ← i + 1)
      found ← false
      for (j ← 0; j < stickers.length; j ← j + 1)
         if (getStickers()[i] = otherCubie.getStickers()[j]) then
            found ← true
            break
         end if
      end for

      if (!found) then
         return -1
      end if
   end for
   return 0
end function
```

5. Update Cubies Algorithm

This algorithm acts as an interface between the cubies of each slice and the all the cubies in the ‘Cube’ class. This means that when one slice is changed, you can change any corresponding slices using the 'updateAll' algorithm.

Variable	Type	Description
sliceIndex	Integer	Holds index of the slice from which the cubies of the ‘Cube’ class get their data.
cEdge	Edge	Holds the current edge object.
cCorner	Corner	Holds the current corner object.
cubieIndex	Integer	Counter variable.

```
procedure updateCubies(sliceIndex)
   for (cubieIndex ← 0; cubieIndex < 4; cubieIndex ← cubieIndex + 1)
      cCorner ← slices[sliceIndex].getCorner(cubieIndex)
      cEdge ← slices[sliceIndex].getEdge(cubieIndex)

      this.corners[cubieIndices[sliceIndex][cubieIndex]] ← cCorner
      this.edges[cubieIndices[sliceIndex][cubieIndex + 4]] ← cEdge
   end for
end procedure
```

6. Twist Corner Algorithm

Changes the orientation of a corner and the cycles the stickers in the corresponding fashion.

Variable	Type	Description
temp	Color	Temporary color holder.
stickersCopy	Array of Color	Stores a copy of the stickers of the corner
direction	Integer	Determines the direction in which the corner stickers are cycled and the change of orientation.

```
procedure twist(direction)
   stickersCopy ← getStickers()
   temp ← stickersCopy[0]

   if (direction > 0) then
      stickersCopy[0] ← stickersCopy[2]
      stickersCopy[2] ← stickersCopy[1]
      stickersCopy[1] ← temp
   else
      stickersCopy[0] ← stickersCopy[1]
      stickersCopy[1] ← stickersCopy[2]
      stickersCopy[2] ← temp
   end if

   setOrientation(((getOrientation() + direction + 4) MOD 3) - 1)
end procedure
```

7. Solve Cross Algorithm

Solves all of the white edges on the cube. When a move is made, it will be recorded and then later simplified.

```
While (!CrossSolved())
    Find E-Slice Edges
    If (Edge found) Then
        If (a Relative-Solved Edge is on White Face) Then
            Move Relative-Solved Edge so that the Working-Edge can be inserted relative-correct, then relative-solve the Working Edge
        Else
            Move EITHER (not both) top or bottom slices until a misoriented edge is in the same slice as the Working-Edge
        End if
    Else
        If (U-Slice contains a flipped cross-edge) Then
            If (D-Slice contains a flipped cross-edge) Then
                Move flipped D-Slice cross edge into same slice as U-Slice edge
            Else
                Perform D move until spot below U-Slice edge is not a cross edge
            End if
        End if
        Move U-Slice edge into E-Slice
    End If
End While
```

8. Solve Corners Algorithm

This algorithm solves all corners containing a white sticker (first-layer corners). This algorithm identifies the unsolved first-layer corners on the cube, then chooses to solve the corner that requires the fewest number of moves to solve.

Variable	Type	Description
solveCandidates	SolveCandidate	This data type can store the index of the corner on the cube, and can store the 'score' for the current corner. The score equals the number of moves needed to solve thecorner.
solveCandidatesIndex	Integer	Holds the index of the current index of the corner under investigation, e.g. the first corner to be examined will have solveCandidatesIndex = 0. Hence, this variable keeps track of how many first-layer corners have been examined.
cornerIndex	Integer	Holds the index of the corner that requires the fewest number of moves to solve.
i	Integer	Counter variable.
corner	Corner	Refactoring variable – holds the corner to be investigated.

```
procedure solveFirstLayerCorners()
      rotateToTop(Color.yellow)

      while (!firstLayerCornersSolved())
         solveCandidatesIndex ← 0

         for (i ← 0; i < 8; i ← i + 1)
            corner ← cube.getCorner(i)
            if (isFLCorner(corner) && (!pieceSolved(corner))) then
               solveCandidates[solveCandidatesIndex].index ← i
                                 solveCandidates[solveCandidatesIndex].score ← getScore(i)
               solveCandidatesIndex ← solveCandidatesIndex + 1
            end if
         end for
         solveCorner(solveCandidates[getIndexOfMinScore(solveCandidatesIndex)].index)
      end while
end procedure
```

9. Solve Individual Corner Algorithm

This algorithm solves the corner at the specified index on the cube and records the moves to solve the corner.

Variable	Type	Description
corner	Corner	The corner to be solved.
orientation	Integer	Stores the orientation of the corner to be solved.
overCornerIndex	Integer	Stores the index of the position on the cube where the corner should be moved so as to 'setup' the corner so that it can be solved.
currentIndex	Integer	The initial index of the corner on the cube.

```
procedure solveCorner(currentIndex)
    corner ← cube.getCorner(currentIndex)
    orientation ← corner.getOrientation()
    overCornerIndex ← getIndexOfDestination(corner)
    overCornerIndex += (overCornerIndex MOD 2 = 0) ? -3 : -5

    if (isPieceSolved(corner)) then
        return
    end if

    while (cube.getCorner((currentIndex >= 4) ? 7 : 2).compareTo(corner) = -1)
        cube.rotate("y")
    end while

    if (currentIndex >= 4) then
        if (orientation > 0)
            cube.performAbsoluteMoves("R U' R'")
        else
            cube.performAbsoluteMoves("R U R' U'")
        end if
        solveCorner(2)
    else if (currentIndex = overCornerIndex) then
        if (orientation = 0)
            cube.performAbsoluteMoves("R U2 R' U' R U R'")
        else if (orientation = 1) then
            cube.performAbsoluteMoves("R U R'")
        else
            cube.performAbsoluteMoves("F' U' F")
        end if
    else
        for (i ← 0; i < ((overCornerIndex - currentIndex) + 4) MOD 4; i ← i + 1)
            cube.performAbsoluteMoves("U")
```

```
      end for

      for (i ← 0; i < ((overCornerIndex - currentIndex) + 4) MOD 4; i ← i + 1)
         cube.rotate("y'")
      end for
      solveCorner(2)

   end if
end procedure
```

10. Solve Edges Algorithm

This algorithm solves all edges that do not contain a white or yellow sticker (middle-layer edges). This algorithm identifies the unsolved middle-layer edges on the cube, then chooses to solve the edge that requires the fewest number of moves to solve.

Variable	Type	Description
solveCandidates	SolveCandidate	This data type can store the index of the edge on the cube, and can store the 'score' for the current edge. The score equals the number of moves needed to solve the edge.
solveCandidatesIndex	Integer	Holds the index of the current index of the edge under investigation, e.g. the first edge to be examined will have solveCandidatesIndex = 0. Hence, this variable keeps track of how many middle-layer edges have been examined.
edgeIndex	Integer	Holds the index of the edge that requires the fewest number of moves to solve.
i	Integer	Counter variable.
edge	Edge	Refactoring variable – holds the edge to be investigated.

```
procedure solveMiddleLayerEdges()
      rotateToTop(Color.yellow)

      while (!middleLayerEdgesSolved())
         solveCandidatesIndex ← 0

         for (i ← 0; i < 8; i ← i + 1)
            corner ← cube.getEdge(i)
            if (isMLEdge(edge) && (!pieceSolved(edge))) then
               solveCandidates[solveCandidatesIndex].index ← i
                                 solveCandidates[solveCandidatesIndex].score ← getScore(i)
               solveCandidatesIndex ← solveCandidatesIndex + 1
            end if
         end for
         solveEdge(solveCandidates[getIndexOfMinScore(solveCandidatesIndex)].index)
      end while
end procedure
```

11. Solve Individual Edge Algorithm

This algorithm is called by the 'solveMiddleLayerEdges' algorithm in order to solve each middle-layer edge.

Variable	Type	Description
edge	Edge	Holds the edge being solved.
currentIndex	Integer	Stores the index on the cube of the edge being solved.

```
procedure solveEdge(currentIndex)
   rotateToTop(Color.yellow)
   edge ← new Edge(cube.getEdge(currentIndex).getStickers())

   if (isPieceSolved(edge)) then
      return
   end if

   if (currentIndex < 4) then
      while (!edgeInSetupPosition(edge))
         cube.performAbsoluteMoves("U")
      end while

      while (cube.getEdge(2).compareTo(edge) = -1)
         cube.rotate("y")
      end while

      if (edge.getStickers()[0].equals(cube.getSlice(2).getCentre())) then
         cube.performAbsoluteMoves("U R U' R' U' F' U F")
      else
         cube.performAbsoluteMoves("U' L' U L U F U' F'")
      end if

   else
      while (cube.getEdge(6).compareTo(edge) = -1)
         cube.rotate("y")
      end while
      cube.performAbsoluteMoves("R U' R' U' F' U F")
      solveEdge(0)
   end if
end procedure
```

12. Solve Edge Orientation Algorithm

This algorithm performs the moves required to solve the orientation of the edges of the last layer, i.e. all yellow stickers are facing upwards.

Variable	Type	Description
moves	String	Stores the moves that the cube will perform.
numEdgesOriented	Integer	Stores the number of edge cubies oriented before the orientation is solve.

```
procedure solveEdgeOrientation()
      rotateToTop(Color.yellow)
   numEdgesOriented ← getNumOriented()
   moves ← ""

   if (numEdgesOriented < 4) then
      if (numEdgesOriented = 0) then
         moves ← "F R U R' U' F' U2 F U R U' R' F'"
      else
         while ((cube.getEdge(2).getOrientation() = 0) || (cube.getEdge(3).getOrientation() = 1))
            cube.performAbsoluteMoves("U")
         end while
      end if
         if (cube.getEdge(0).getOrientation() = 1) then
            moves ← "F R U R' U' F'"
         else
            moves ← "F U R U' R' F'"
         end if
      end if

      cube.performAbsoluteMoves(moves)

   end if
end procedure
```

13. Solve Corner Orientation Algorithm

This algorithm performs the moves required to solve the orientation of the corners of the last layer, i.e. all yellow stickers are facing upwards.

Variable	Type	Description
count	Integer	All four corners need to be examined, so the for loop executes four times. Count acts as a stepper variable.
orientation	Integer	Stores the orientation {-1, 0, 1} of the corner being examined.

```
procedure solveCornerOrientation()
   for (count ← 0 count < 4 count ← count + 1)
      moves ← ""
      orientation ← cube.getCorner(2).getOrientation()
      if (orientation = 1) then
         moves ← "R' D' R D R' D' R D"
      else if (orientation = -1) then
         moves ← ("D' R' D R D' R' D R")
      end if

      cube.performAbsoluteMoves(moves + "U")
   end for
end procedure
```

14. Solve Corner Permutation Algorithm

This algorithm performs the moves required to solve the permutation of the corners of the last layer.

Variable	Type	Description
count	Integer	This algorithm looks for 'headlights' (matching corner stickers on the same face); if it cannot find any headlights, then *count* will equal 4, in which case, it will have to perform a different sequence of moves. The value of count after the for loop determines the sequence of moves performed.

```
procedure solveCornerPermutation()
      count ← 0

      if (cornersSolved()) then
         return

      while ((!headlightsAtBack()) && (count < 4))
         count ← count + 1
         cube.performAbsoluteMoves("U")
      end while

      if (count < 4) then
         cube.performAbsoluteMoves("L' U R' D2 R U' R' D2 R L")
      else
        cube.performAbsoluteMoves("x' R U' R' D R U R' D' R U R' D R U' R' D' x")
      end if
end procedure
```

15. Solve Edge Permutation Algorithm

This algorithm performs the moves required to solve the permutation of the edges of the last layer.

Variable	Type	Description
count	Integer	This algorithm looks for a block of three matching stickers on one face. If the block is found, count will be less than four, in which case, an anti/clockwise three-cycle will be performed; if not, then another sequence of moves will be performed.
moves	String	Stores the moves to be performed.

```
procedure solveEdgePermutation()
      count ← 0
      moves ← ""

      if (edgesSolved()) then
         return

      while ((count < 4)
            && (!cube.getEdge(0).getSecondaryColor()
                              .equals(cube.getCorner(0).getStickers()[2])))
         count ← count + 1
         cube.performAbsoluteMoves("U")
      end while

      if (count < 4) then
         if (edgesAreOpposite(cube.getEdge(1), cube.getEdge(2))) then
            moves ← ("R U' R U R U R U' R' U' R2")
         else
            moves ← ("R2 U R U R' U' R' U' R' U R'")
         end if
      else if ((cube.getEdge(0).getSecondaryColor().equals(cube.getCorner(3).getStickers()[1]))
            && (cube.getEdge(1).getSecondaryColor().equals(cube.getCorner(3).getStickers()[2])))
         moves ← ("M2 U M2 U2 M2 U M2")
      else
         if (!cube.getEdge(2).getSecondaryColor().equals(cube.getCorner(1).getStickers()[2])) then
            moves ← "U "
         end if

         moves ← moves + "M2 U M2 U M' U2 M2 U2 M'"
```

```
      end if

      cube.performAbsoluteMoves(moves)
end procedure
```

16. Strict Compare Cubies Algorithm

This algorithm is similar to the compareTo algorithm, but this algorithm checks that the order of the stickers is the same as well. E.g. the compareTo algorithm would return '0' with two corners {green, red, blue} and {green, blue, red}, but this algorithm would return -1 since the order of the stickers is not the same. Note: this algorithm would return '0' for {green, red, blue} and {red, blue, green} since the stickers are in the same order.

Variable	Type	Description
current	Color	Stores the first color in the stickersCopy array and acts as a temporary variable in order to swap the elements of the stickersCopy array.
length	Integer	Stores the number of stickers on the otherCubie (2 for edges, 3 for corners).
stickersCopy	Array of Color	Used as a copy of the stickers of otherCubie.
matching	Boolean	If the stickers of the two cubies match, then this will be set as true.
otherCubie	Cubie	The current instance of the cubie is being compared to 'otherCubie'
i	Integer	If the cubie has *x* stickers, then *x* different cycles of stickers need to be checked, so 'i' is used to execute the for loop *x* times.
j	Integer	Iterates over each stickers of the cubie.

```
function strictCompareTo(otherCubie) : Integer
   stickersCopy ← copyOf(otherCubie.getStickers(), otherCubie.getStickers().length)
   length ← stickersCopy.length

   if (length != this.stickers.length) then
      return -1
   end if

   for (i ← 0; i < length; i ← i + 1)
      current ← stickersCopy[0]
      matching ← true

      for (j ← 0; j < length; j ← j + 1)
         if (!stickersCopy[j].equals(this.stickers[j])) then
            matching ← false
            break
         end if
      end for

      if (matching) then
         return 0
      end if
```

```
      for (j ← 0; j < length - 1; j ← j + 1)
         stickersCopy[j] ← stickersCopy[j + 1]
      end for

      stickersCopy[length - 1] ← current

   end for

   return -1
end function
```

17. Linear Search Corner Orientation Algorithm

This acts as a linear search, but if the result is 2, then it returns -1.

Variable	Type	Description
list	Array of Color	The list of elements to be searched.
element	Color	The element for which the algorithm is searching.
i	Integer	Iterates over each element of the list.

```
function linearSearchCornerOrientation(list, element) : Integer
   for (i ← 0; i < list.length; i ← i + 1)
      if (list[i].equals(element)) then
         return (i = 2) ? -1 : i
   end for
   return -2   //Not found
end function
```

18. Are edges opposite (permutation) Algorithm

This algorithm returns true if the secondary colors (i.e. not white and not yellow) of the parameterised edges correlate to opposite centres on the cube.

Variable	Type	Description
colorOne	Color	The secondary color of the first edge.
colorTwo	Color	The secondary color of the second edge.
one	Integer	The index of colorOne in the topEdges array.
two	Integer	The index of colorTwo in the topEdges array.
topEdges	Array of Edge	Holds the four cross edges in order to get the secondary colors.
edgeOne	Edge	The secondary color of this edge is compared to edgeTwo in order to determine whether or not they are opposite.
edgeTwo	Edge	The secondary color of this edge is compared to edgeOne in order to determine whether or not they are opposite.

```
function edgesAreOpposite(edgeOne, edgeTwo) : boolean
   colorOne ← edgeOne.getSecondaryColor()
   colorTwo ← edgeTwo.getSecondaryColor()
      one ← -1, two ← -2

   for (int i ← 0; i < 4; i ← i + 1)
      if (topEdges[i].getSecondaryColor().equals(colorOne)) then
         one ← i
      else if (topEdges[i].getSecondaryColor().equals(colorTwo)) then
         two ← i
      end if
   end for

   return ((one + two) MOD 2 = 0)
end function
```

19. Is Valid Time Algorithm

This algorithm uses a regular expression to check whether or not the time parameter is valid. Valid times are:

- MM:SS.
- MM:S.
- M:SS.
- M:S.
- SS.
- S.
- DNF

There can be any number of numbers after the decimal point. I decided that a decimal point *must* be included since the probability of getting a time in the form '*xx*.00' is $\frac{1}{100}$ whereas the probability of *accidentally* forgetting to include the digits after the decimal point is quite high.

Variable	Type	Description
time	String	Holds the time to be tested. Example, "12.30"

```
function isValidTime(String time) : boolean
   if (time = null) then
      return false
   else if (time.equalsIgnoreCase("DNF")) then
      return true
   else if (!time.matches("(\\d{1,2}:)?\\d{1,2}\\.\\d*")) then
      return false
   else if (getFormattedStringToDouble(time) > 3599.59) then
      return false
   else
      return true
end function
```

20. Get Padded Time Algorithm

This algorithm converts a string in a format such as “1:1.5” to “1:01.50”. It adds leading and trailing zeros in order to get the time in to a more appropriate format.

Variable	Type	Description
time	String	Holds the unpadded time to be converted.
paddedTime	String	The string to be returned.
indexOfColon	Integer	Stores the index of the colon in ‘time’. Using this variable, leading zeros can be removed.
indexOfPeriod	Integer	Stores the index of the ‘.’ in ‘time’. Using this variable, leading zeros can be added between the colon and the decimal point.
i	Integer	Counter variable used to control the number of iterations of the for loop in order to append the correct number of trailing zeros.
j	Integer	Traverses ‘time’.
end	Integer	Holds the index of the end of the string to be checked.

```
function getPaddedTime(time) : String
    j = 0
    if ((indexOfColon = time.indexOf(":")) != -1) then
        while (time.substring(j, j + 1).equals("0"))
            j = j + 1
        end while
        if (j < indexOfColon) then
            paddedTime += time.substring(j, indexOfColon + 1)
        end if
    end if

    indexOfPeriod = time.indexOf(".")

    if (indexOfColon != -1) then
        for (int i = 0; i < 3 - indexOfPeriod + indexOfColon; i = i + 1)
            paddedTime += "0"
        end for
    end if

    paddedTime += time.substring(indexOfColon + 1, indexOfPeriod + 1)
    paddedTime += time.substring(indexOfPeriod + 1)
    end = 3 - time.length() + indexOfPeriod
    for (i = 0; i < end; i = i + 1)
        paddedTime += "0"
    end for

    return paddedTime
end function
```

21. Get Formatted String to Double Algorithm

This algorithm returns the number of seconds represented by a formatted time-string, e.g. given “1:10.50”, it would return 70.5 since “1:10.50” represents 60 seconds + 10 seconds + 50 hundredths of a second.

Variable	Type	Description
index	Integer	Traverses ‘time’.
result	Double	Used as an accumulator of minute, second, and hundredth of a second values.
time	String	The time of which the value is being found.

```
function getFormattedStringToDouble(time) : double
   if (time.equalsIgnoreCase("DNF")) then
      return -1

   index ← 0
   result ← 0

   if ((index ← time.indexOf(":")) != -1) then
      result += 60 * Integer.valueOf("0" + time.substring(0, index))
   end if

   result += Double.parseDouble("0" + time.substring(index + 1))

   return result
end function
```

22. Get Seconds to Formatted String Algorithm

Converts a real number to a formatted string in the aforementioned format. E.g. it would convert 100.43 to “1:40.43”.

Variable	Type	Description
seconds	double	The number of seconds which needs to be converted to the formatted string.
formattedString	String	The string to be returned.

```
function getSecondsToFormattedString(seconds) : String
   formattedString ← Integer.toString((int)seconds/60)

   seconds ← seconds MOD 60
   formattedString += ":" + String.format("%.02f", seconds)

   return getPaddedTime(formattedString)
end function
```

23. Rotate Color to Top Algorithm

Rotates the cube so that the ‘color’ parameter is on the top.

Variable	Type	Description
color	Color	The color that should end up on the top face.

```
procedure rotateToTop(color)
   if (getSliceIndexOfCentre(color) = 0) then
      return
   end if

   for (int i ← 0; i < 4; i ← i + 1)
      cube.rotate("x")
      if (getSliceIndexOfCentre(color) = 0) then
         return
      end if

      cube.rotate("z")
      if (getSliceIndexOfCentre(color) = 0) then
         return
      end if
   end for
end procedure
```

24. Simplify Cross Solution Algorithm

This algorithm removes all rotations from the solution for the cross and modifies the moves so that the solution is still valid. E.g. “y R U” would be changed to “B U”.

Variable	Type	Description
current	String	Holds the current move being examined
i	Integer	The rotations must be applied to move without other rotations, so the first for loop traverses starting at the penultimate element then starts at the previous index etc. During each iteration, the moves after the rotation are modified then the rotation is removed.
j	Integer	Traverse the moves after the rotation being examined.
originalMoves	LinkedList<String>	The moves to be simplified.

```
procedure simplifyCross(originalMoves)
   for (i ← originalMoves.size() - 2; i >= 0; i ← i - 1)
      current ← originalMoves.get(i)

      if (current.substring(0, 1).equals("y") && (current.length() <= 2)) then
         for (j ← i + 1; j < originalMoves.size(); j ← j + 1)
            originalMoves.set(j, applyRotationToMove(current, originalMoves.get(j)))
         end for

         originalMoves.remove(i)
      end if
   end for
end procedure
```

25. Apply Rotation to Move Algorithm

This algorithm is needed to complete the previous algorithm (Simplify Cross Algorithm).

Variable	Type	Description
rotation	String	A string representing the rotation - “x”, “y”, or “z”.
move	String	The move to which the rotation is being applied.
movePairings	Array of String	This stores strings representing the result of applying the rotation. The algorithm could have been hard-coded without movePairings, but using this array makes debugging easier etc.
offset	Integer	If the move is anti-clockwise, then offset is 1, otherwise it is 0.
index	Integer	Used in order to identify the resulting move if ‘move’ is a 180° move.

```
function applyRotationToMove(rotation, move) : String
   movePairings ← { "D", "U", "R", "L", "B", "F"}
   offset ← (rotation.contains("'") ? 1 : 0)

   if (!rotation.contains("2")) then
      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
         end switch
      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
         end switch
      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
         end switch
      end switch
   else
      switch (rotation.substring(0, 1))
      case "x":
         if ("LR".contains(move.substring(0, 1))) then
            return move
      case "y":
         if ("UD".contains(move.substring(0, 1))) then
            return move
      case "z":
         if ("FB".contains(move.substring(0, 1))) then
            return move
      end switch

      index ← LinearSearch.linearSearch(movePairings, move.substring(0, 1))
      return movePairings[index + ((index % 2 = 0) ? 1 : -1)]
   end if

   return "-1"
end function
```

26. Cancel Moves Algorithm

This algorithm cancels moves and simplifies moves.
Examples:

- R R → R2
- R R R → R'
- F R U U' R' → F

Variable	Type	Description
index	Integer	Traverses the originalMoves list.
combination	String	Used to identify the type of cancellation. If 'combination' = -1, then the moves cancel (e.g. R R'), otherwise, then the combination is appended to the move (e.g. R R → R2).
one	String	Holds the first element being compared.
two	String	Holds the second element being compared.
moveType	String	Holds the type of move, e.g. if the move is "R2" then the move type is "R".
originalMoves	LinkedList<String>	The list of moves to be simplified.

```
procedure cancelMoves(originalMoves)
   index ← 0
   while (index < (originalMoves.size() - 1))
      one ← originalMoves.get(index).trim()
      two ← originalMoves.get(index + 1).trim()

      if (one.substring(0, 1).equals(two.substring(0, 1))) then
         moveType ← one.substring(0, 1)
         one ← one.substring(1)
         two ← two.substring(1)

         combination ← getCombination(one, two)
         if (combination.equals("NOT_MATCHING")) then
            index ← index + 1
         else
            if (combination.equals("-1")) then
               originalMoves.remove(index)
               originalMoves.remove(index)
            else
               originalMoves.remove(index + 1)
               originalMoves.set(index, moveType + combination)
            end if

            index ← (index = 0) ? 0 : index - 1
```

```
         end if
      else
         index ← index + 1
      end if
   end while

   for (int i ← 0; i < originalMoves.size() - 2; i ← i + 1)
      if ((originalMoves.get(i).equals("x")) then
            && (originalMoves.get(i+1).equals("z"))
            && (originalMoves.get(i+2).equals("x")))
         originalMoves.remove(i)
         originalMoves.set(i, "z2")
         originalMoves.set(i + 1, "y'")
      end if
   end for
end procedure
```

27. Get Move Combination Algorithm

This algorithm returns the type of combination (used by the previous algorithm, 'Cancel Moves Algorithm').

Variable	Type	Description
one	String	The first move to be compared.
two	String	The second move to be compared.

```
function getCombination(one, two) : String
   if (((one.startsWith("w")) || (two.startsWith("w")))) then
      try
         if (!one.substring(0,1).equals(two.substring(0,1))) then
            return "NOT_MATCHING"
         else
            one ← one.substring(1)
            two ← two.substring(1)
         end if

      catch (IndexOutOfBoundsException e)
         return "NOT_MATCHING"
      end try
   end if

   if ((one.equals("")) && (two.equals(""))) then
      return "2"
   else if (((one.equals("")) && (two.equals("'"))) || ((one.equals("'")) && (two.equals("")))) then
      return "-1"
   else if (((one.equals("")) && (two.equals("2"))) || ((one.equals("2")) && (two.equals("")))) then
      return "'"
   else if ((one.equals("'")) && (two.equals("'"))) then
      return "2"
   else if (((one.equals("'")) && (two.equals("2"))) || ((one.equals("2")) && (two.equals("'")))) then
      return ""
   else
      return "-1"
end function
```

28. Simplify Corner/Edge Solution Algorithm

This algorithm deals with a solution that has “y” rotations and “U” moves so that they cancel properly. For example, y U y U is the same as y2 U2 because the directions are colinear.

Variable	Type	Description
temp	String	Used as a temporary store so that two elements of ‘originalMoves’ can be swapped.
i	Integer	Traverses ‘originalMoves’ so that the appropriate moves can be compared.
originalMoves	LinkedList<String>	The list of moves to be simplified.

```
procedure simplifyCornerEdgeSolution(LinkedList<String> originalMoves)
   cancelMoves(originalMoves)

   for (i ← originalMoves.size() - 3; i >= 0; i ← i - 1)
      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))))) then
         temp ← originalMoves.get(i)
         originalMoves.set(i, originalMoves.get(i + 1))
         originalMoves.set(i + 1, temp)
         cancelMoves(originalMoves)
      end if
   end for
end procedure
```

29. Load Tutorial Algorithm

This algorithm loads a tutorial from a text file and stores the information contained in the file. The tutorial text file has standard format, but the number of *hints* and *optimal solutions* may vary, so this algorithm needs to handle all forms of tutorial.

Variable	Type	Description
filePath	String	The file path of the tutorial to be loaded.

```
procedure loadTutorial(filePath)
   while (!End of File) do
      Read and assign scramble and description.
      if (Pieces are expected to be solved) then
         Read and assign each of the expected-to-be-solved pieces
         Read and assign hints.
         Read and assign optimal solutions.
         Read and assign explanation.
      end if
   end while
end procedure
```

30. Get Number of Moves without Rotations Algorithm

This algorithm finds the number of moves in a sequence of moves that are not rotations. E.g. given “y R z B”, the algorithm would return ‘2’ since there are only two non-rotation moves in the sequence of moves.

Variable	Type	Description
moves	String	The moves to be examined.
numMoves	Integer	Accumulates the number of moves.
i	Integer	Used as a counter to traverse each character of ‘moves’.

```
function getNumMovesWithoutRotations(moves) : Integer
   numMoves ← 0

   for (i ← 0; i < moves.length(); i ← i + 1)
      if ("UDRLFBM".contains(moves.substring(i, i+1))) then
         numMoves ← numMoves + 1
   end for

   return numMoves
end function
```

31. Is Valid Cube State Algorithm

This algorithm determines whether or not the cube is in a state that can be solved. This method will be called when the user wants to modify the stickers on the cube in order to paint a custom state. It first checks that each cubie is a valid cubie (i.e. there exists a {white, red, green} corner, and there aren't any incorrect cubies such as a {white, white} edge). If any of the cubies are invalid (or there are duplicates) then the method returns false. If all cubies are valid, it then attempts to solve the cube. After the solving algorithms have completed, if the cube is not solved, then that means it is in an unsolvable state and the method returns false.

Variable	Type	Description
validCornerStickers	Array of Array of Color	An array of valid corner sticker permutations, e.g. an element of validCornerStickers is {yellow, red, blue}.
validEdgeStickers	Array of Array of Color	An array of valid sticker permutations, e.g. an element of validEdgeStickers is {blue, red}.
validCorners	Array of Corner	An array of valid corners using the stickers in validCornerStickers.
validEdges	Array of Edge	An array of valid edges using the stickers in validEdgeStickers.
isValidCubeState	boolean	If the cube is in a valid state then this will be true, otherwise it will be false.
cornersFound	Array of boolean	When a valid corner is found, its index is stored in here so that duplicates can be identified later. For example, if the {white, red, green} corner is found, cornersFound[2] will be set as true. This means that if another {white, red, green} corner is found, it will be checked against this array and it will find that this corner has already beend found, so that method will return false.
edgesFound	Array of boolean	Same as above but for edges.
validCubieIndex	Integer	Returns the index of the cubie on a solved cube with white top and green front.
i	Integer	Used as a counter to traverse the elements of the arrays.

```
function isValidCubeState() : boolean
   validCornerStickers ← Corner.getAllInitialStickers()
   validEdgeStickers ← Edge.getAllInitialStickers()
   validCorners ← new Corner[8]
   validEdges ← new Edge[12]
   isValidCubeState ← true
   cornersFound ← new boolean[8]
   edgesFound ← new boolean[12]
   validCubieIndex

   for (i ← 0; i < 8; i ← i + 1)
      validCorners[i] ← new Corner(validCornerStickers[i])
      cornersFound[i] ← false
   end for

   for (i ← 0; i < 12; i ← i + 1)
```

```
      validEdges[i] ← new Edge(validEdgeStickers[i])
      edgesFound[i] ← false
   end for

   for (i ← 0; i < 8; i ← i + 1)
      validCubieIndex ← getValidCubieIndex(cube.getCorner(i), validCorners)

      if ((validCubieIndex = -1) || (cornersFound[validCubieIndex])) then
         isValidCubeState ← false
      end if
      else
         cornersFound[validCubieIndex] ← true
      end if
   end for

   for (i ← 0; i < 12; i ← i + 1)
      validCubieIndex ← getValidCubieIndex(cube.getEdge(i), validEdges)

      if ((validCubieIndex = -1) || (edgesFound[validCubieIndex])) then
         isValidCubeState ← false
      else
         edgesFound[validCubieIndex] ← true
      end if
   end for

   if (!isValidCubeState) then
      return false

   *Solve Cube*

   for (i ← 0; i < 4; i ← i + 1)
      if (!crossSolver.isPieceSolved(cube.getCorner(i))) then
         isValidCubeState ← false
      end if
      if (!crossSolver.isPieceSolved(cube.getEdge(i))) then
         isValidCubeState ← false
      end if
   end for

   return isValidCubeState
end function
```

32. Get Valid Cubie Index Algorithm

This algorithm returns the index of the specified cubie on a solved cubed with white on top and green on front.

Variable	Type	Description
cubie	Cubie	The cubie whose index needs to be found.
validCubies	Array of Cubie	The list of valid cubies in which the index of ‘cubie’ needs to be found.
i	Integer	Used as a counter to traverse validCubies.

```
function getValidCubieIndex(cubie, validCubies) : Integer
   for (i ← 0; i < validCubies.length; i ← i + 1)
      if (cubie.strictCompareTo(validCubies[i]) = 0) then
         return i
   end for

   return -1
end function
```

33. Assign Facelet Painting Colors Algorithm

This assigns the colours to be painted on screen.

Variable	Type	Description
sliceIndices	Array of Integer	The top face, front face, and right face are painted on screen, and these faces' indices are {0, 4, 2}.
currentSlice	Slice	The current slice from which colors are being obtained.
cornerIndex	Integer	The index of the current corner from which the colors are being obtained.
edgeIndex	Integer	The index of the current edge from which the colors are being obtained.
i	Integer	Used to iterate over each row of the current slice.
j	Integer	Used to iterate over each sticker in the current row of the current slice.
faceColors	Three dimensional array of Color	faceColors[x] stores the colors to be painted on a chosen face. faceColors[x][x] stores the colors to be painted on a particular face in a chosen row. faceColors[x][x][x] stores the color to be painted on the chosen face, in the chosen row, in the chosen column.

```
procedure assignFaceletPaintingColors()
   sliceIndices = {0, 4, 2}

   currentSlice = cube.getSlice(sliceIndices[0])
   cornerIndex = 0
   edgeIndex = 0
   for (i = 0; i < 3; i = i + 1)
      for (j = 0; j < 3; j = j + 1)
         if ((i = 1) && (j = 1)) then
            faceletColors[0][i][j] = currentSlice.getCentre()
         else if ((i + j) % 2 = 0) then
            faceletColors[0][i][j] = currentSlice.getCorner(cornerPaintOrder[cornerIndex++]).getStickers()[0]
         else
            faceletColors[0][i][j] = currentSlice.getEdge(edgePaintOrder[edgeIndex++]).getStickers()[0]
      end for
   end for

   currentSlice = cube.getSlice(sliceIndices[1])
   cornerIndex = 0
   edgeIndex = 0
   for (i = 0; i < 3; i = i + 1)
      for (int j = 0; j < 3; j = j + 1)
         if ((i = 1) && (j = 1)) then
            faceletColors[1][i][j] = currentSlice.getCentre()
         else if ((i + j) MOD 2 = 0) then
```

```
            faceletColors[1][i][j] = currentSlice.getCorner(
                  cornerPaintOrder[cornerIndex]).getStickers()[(cornerPaintOrder[cornerIndex++] MOD 2 = 0) ? 1 : 2]
         else
            faceletColors[1][i][j] = currentSlice.getEdge(edgePaintOrder[edgeIndex]).getStickers()[(edgePaintOrder[edgeIndex++]
MOD 2 = 0) ? 1 : 0]
      end for
   end for

   currentSlice = cube.getSlice(sliceIndices[2])
   cornerIndex = 0
   edgeIndex = 0
   for (int i = 0; i < 3; i = i + 1)
      for (int j = 0; j < 3; j = j + 1)
         if ((i = 1) && (j = 1)) then
            faceletColors[2][i][j] = currentSlice.getCentre()
         else if ((i + j) MOD 2 = 0) then
            faceletColors[2][i][j] =
currentSlice.getCorner(cornerPaintOrder[cornerIndex]).getStickers()[(cornerPaintOrder[cornerIndex++] MOD 2 = 0) ? 1 : 2]
         else
            faceletColors[2][i][j] = currentSlice.getEdge(edgePaintOrder[edgeIndex++]).getStickers()[1]
      end for
   end for
end procedure
```

34. Catalog Moves Algorithm

This algorithm records each of the the moves specified in the argument. For example, “R U R’ U’” would be recorded as “R”, “U”, “R’”, “U’”.

Variable	Type	Description
moves	String	This string stores the moves to be recorded.
index	Integer	Iterates over each character in the string.
indexOfSpace	Integer	Used to store the index of the next ‘ ‘ character in the string.
remainingMoves	String	Stores the moves to be catalog

```
procedure catalogMoves(String moves)
   if (moves = null) then
      return
   end if

   catMoves(moves, 0)
end procedure

procedure catMoves(String moves, int index)
   if (index >= moves.length) then
      return
   end if

   remainingMoves = moves.substring(index).trim()
   indexOfSpace = remainingMoves.indexOf(" ")

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

   catalogMove(remainingMoves.substring(0, indexOfSpace))
   catMoves(moves, index + indexOfSpace + 1) //Recursively records moves
end procedure
```

35. Is Better Than Algorithm

This algorithm compares the current average to another average. An average is better than another average if the 'average of 5' is faster, or the fastest time of the current average 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

Variable	Type	Description
other	MemberCompetition	The average to which the current average is being compared.
thisAverage	double	A numerical representation of the current average. For example, if the times were {1:01.00, 1:02.00, 1:03.00, 1:04.00, 1:05.00}, then `thisAverage = 63.0`
otherAverage	double	A numerical representation of the other average.
list1	Array of double	An array holding a numerical representation of each time in the current average, e.g. if the times were {1:05.00, 1:02.00, 1:04.00, 1:03.00, 1:01.00}, then `list1 = {65.0, 62.0, 64.0, 63.0, 61.0}`
list2	Array of double	An array holding a numerical representation of each time in the other average.
i	Integer	Stepper variable; iterates over each element in `list1` and `list2`.

```
function isBetterThan(other) : function
   thisAverage ← this.get2DPAverage()
   otherAverage ← other.get2DPAverage()

   if (thisAverage = -1) then
      thisAverage ← inf
   end if
   if (otherAverage = -1) then
      otherAverage ← inf
   end if

   if (thisAverage < otherAverage) then
      return true
   else if (otherAverage < thisAverage) then
      return false
   end if

   list1 ← this.getNumericTimeArray()
   list2 ← other.getNumericTimeArray()

   for (i ← 0; i < 5; i ← i + 1)
      if (list1[i] = -1) then
         list1[i] ← inf
      end if
      if (list2[i] = -1) then
```

```
            list2[i] ← inf
        end if
    end for

    quicksort(list1)
    quicksort(list2)

    for (i ← 0; i < 5; i ← i + 1)
        if (list1[i] < list2[i]) then
            return true
        else if (list2[i] < list1[i]) then
            return false
        end if
    end for

    return false
end function
```

36. ‘Get Average Of’ Algorithm

This algorithm calculates the mean of the middle (size – 2) times. For example, the average of {0.0, 10.00, 11.00, 12.00, 20.00} would be 11.00 because (10 + 11 + 12)/3 = 11.00 and the fastest (0.0) and slowest (20.00) times are ignored. It returns -1 if there is more than one DNF time in the times.

Variable	Type	Description
times	LinkedList<Solve>	The times whose average is going to be found.
x	Integer	The size of the average, e.g. to get an average of 5, x would be 5.
raw	Array of double	An array holding a numerical representation of the times.
size	Integer	Holds the size of the times list.
acc	double	An accumulator variable that holds the sum of the times.
factor	double	The dividing factor used to find the mean.
start	Integer	Holds an index to find the starting element of the average.
end	Integer	Holds an index to find the ending element of the average.
i	Integer	Stepper variable to iterate over each element in the `raw` array.
numDNF	Integer	Holds the number of DNF times (-1.0 times) in the average.

```
function getAverageOf(x, times) : double
   size ← times.size()
   numDNF ← getNumDNF(x, times)

   if ((size < x) OR (x < 5) || (getNumDNF(x, times) > 1)) then
      return -1
   end if

   acc ← 0
   factor ← x - 2

   start ← 1
   end ← x - 1

   for (i ← 0; i < x; i ← i + 1)
      raw[i] ← times.get(size - x + i).getNumericTime()
   end for

   quicksort(raw)

   if (numDNF > 0) then
      start ← start + 1
      end ← end + 1
   end if
```

```
   for (int i ← start; i < end; i ← i + 1)
      acc ← acc + raw[i]
   end for

   return (acc/factor)
end function
```

SQL Algorithms

General statements

- `CREATE TABLE tableName(variables);`
- `DROP TABLE IF EXISTS tableName;`
- `ALTER TABLE tableName RENAME TO newTableName;`
- `SELECT * FROM tableName;`

Algorithm Table

- **Initialise Table**

```
DROP TABLE IF EXISTS algorithm;
CREATE TABLE algorithm(
   algorithmID INTEGER PRIMARY KEY AUTOINCREMENT,
   moveSequence TEXT,
   comment TEXT
);
for (i ← 0; i < PRESET_ALGORITHMS.length; i ← i + 1) do
      INSERT INTO algorithm(moveSequence, comment)
      VALUES (PRESET_ALGORITHMS[i].moveSequence,
      PRESET_ALGORITHMS[i].comment)
end for
```

- **Reset algorithm table's IDs**

```
rs = SELECT * FROM algorithm;
DROP TABLE IF EXISTS algorithmCopy;
CREATE TABLE algorithmCopy(
   algorithmID INTEGER PRIMARY KEY AUTOINCREMENT,
   moveSequence TEXT,
   comment TEXT
);
while rs has records do
   INSERT INTO algorithmCopy(moveSequence, comment)
   VALUES (rs.moveSequence, rs.comment);
end while

DROP TABLE algorithm;
ALTER TABLE algorithmCopy RENAME TO algorithm;
```

- **Update a record in the algorithm table**

```
UPDATE algorithm
SET moveSequence = getMoveSequenceFromCurrentRowInTable,
comment = getCommentFromCurrentRowInTable
WHERE algorithmID = getAlgorithmIDFromCurrentRowInTable;
```

- **Insert a new record into the table**

```
INSERT INTO algorithm (moveSequence, comment)
VALUES ("", "");
```

- **Delete all selected rows in the table from the database (except preset algorithms)**

```
for (i ← 0; i < selectedIndices.length; i ← i + 1) do
   if (selectedIndices[i] >= NumPresetAlgorithms)
      DELETE FROM algorithm
      WHERE algorithmID = getAlgorithmIDFromCurrentRowInTable;
   end if
end for
```

Competition Table

- **Initialise competition table**

```
DROP TABLE IF EXISTS competition;
CREATE TABLE competition(
   competitionID PRIMARY KEY AUTOINCREMENT,
   competitionDate TEXT
);
```

- **Update a record in the competition table**

```
UPDATE competition
SET competitionDate = getCompetitionDateFromCurrentRowInTable
WHERE competitionID = getCompetitionIDFromCurrentRowInTable;
```

- **Insert a new record into the competition table**

```
INSERT INTO competition (competitionDate)
VALUES (“”);
```

- **Delete a record from the competition table**

```
DELETE FROM competition
WHERE competitionID = getCompetitionIDFromCurrentRowInTable;
```

- **Get the most recent competition ID to put in the table**

```
SELECT * FROM competition ORDER BY competitionID DESC LIMIT 1
```

Solve Table

- **Initialise solve table**

```
DROP TABLE IF EXISTS solve;
CREATE TABLE solve(
   solveID INTEGER PRIMARY KEY AUTOINCREMENT,
   solveTime TEXT,
   penalty TEXT,
   comment TEXT,
   scramble TEXT,
   solution TEXT,
   dateAdded TEXT
);
```

- **Update a record in the solve table**

```
UPDATE solve
SET solveTime = getSolveTimeFromCurrentRowInTable,
time1 = getTime1FromCurrentRowInTable,
time2 = getTime2FromCurrentRowInTable,
time3 = getTime3FromCurrentRowInTable,
time4 = getTime4FromCurrentRowInTable,
time5 = getTime5FromCurrentRowInTable
WHERE solveID = getSolveIDFromCurrentRowInTable;
```

- **Insert a blank record into the solve table**

```
INSERT INTO solve (solveTime, penalty, comment, scramble, solution,
dateAdded)
VALUES ("", "", "", "", "", "");
```

- **Delete a record from the solve table**

```
DELETE FROM solve
WHERE solveID = getSolveIDFromCurrentRowInTable;
```

- **Add all solves from the main timer window to the solve table**

```
for (i ← solves.size() - 1; i >= 0; i ← i - 1) do
currentSolve = solves.get(i)

INSERT INTO solve(solveTime, penalty, comment, scramble, solution,
dateAdded)
VALUES (currentSolve.getStringTime(), currentSolve.getPenalty(),
currentSolve.getComment(), currentSolve.getScramble(),
currentSolve.getSolution(), getCurrentTimeInSQLFormat());
end for
```

Member-Competition Table

- **Initialise member-competition table**

```
DROP TABLE IF EXISTS memberCompetition;
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)
);
```

- **Update a record in the member-competition table**

```
UPDATE memberCompetition
SET time1 = getTime1FromTable,
time2 = getTime2FromTable,
time3 = getTime3FromTable,
time4 = getTime4FromTable,
time5 = getTime5FromTable
WHERE memberID = getMemberIDFromTable
AND competitionID = currentCompetitionID;
```

- **Insert a new record into the member-competition table**

```
INSERT INTO memberCompetition (competitionID, memberID, time1, time2,
time3, time4, time5)
VALUES (currentCompetitionID, getCompetitionIDFromTable, "", "", "", "",
"");
```

- **Delete the selected record from the table**

```
DELETE FROM memberCompetition
WHERE competitionID = currentCompetitionID
AND memberID = getMemberIDFromTable;
```

Member Table

- Initialise member table

```
DROP TABLE IF EXISTS member;
CREATE TABLE member(
   memberID INTEGER PRIMARY KEY AUTOINCREMENT,
   forenames TEXT,
   surname TEXT,
   gender TEXT,
   dateOfBirth TEXT,
   email TEXT,
   formClass TEXT
);
```

- Update a record in the member table

```
UPDATE member
SET forenames = getForenameEnteredInForm,
surname = getSurnameEnteredInForm,
gender = getGenderSelectedInForm,
dateOfBirth = getDateOfBirthEnteredInForm,
email = getEmailEnteredInForm,
formClass = getFormClassEnteredInForm
WHERE memberID = getMemberIDFromCurrentRowInTable;
```

- Insert a new record into the member table

```
INSERT INTO member (forenames, surname, gender, dateOfBirth, email,
formClass)
VALUES ("", "", "", "", "", "");
```

- Delete a record from the member table

```
DELETE FROM member
WHERE memberID = getMemberIDFromCurrentRowInTable;
```

- Get the most recent member ID to put in the table

```
SELECT * FROM member ORDER BY memberID DESC LIMIT 1
```

Class Definitions

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Algorithm

Fields	Methods
(Constructor) Algorithm (int algorithmId, String moveSequence, String comment)	
private algorithmID	public getAlgorithmID
private moveSequence	public setAlgorithmID
private comment	public getMoveSequence()
	public setMoveSequence
	public getComment()
	public setComment(String comment)

CrossSolver(SolveMaster)

Fields	Methods
(Constructor) CrossSolver(Cube cube)	
private static crossEdges	public solveCross()
	private edgesAreOpposite
	private sliceContainsCrossEdgeOriented
	public isCrossEdge
	private getExpectedOffset
	private getIndexOfCrossEdgeInSlice
	public isCrossSolved

EdgeSolver(SolveMaster)

Fields	Methods
(Constructor) EdgeSolver(Cube cube)	
private static mLEdges	public solveMiddleLayerEdges
private solveCandidates	public solveEdge
	private edgeInSetupPosition
	public isMLEdge
	public middleLayerEdgesSolved
	private getScore
	private getIndexOfMinScore

Main

Fields	Methods
private timeGraph	public refreshTimeGraph
private statistics	public refreshStatistics
private solves	public addSolveToList
private solveMaster	public getFormattedCubeSolution
private incTimer	private cancelSolve
private cube	public isValidCubeState
private crossSolver	private getValidCubieIndex
private cornerSolver	public assignOrientationsToCubies
private edgeSolver	private assignFaceletPaintingColors
private orientationSolver	public performRealTimeSolving
private permutationSolver	public isCubeSolved
private tutorial	private resetCube
private selectionSolver	private endSolve
private colorSelection	public refreshTimeList
private preferencesPopUp	public copyAllTimesToDisplay
private scramblePopUp	
private algorithmDatabasePopUp	
private solveDatabasePopUp	
private competitionDatabasePopUp	
private memberDatabasePopUp	
private faceletColors	
private edgePaintOrder	
private cornerPaintOrder	
private numTimesToGraph	
private inspectionTimer	
private realTimeSolutionTimer	
private elapsedTimingSeconds	
private elapsedMinutes	
private movesToBeRecorded	
private timerHasPermissionToStart	
private cubeSolved	
private movesAllowed	
private timeToBeRecorded	
private tutorialIsRunning	
private inspectionTimeRemaining	
private currentScramble	
private currentPenalty	
private timeListPopUp	
private clickToSolve	
private customPaintingInProgress	
private customTimerRunning	
private timingDisplayInterval	
private trackingMoves	
private mouseSelectionSolver	

MouseSelectionSolver(SolveMaster)

Fields	Methods
(Constructor) MouseSelection(Cube cube, CrossSolver a, CornerSolver b, EdgeSolver c, OrientationSolver d, PermutationSolver e)	
private solveMaster	public getSolution
private crossSolver	public solvePiece
private cornerSolver	public getIndexOfPieceOnScreen
private edgeSolver	public getIndexOfFaceletOnScreen
private orientationSolver	public getQuestionDialogResponse
private permutationSolver	
private solution	

CornerSolver(SolveMaster)

Fields	Methods
(Constructor) CornerSolver(Cube cube)	
private solveCandidates private static fLCorners	public solveFirstLayerCorners private lastMoveWasU public solveCorner public isFLCorner public firstLayerCornersSolved private getScore private getIndexOfMinScore

OrientationSolver(SolveMaster)

Fields	Methods
(Constructor) OrientationSolver(Cube cube)	
	public solveOrientation private solveEdgeOrientation private solveCornerOrientation private getNumOriented private isEdgeOriented public isOrientationSolved public isCornerOrientationSolved public isEdgeOrientationSolved

PermutationSolver(SolveMaster)

Fields	Methods
(Constructor) PermutationSolver(Cube cube)	
	public solvePermutation private performAUF private solveCornerPermutation private solveEdgePermutation public isEdgePermutationSolved private edgesAreOpposite public isCornerPermutationSolved public permutationSolved

Solve

Fields	Methods
(Constructor) Solve(String time, String penalty, String comment)	
(Constructor) Solve(String time, String penalty, String comment, String scramble, String solution)	
(Constructor) Solve(String time, String penalty, String comment, String scramble, LinkedList<String> solution)	
private time private comment private penalty private scramble private solution	public setStringTime public getStringTime public setComment public getComment public setPenalty public getPenalty public setScramble public getScramble public setSolution public getSolution public isValidTime public getPaddedTime public getFormattedStringToDouble public getSecondsToFormattedString public isValidDate

SolveMaster

Fields	Methods
(Constructor) SolveMaster(Cube cube)	
private solveMoves	public getReverseStringMoves
protected solutionExplanation	public getCatalogMoves
protected cube	public clearMoves
protected sliceEdgeSharing	public getSolutionExplanation
protected sliceCornerSharing	public getIndexOf
	public rotateToTop
	public rotateToTopFront
	public simplifyMoves
	private simplifyCross
	private applyRotationToMove
	private cancelMoves
	private getCombination
	protected isPieceSolved
	protected newPieceSolved
	private getSliceIndexOfCentre
	public isPieceInCorrectPosition
	protected getIndexOfDestination
	private getShortestOffset
	public getNumTrailingU
	public getKeyToMove
	public isValidMove
	public getStateString
	public applyStateString
	public getNumMoves

Sorter

Fields	Methods
	public sortByTime
	public sortByDateAdded
	private swap
	public reverseArray
	public sortByAverageThenTime

Statistics

Fields	Methods
(Constructor) Statistics(LinkedList<Solve> times)	
private times	public setTimes
private static averages	public getOverallMean
	public getFormattedStandardStatisticsString
	public getAverageOf
	public getFormattedAverage
	private sortByDNF
	public getBestAverageOf
	public getNumDNF

Tutorial

Fields	Methods
(Constructor) Tutorial(Cube cube)	
private scrambles	public getScramble
private descriptions	public getDescription
private expectedSolvedPieces	public criteriaFilled
private hints	public getHint
private optimalSolutions	public getOptimalSolutions
private explanations	public getOptimalSolutionLength
private hintIndex	public getExplanation
private subTutorialIndex	public loadNextHint
private numSubTutorials	public loadPreviousHint
private tutorialLoaded	public loadNextSubTutorial
private tutorialsRequiringUserAction	public loadPreviousSubTutorial
private cube	public requiresUserAction
private crossSolver	public getSubTutorialIndex
private private cornerSolver	public isFirstSubTutorial
private edgeSolver	public isLastSubTutorial
private orientationSolver	public isLoaded
private permutationSolver	private getNumSubTutorials
	public loadTutorial
	private getStickers
	public getNumMovesWithoutRotations

Competition

Fields	Methods
(Constructor) Competition(int competitionID, String date)	
private competitionID	public getID
private date	public setID
	public getDate
	public setDate

Member

Fields	Methods
(Constructor) Member(int id, String forenames, String surname, String gender, String dob, String email, String formClass)	
private memberID	public getMemberID
private forenames	public setMemberID
private surname	public getForenames
private gender	public setForenames
private dateOfBirth	public getSurname
private email	public setSurname
private formClass	public getGender
	public setGender
	public getDateOfBirth
	public setDateOfBirth
	public getEmail
	public setEmail
	public getFormClass
	public setEmail

MemberCompetition

Fields	Methods
(Constructor) MemberCompetition(int competitionID, int memberID, String[] times)	
(Constructor) MemberCompetition(int competitionID, int memberID, String time1, String time2, String time3, String time4, String time5)	
private competitionID private memberID private times	public getCompetitionID public setCompetitionID public getMemberID public setMemberID public getTimes public getAverage public get2DPAverage public getNumericTimeArray public isBetterThan

SolveDBType(Solve)

Fields	Methods
(Constructor) SolveDBType(String time, String penalty, String scramble, String solution, String dateAdded)	
private dateAdded private getID	public getDateAdded public getID

Corner(Cubie)

Fields	Methods
(Constructor) Corner(Color[] stickers)	
(Constructor) Corner(Color zero, Color, one, Color two)	
(Constructor) Corner(Corner corner)	
private static final INITIAL_CORNERS	public getAllInitialStickers public getInitialStickers public setStickers public twist

Cubie

Fields	Methods
(Constructor) Corner(Color[] stickers)	
(Constructor) Corner(Color zero, Color, one, Color two)	
(Constructor) Corner(Corner corner)	
private stickers private cubieIndex private orientation	public setStickers public getStickers public setSticker public setCubieIndex public getCubieIndex public setOrientation public getOrientation public compareTo public strictCompareTo public getColorToWord public getWordToColor

Edge(Cubie)

Fields	Methods
(Constructor) Edge(Color[] stickers)	
(Constructor) Edge(Color zero, Color, one)	
(Constructor) Edge(Edge edge)	
private static final INITIAL_EDGES	public getAllInitialStickers public getInitialStickers public setStickers public flipStickers public flipOrientation public getSecondaryColor

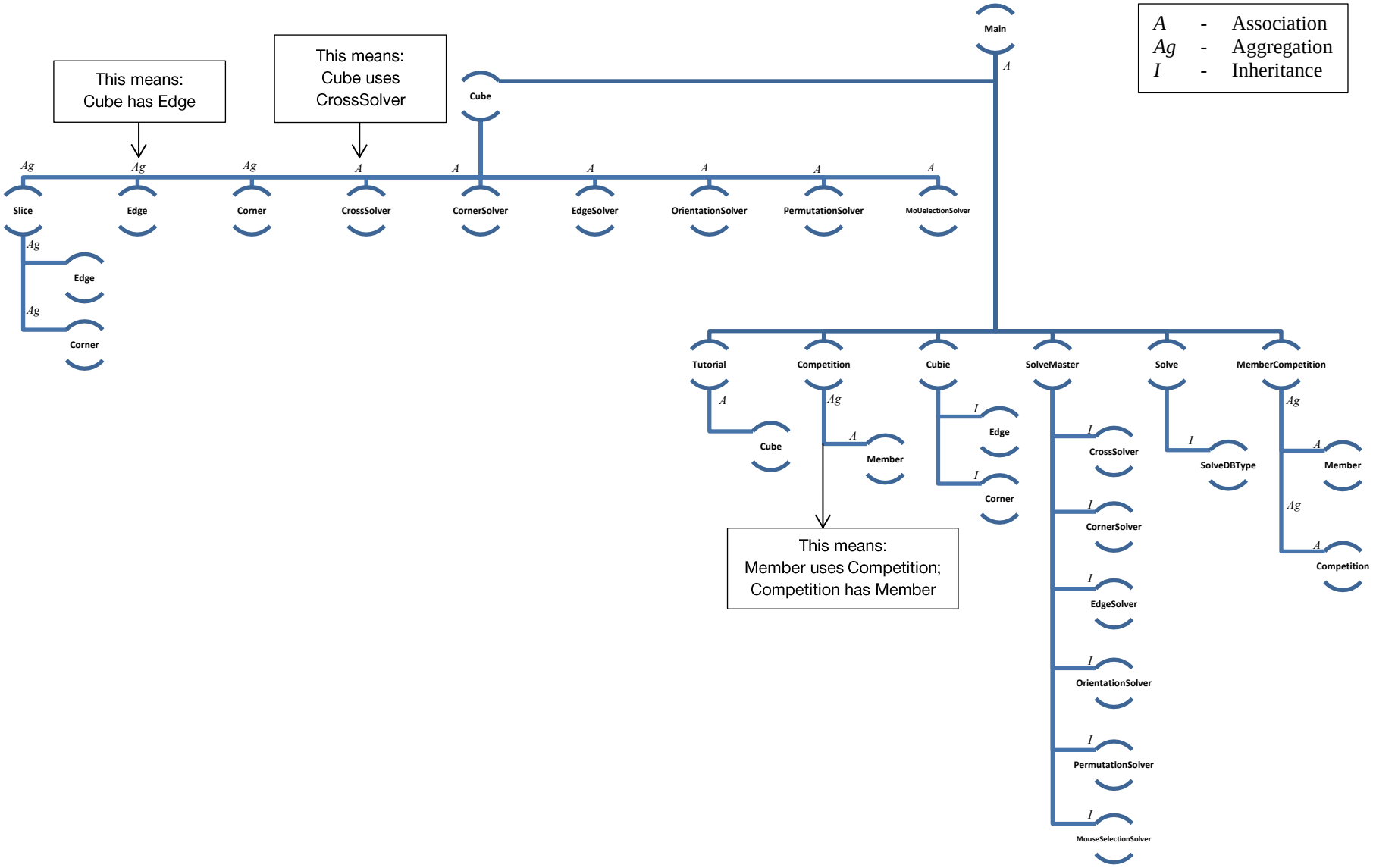
Slice

Fields	Methods
(Constructor) Slice(int[] cubieIndices)	
(Constructor) Slice(Edge[] edges, Corner[] corners)	
private edges private corners private centre private cubieIndices	public setEdges public setEdge public getEdges public getEdge public setCorners public setCorner public getCorners public getCorner public setCentre public getCentre public setCubieIndices public getCubieIndices public performMove public flipAllEdges public flipEdge public twistAllCorners public twistCorner public swapEdges public swapCorners public contains

Cube

Fields	Methods
(Constructor) Cube(Cube cube)	
private edges private corners private slices	public getEdges public getEdge public getCorners public getCorner public getSlice private resetCubieIndices public performAbsoluteMoves public rotate public updateCubies public updateAll public getSlices public resetCube

Class Relationships Diagram



Hardware Specification

Identification of Storage Media

- All permanent data, such as member details, competition details, and recorded solves, will be stored in the database on Dr McIvor's hard disk drive.
- The entire system (without stored data) will be approximately 8 MB is size, and this will be stored on the HDD.
- Backups can be made on any suitable storage medium, such as a USB flash drive.
- Since I am using SQLite, nothing needs to be installed in order to access the database.
- No other specific hardware is required – only standard peripherals (keyboard and mouse).

OS: Windows XP/Vista/7
Processor: 1.0 GHz
Memory: 128 MB RAM
Graphics: N/A
Hard Drive: 8 MB available space

Security and Integrity of Data

- All data relating to members will be stored on Dr McIvor's computer, which will be offline, so there is no need to enforce additional passwords since his computer is already protected by a password.
- All other data (e.g. solves, algorithms etc.), which will be stored on the members' computers, is not sensitive, so the security of this data is not an issue.
- The safety of the recorded data will be enforced by the use of warning messages, e.g. if the user tries to delete more than 5 solve records at a time, they will be given a warning message asking whether or not they wish to continue.
- When a user tries to submit data, validation checks will be run on the data to maintain data integrity.
- Reminders will be given to the user when the program starts about the backing-up of data to increase its safety.

Security and Integrity of System

- The only person who needs to be concerned about the security of the system is Dr McIvor; his computer will be storing personal information, which must be kept secure. Since the data is local to the HDD, a password-protected account will provide the security required to make the system secure.

Design Data Dictionary

Solve Table

Field Name	Data Type	Example	Validation	Default Value
solveID (Primary Key)	Integer	1	N/A	Auto increment
solveTime	Text	1:04.58	Presence check. Format check (see page 73) so that the time can be processed correctly throughout in the rest of the program.	DNF
penalty	Text	2	N/A	0
comment	Text	“Amazing”	N/A	-
scramble	Text	R U R’ U’	N/A	-
solution	Text	U R U’ R’	N/A	-
dateAdded	Text	2014-01-01 12:12:12	Presence check. Format check: yyyy-MM-DD HH:mm:ss so that it can be sorted properly later.	-

Algorithm Table

Field Name	Data Type	Example	Validation	Default Value
algorithmID (Primary Key)	Integer	5	N/A	Auto increment
moveSequence	Text	R U R’ U R U2 R’	N/A	-
comment	Text	Sune	N/A	-

Member Table

Field Name	Data Type	Example	Validation	Default Value
memberID (Primary Key)	Integer	14	N/A	Auto increment
forenames	Text	John Joe	Presence check.	-
surname	Text	Johnston	Presence check.	-
gender	Male/Female	Male	N/A because gender will be chosen from a drop-down list.	-
dateOfBirth	Text	01-01-1998	Presence check. Format check: DD-MM-yyyy	-
email	Text	john@gmail.com	Regex Match defined by the RFC822 standard (see page 72)	-
formClass	Text	11M	Regex Match: "0*(|8|9|10|11|12|13|14)[MRSTW]" so that no non-existent form class can be entered.	-

Competition Table

Field Name	Data Type	Example	Validation	Default Value
competitionID (Primary Key)	Integer	7	N/A	Auto increment
date	Text	10-10-2014	Presence check. Format check: DD-MM-yyyy	-

MemberCompetition Table

Field Name	Data Type	Example	Validation	Default Value
competitionID (Primary Key, Foreign Key)	Integer	9	N/A	-
memberID (Primary Key, Foreign Key)	Integer	3	N/A	-
time1	Text	12.30	Format check (see page 73) so that the average of 5 can be calculated correctly.	-
time2	Text	1:00.45	Format check (see page 73)	-
time3	Text	27.62	Format check (see page 73)	-
time4	Text	09.66	Format check (see page 73)	-
time5	Text	12.70	Format check (see page 73)	-

Preferences (Stored in a text file)

Field Name	Data Type	Example	Validation	Default Value
realTimeSolvingSpeed	Integer	130	Must be numeric because the speed must be a whole number. The number will be truncated to an integer.	250
inspectionTime	Integer	15	Must be numeric. The number will be truncated to an integer.	15
clickToSolveWarning	Boolean	No	N/A because this is chosen from radio buttons, which have a default option.	Yes
scrambleTextSize	Integer	32	Must be numeric. The number will be truncated to an integer.	27

RFC822 Email Regular Expression

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[
\t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[
\t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[
\t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[
\t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[
\t])*)(?:,\s*(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[
\t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[
\t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[
\t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*:(?:(?:\r\n)?[
\t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*))*)?;\s*)
```

Solve-Time Regular Expression

```
(\d{1,2}:)?\d{1,2}\.\d*
```

This regular expression accepts the following patterns:

- MM:SS.ss
- MM:S.ss

- M:SS.ss
- M:S.ss

- SS.ss
- S.ss

The number of digits after the decimal point must be at least one, but there is no maximum length for the number of digits after the decimal point.

Data Structures

LinkedList<String> trackingMoves

When the user is solving the cube in timed conditions, their moves will be recorded. There are two ways I could record their moves:

1. Use a static array, and then change the size of the array when a move needs to be added. The pseudo-code would look something like this:

```
while (timerIsRunning) do
   if (keyTyped) then
      arrayCopy ← copyOf(trackingMoves)
      trackingMoves ← new array[trackingMoves.length + 1]
      for (i ← 0; i < arrayCopy.length; i ← i + 1)
         trackingMoves[i] ← arrayCopy[i]
      end for

      trackingMoves[trackingMoves.length - 1] ← getKeyToMove(keyTyped)
   end if
end
```

This is very inefficient since the entire array needs to be refreshed in order to add a single element.

2. Use a dynamic data structure (such as a linked list) in order to add elements:

```
if (keyTyped) then
   trackingMoves.add(getKeyTypedToMove())
end if
```

This is much more efficient, and is more appropriate for the situation since a completely unknown number of elements will be in the list. Some users may take 20-30 moves, and others may take more than 500 moves to complete a solve, so a dynamic data structure is most appropriate here.

Tutorial data

When a tutorial is loaded, the data will be stored in arrays of String. Some tutorial data has multiple pieces of information, such as multiple hints, expected solved pieces, and optimal solutions. These will be stored in two-dimensional arrays. Each element in the first dimension will be used to store the information for n^{th} sub-tutorial, and the second dimension will be used to store the individual pieces of information for that sub-tutorial.

SolveCandidates[] solveCandidates

The CornerSolver class and the EdgeSolver class both implement an algorithm which solves their respective steps using this basic structure:

```
1. Find the number of moves required to solve each cubie of the step.
2. Solve the cubie that requires the fewest number of moves to solve.
```

The SolveCandidates class doesn't have any methods since it is a nested class and only requires the functionality of a 'record' data type.
The SolveCandidates class has this structure:

```
class SolveCandidate {
      int index = -1000;
      int score = -1000;
}
```

The 'index' field stores the index of the cubie on the cube. The 'score' field stores the number of moves required to solve the cubie.
I chose a static array structure since there will be a maximum of four elements in the array, and it doesn't matter if there are fewer elements than this. Another variable, 'solveCandidatesIndex' stores the number of cubies examined, and this variable is used in the boolean expression in the for loop which determines the cubie to solve, so errors won't occur.

Solves

After the user completes a time, it will be stored using the methods and constructors in the 'Solve' class. The Solve class contains the following fields:

```
private String time
private String comment
private String penalty
private String scramble
private String solution
```

I am storing 'time' as a string because it makes storing the data easier and means that, overall, there are fewer conversions between the displayed time (xx:xx.xx) and the numerical value, which should make the system faster.

File Organisation

Saving Cube State

The user can save the current state of the cube in a text file with this structure:
(Top Centre), (Front Centre), $(\sum_{i=0}^{7}(C_i[0], C_i[1], C_i[2]))$, $(\sum_{i=0}^{11}(E_i[0], E_i[1]))$

Example
orange,green,yellow,green,red,orange,blue,white,orange,white,green,red,blue,yellow,red,white,blue,
green,yellow,orange,yellow,blue,orange,red,green,white,orange,blue,orange,white,orange,green,oran
ge,yellow,red,yellow,blue,white,green,white,blue,yellow,red,blue,green,yellow,red,green,red,white

The user can load a cube state from file stored in this structure. The cube will be rotated so that the first colour in the text file is at the top, and the second colour is on the front. The other colours are then assigned to each cubie.

Save Statistics

Statistics can be shown in the bottom panel of the main window, and can be exported as PDF. The data are stored in a table in this format:

Average Type	Average	Times
Best Average of 5	14.24	(13.09), (17.64), 13.99, 13.30, 15.42
Best Average of 12	15.19	13.09, 16.90, (19.13), 13.09, 17.64, 13.99, 13.30, 15.42, 14.31, 15.05, 19.13, (12.66)
Best Average of 50	…	…
Best Average of 100	…	…
Current Average of 5	17.42	16.96, 21.09, (13.25), 14.20, (23.37)
Current Average of 12	18.44	13.46, 16.96, 17.49, 21.62, 17.22, 22.04, (25.22), 16.96, 21.09, (13.25), 14.20, 23.37
Current Average of 50	…	…
Current Average of 100	…	…

Saving Individual Solve Information

If the user wishes to store the information about a particular solve, they can save it in a text file since. Text files are a very appropriate format, because these can be shared over the internet as plain text, which is very accessible. Here is the general format:

```
Line 1: time
Line 2: penalty
Line 3: comment
Line 4: scramble
Line 5: solution
```

Example

```
18.86
0
Fantastic
R U R' U'
U R U' R'
```

If the user wishes to load 'solve' information into the program, they can select the appropriate menu item and then select the file containing the information and the solve will be added to the list in the main window.

Tutorials

The information for tutorials will be stored in text files. A tutorial consists of sub-tutorials, and the format of these tutorials is:

```
Comment
"SCRAMBLE:"
Scramble to be applied. If no scramble is to be applied, the
character "N" will be used
"DESCRIPTION:"
Description to be shown to user
"EXPECTED FINAL STATE:"
The steps that are expected to be solved
"HINT:"
Any number of hints here
"OPTIMAL SOLUTION:"
Optimal/expected solutions to the problem.
"EXPLANATION:"
Explanation of the solution to the problem
"*"
```

If the sub-tutorial requires the user to perform moves, then the word "ENABLE" (in uppercase) should be included somewhere in the comment line, e.g. "Beginner Tutorial – ENABLE".

If the sub-tutorial requires to the user to solve a problem, then they must include an 'expected final state' section, then the subsequent sections (hint, optimal solution, explanation) must also be included. The tutorial file can either have (Comment, Scramble, Description, Expected Final State, Hint, Optimal Solution, Explanation) or (Comment, Scramble, Description), but no other forms.

In the ‘expected final state’ section, the author can specify which pieces/sub-steps they wish to be solved. For example:

```
EXPECTED FINAL STATE
w-g
w-o
w-b
```

means that the author wants the white-green, white-orange, and white-blue edges to be solved before the user can receive a completion message.

The hints section should include one hint per line. For example:

```
HINT
Try moving the white-orange edge out of the way first.
Perform F’ R then try to solve it from there.
```

The user can click the ‘Hint’ button in order to show the next hint.

Here is an example of a tutorial stored in a text file:

```
Example of R U R' case. ENABLE
SCRAMBLE:
L2 D L U F' U2 D F2 L D F' R2 B U2 R2 F' B' U2 L2 F2 D2 L' B R2 U L2 z2 U2
DESCRIPTION:
There are four situations you need to recognise when solving the corners. For now, we
will learn how to solve a corner if it is in the top layer and white is facing to the
right.

Look at the White-Green-Orange corner.

Its 'destination' is between the White, Green, and Orange centres (since those are its
three stickers), so we must bring the corner 'above' its destination. In this case,
the corner is already above its destination. To solve a corner, we must hold the cube
so that the corner is at the Top-Front-Right position, which is where the corner is in
this case. The white sticker is facing to the right, so now we can learn an algorithm
to solve this piece:
   R U R'
If you perform these moves on the cube, the White-Green-Orange corner will be solved.

Try solving the corner using the moves provided.
EXPECTED FINAL STATE:
cross
w-g-o
HINT:
Perform R U R' - ('i j k' on your keyboard)
OPTIMAL SOLUTION:
R U R'
EXPLANATION:
We rotate the cube so that the corner is at the Top-Front-Right position, then we see
that the white sticker is on the right, so we perform the algorithm (R U R').
*
```

Database Storage

Algorithm Table
Each record will have an average size of 230 Bytes. Any number of algorithms could be entered, but it is unlikely that a user would store more than 200 algorithms in the program, so one can assume a maximum size of 50 KB.

Solve Table
Each record will have an average size of 560 Bytes. The number of records stored in this table could be very large, so care must be taken with the maintenance of the data in this table, but with 1000 records, this would be only 560 KB.

Competition Table
Each record will have an average size of 9 Bytes. Over ten years, assuming one competition per two weeks, there would only be ≈ 200 records, so approximately 1.8 KB.

Member-Competition Table
Each record will have an average size of 88 Bytes. If all competitors compete in every competition over one year, then this will have a maximum size of 90 KB.

Member Table
Each record will have an average size of 120 Bytes. Assuming a maximum club size of 40 members, the entire table will have a maximum size of 50 KB.

Preferences
The memory required to store the preferences is approximately 18 Bytes.

Overall, the system will require approximately 1 MB to store all data.

Normalisation

1NF

Member(MemberID, Forenames, Surname, Gender, DateOfBirth, Email, FormClass, CompetitionID, Date, Time1, Time2, Time3, Time4, Time5)

Algorithm(AlgorithmID, MoveSequence, Comment)

Solve(SolveID, SolveTime, Penalty, Comment, Scramble, Solution, DateAdded)

2NF

Member(MemberID, Forenames, Surname, Gender, DateOfBirth, Email, FormClass)
Competition(CompetitionID, Date)
MemberCompetition(MemberID, CompetitionID, Time1, Time2, Time3, Time4, Time5)

Algorithm(AlgorithmID, MoveSequence, Comment)

Solve(SolveID, SolveTime, Penalty, Comment, Scramble, Solution, DateAdded)

3NF

(*Already normalised*)

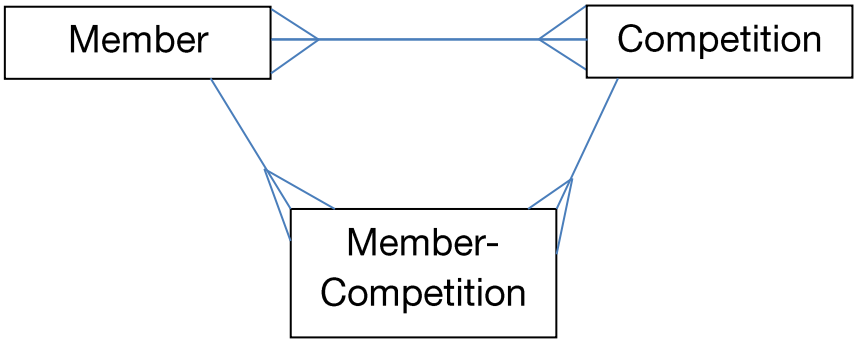
Member(MemberID, Forenames, Surname, Gender, DateOfBirth, Email, FormClass)
Competition(CompetitionID, Date)
MemberCompetition(MemberID, CompetitionID, Time1, Time2, Time3, Time4, Time5)

Algorithm(AlgorithmID, MoveSequence, Comment)

Solve(SolveID, SolveTime, Penalty, Comment, Scramble, Solution, DateAdded)

Normalised E-R Diagram

(Repeated from analysis since no change)

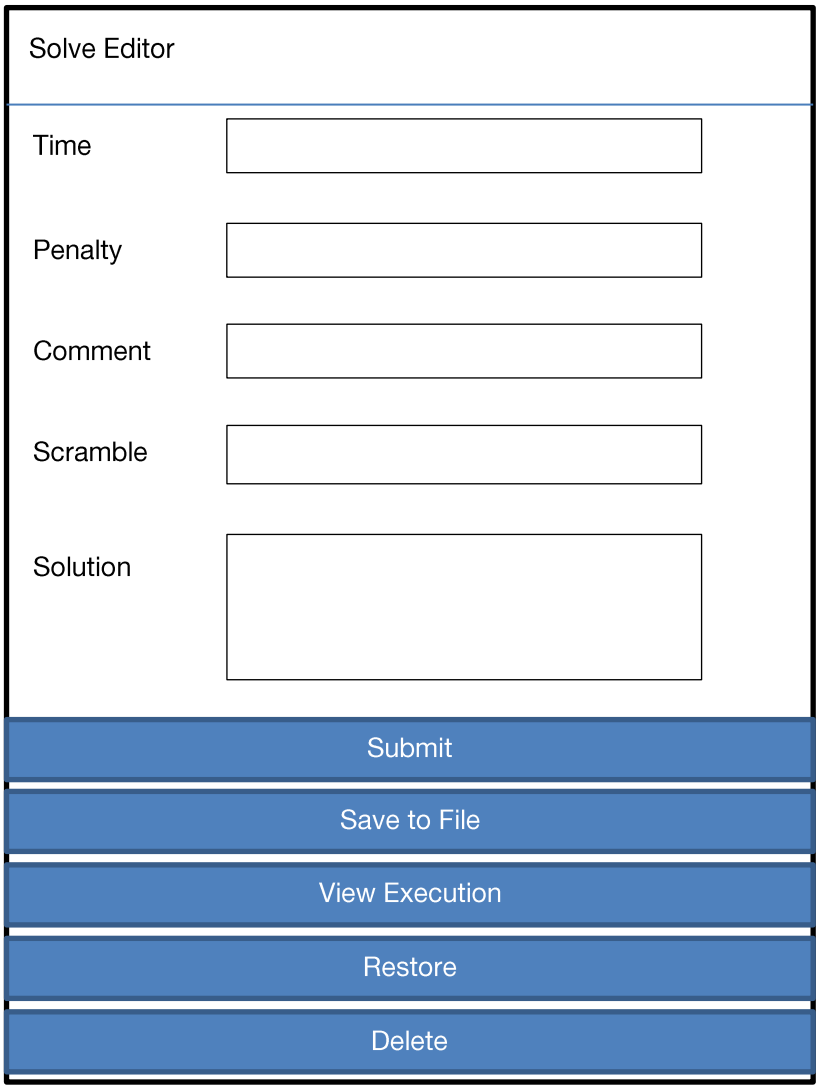


- A `member` can compete in many competitions
- A `competition` can have many competitors
- A `member` can have many `member-competition`s (which consist of their details/performance for that competition)
- A `competition` can have many `member-competitions` (which, in this context, are essentially registrations for that competition)

Human-Computer Interaction

Enter ‘Solve’ Information Form

- A user can modify an existing ‘solve’, or add a new time, by modifying/entering the information into this form.
- When the user clicks submit, validation checks will be run.
- If the validation checks pass, the form will close.



- **Time field**: a format check will be run using a regular expression to ensure the time is of the correct form: `"(\d{1,2}:)?\d{1,2}\.\d*"`

- Validation checks do not need to be run on the other fields since they are text based. The penalty field doesn’t necessarily have to be an integer since no calculations are performed with this field.

- If the user is editing the information about a solve, the fields will be populated with the existing data when the form opens.

- The comment, scramble, and solution fields will automatically show a scroll bar when the text overflows.

- **Save to File button**: when this button is pressed, a save dialog will open prompting the user to select a location to save the information. The data will be stored in a text file.

- **View Execution button**: when this button is pressed, the virtual cube will apply the scramble, then it will show the moves being applied at a rate specified by the user (e.g. 1 move per second).

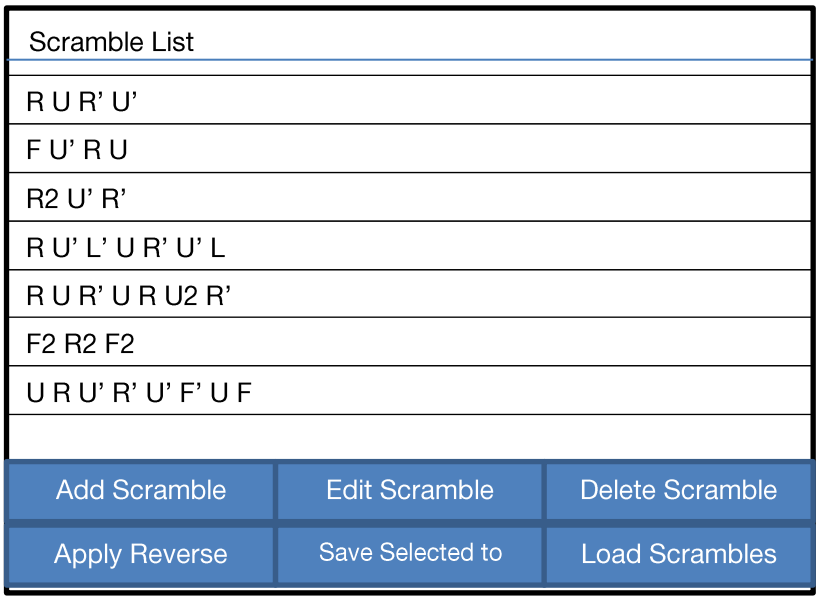
- **Restore Values button**: the pressing of this button will cause any unsaved changes to be reverted back to the original information.

- **Delete button**: the user can delete the information about the current solve by pressing this button, and the solve will be removed from the list displayed in the window.

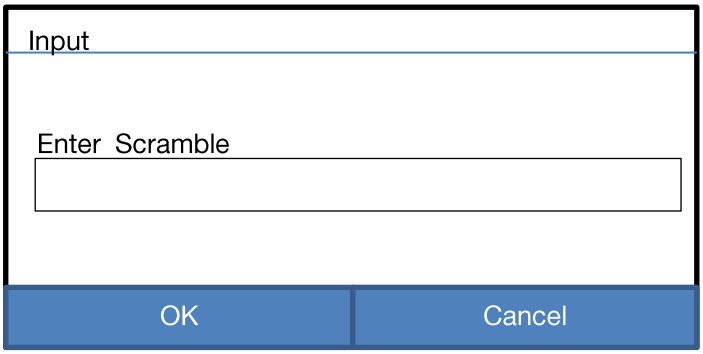
Scramble List Window

- A user can open the scramble list in order to save a scramble for later use. These scrambles will not be permanently stored since this list is intended to be temporary.
- The main purpose of this list is to provide an easy means of applying/reversing scrambles. This would be useful if, for example, if the user is learning an algorithm; instead of having to reverse the algorithm, then apply to the normal algorithm, they could click the 'Apply Reverse' button, which will then apply the selected scramble to the cube, and then perform the algorithm as normal.
- This scramble list may also be used to provide the 5 scrambles for a competition. If the appropriate menu option is selected, then the scrambles in this list will be used to scramble the cube.

(i)



(ii)



- **Add Scramble button**: the form shown in *(ii)* will be shown in order to capture the data required. A scrollbar will be shown when the list begins to flow over the viewable area.

- **Edit Scramble button**: the form shown in *(ii)* will be shown with the existing scramble shown in the text field, and they will be prompted with this message: "Enter New Scramble".

- **Delete Scramble button**: the selected scrambled will be removed from the list.

- **Apply Reverse Button**: The reverse of the selected scramble will be applied to the cube in the simulator.

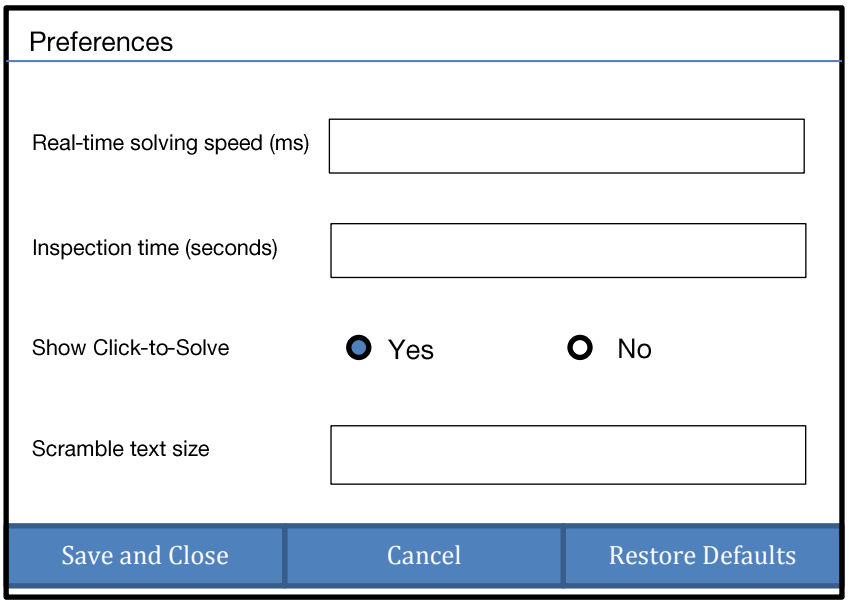
- **Save Selected to File button**: The user will be prompted to select a location to which the selected scrambles will be saved in a text file – one scramble per line.

- **Load Scrambles button**: the user will prompted to select a text file on their computer containing the scrambles to be loaded. The system will then add each line of the text file to the list of scrambles.

- **Enter Scramble field**: this is the text field into which the user enters the scramble. The only validation check that will be run on this field will be a presence check.

Preferences Form

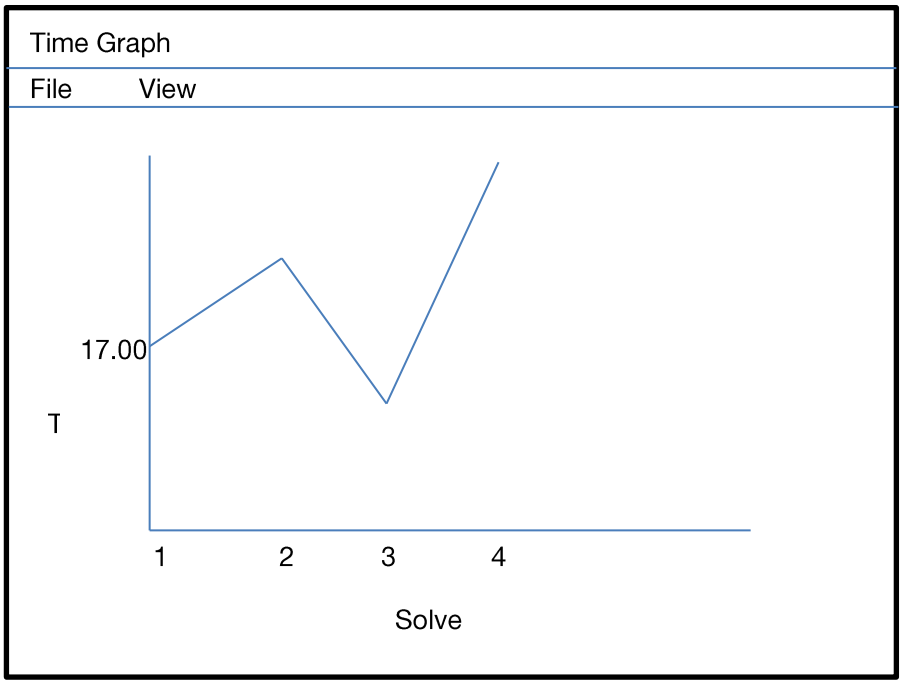
- The user can save preferences by entering information into this form.
- The form can be shown by clicking the preferences menu item in the main window.
- The preferences will be stored in a text file, and each time the user opens the program, the fields will be populated with the data stored in the text file. If the preferences file does not exist, then the fields will be populated with default values.



- **Save and Close button**: when clicked, the program will attempt to store the data entered in the preferences file.
- **Cancel button**: the form will close, and any unsaved changes will be deleted.
- **Restore Defaults button**: when clicked, the default preferences will be restored, the fields will be updated, and the default values will be written to the preferences file.
- **Real-time Solving Speed field**: a type check will be run to ensure that the value entered is a number, and a range check to ensure the number fits the criteria: $0 \leq x < 10000$, and a presence check.
- **Inspection Time field**: a type check will be run to ensure that the value entered is a number, and a range check to ensure the number fits the criteria: $0 < x \leq 100$, and a presence check.
- **'Show Click to Solve Warning' buttons**: if the 'yes' radio button is selected, then a warning will be shown when the user wishes to enter 'Click to Solve' mode, otherwise, no warning will be shown. The 'yes' button is selected by default.
- **Scramble Text Size field**: a type check will be run to ensure that the value entered is a number, and a range check to ensure the number fits the criteria: $0 < x \leq 100$.

Time Graph Window

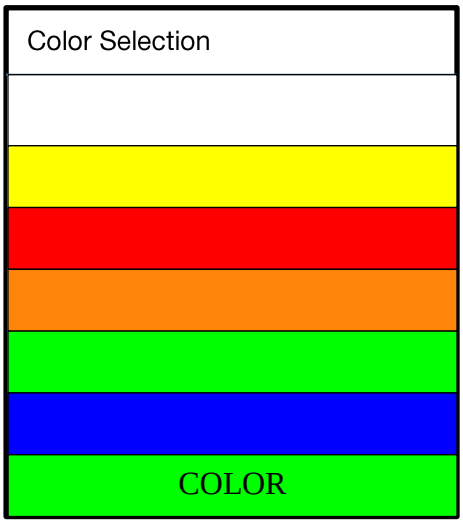
- This window will automatically update when the user adds a time to the time list. It will scale in order to show the previous 12 times (or all times if there are fewer than 12).
- 'DNF' times will not be added since these essentially represent $time = \infty$.



- **File menu**: this will have two menu items:
  - **Save As Image item**: the current display of the time graph can be saved as a .png image. When this menu item is clicked, the user will be prompted to select a location where the image will be saved.
  - **Close Window item**: when clicked, the window will close.
- **View menu**: this will have four menu items:
  - **Window Always on Top Checkbox item**: this will be a checkbox menu item. If the box is checked, the time graph will always be on top (i.e. it will be above the other windows running), otherwise, it will allow other windows to be on top.
  - **Reset Zoom item**: the user can zoom in on a particular region of the graph, and can reset to the default zoom by clicking this menu item.
  - **Dimension Selector buttons**: there will be a radio button which, when selected, will show the graph in 2D, and another radio button which, when selected, will show the graph in 3D.

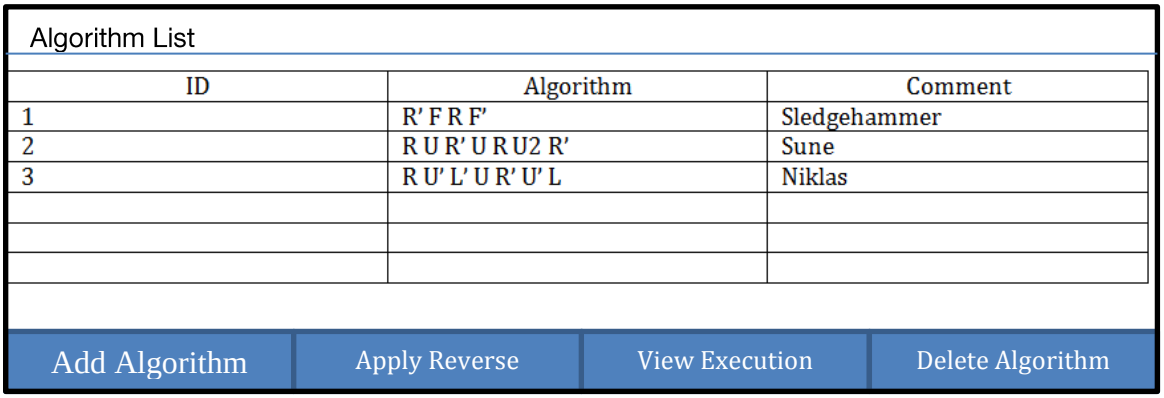
Color Selection Window

– If the user wishes to paint a custom state on the cube, then they will need a means of selecting the current color. The window shows several colored regions. If the user clicks the red region, for example, this will set the current painting color as red. The background color of the bottom label will always be the same as the current painting color. When the user clicks a sticker on the cube, the sticker's color will change to the selected color.



Algorithm List Window

- The user can view stored algorithms in this window.
- There will be some preset algorithms, which cannot be deleted.
- The ID column cannot be edited since this is the primary key in the algorithm table in the database.
- The user can edit any cell (other than those in the ID column) by double-clicking on the cell.



- **Add Algorithm button**: This adds a new row to the table.
- **Apply Reverse button**: This will apply the reverse of the algorithm to the cube in the program.
- **View Execution button**: This shows the algorithm being executed on the cube in the program.
- **Delete Algorithm button**: This deletes the selected rows from the table and from the database. If more than 5 rows are selected, the program will show a warning message and ask if the user wishes to continue.

Solve Table Window

- This window shows all of the solves stored in the 'solve' table in the database.
- The user will be able to sort the records according to specified criteria, and filter the data by certain criteria.

Solve Table

Sorting Filter

ID	Time	Penalty	Comment	Scramble	Solution	Date Added
1	0.15	0				2013-12-06 22:58:48
2	0.20	0		U R U' R'	R U R' U'	2013-11-15 11:14:01
3	0.26	0				2013-11-20 14:43:27

Add Solve | Edit Solve | Delete Solve | Load into Program

Solve Form

Time

Penalty

Comment

Scramble

Solution

Date Added

Submit

- **Add Solve button**: when clicked, the solve form will be shown, and the user can enter the information about the solve into this form. Validation checks will be run on the data as described on page 82 as well as a format check being run on the date to ensure that it is of the form yyyy-MM-DD HH:mm:ss. If the date is not in this form, the program will notify the user and will offer the option of using the current time. (The 'submit' button in the solve form will cause the data to be validated and, if all validation tests pass, then a row will be added to the table with the data entered into the form).

- **Edit Solve button**: when clicked, the solve form will be shown, and each of the fields will be populated with the data in the selected row.

- **Delete Solve button**: the selected rows will be removed from the table and from the database. If more than 5 rows are selected, a warning will be shown asking the user whether or not they wish to continue.

- **Load into Program button**: the selected solves in the table will be loaded into the main program. This is beneficial for the user since it allows the other features of the program to act on the information of that solve (e.g. save it as part of an average, view execution, view on time graph etc.).

- **Date Added field**: if the user hovers the mouse pointer over the field, a tool-tip-text will appear showing the text, "yyyy-mm-dd hh:mm.ss", informing the user that this is the form required.

- **Submit button**: when clicked, the data in the form will be run through validation checks, and if all checks pass, the form will be submitted, otherwise, the user will be prompted to correct the errors. If the date has been entered incorrectly, the program will ask the user if they wish to use the current time.

- **Sorting menu**: this will have six menu items:
    - **Sort by Time Ascending**: The rows in the table will be sorted with the fastest solve in the first row and the slowest solve in the bottom row.

    - **Sort by Time Descending**: The rows in the table will be sorted with the fastest solve in the first row and the slowest solve in the bottom row.

    - **Sort by Date Ascending**: The rows in the table will be sorted with the earliest date in the first row and the latest date in the bottom row.

    - **Sort by Date Descending**: The rows in the table will be sorted with the latest date in the first row and the earliest date in the bottom row.

    - **Sort by ID Ascending**: The rows in the table will be sorted with the least ID in the first row and the greatest ID in the bottom row.

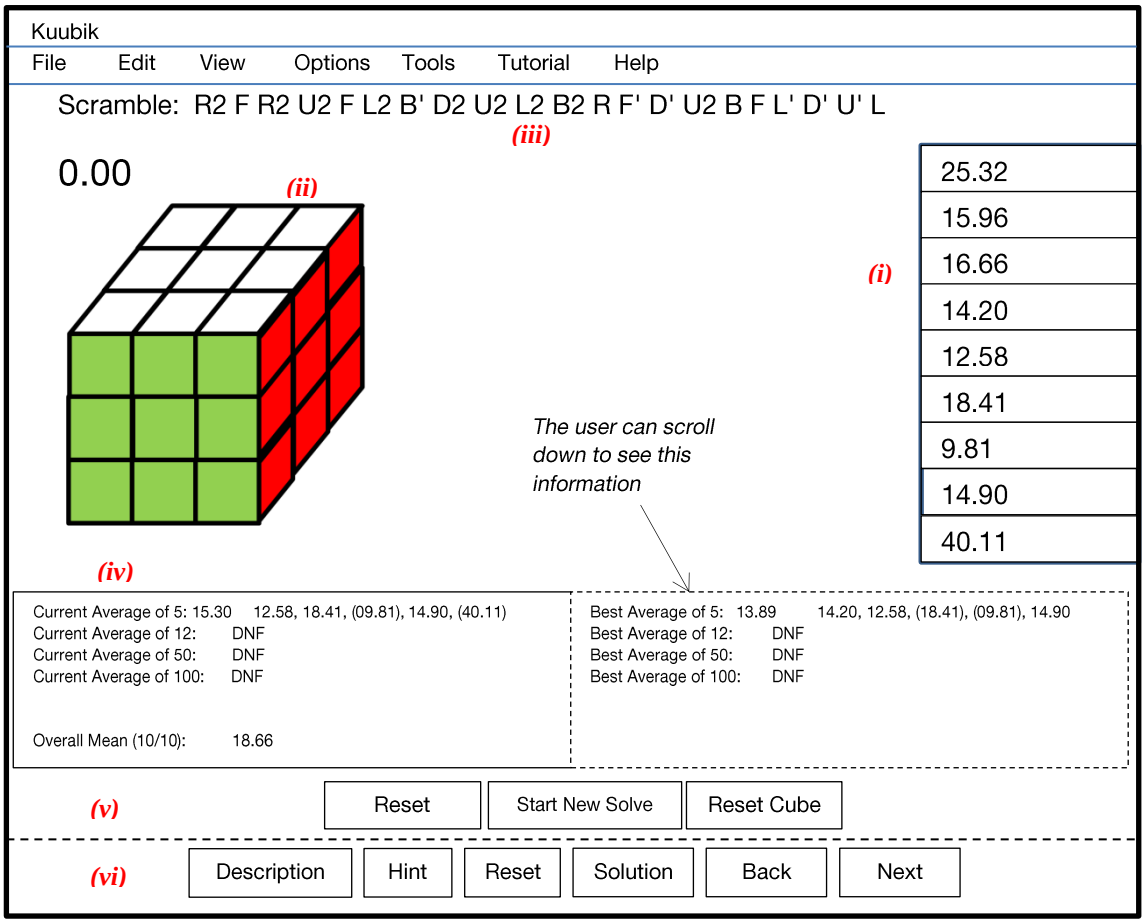
    - **Sort by ID Descending**: The rows in the table will be sorted with the greatest ID in the first row and the least ID in the bottom row

- **Filter menu**: this will have two menu items:
    - **Filter by Time**: The user can enter the lower and upper time boundaries for the data shown in the table. After entering the data, only data that fits the specified criteria will be displayed in the table.

    - **Remove Filter**: The filter being applied to the data will be removed so that all data will be shown in the table.

Main Window

- This is the window that will be loaded when the program starts.
- The button panel, *(v)*, will be shown when in the timing view, and the button panel, *(vi)*, will be shown when in tutorial mode.
- The user will be able to sort the records according to specified criteria, and filter the data by certain criteria.



- **File menu**: this will have two menu items:
  - **Save Cube State**: this will open a save dialog, asking the user to choose a destination. This will save the current state of the cube in a text file.
  - **Save Statistics**: this will open a save dialog, asking the user to choose a destination. The statistics shown in the bottom panel will be saved to a pdf file in a table.
  - **Save Session to Database**: when clicked, all of the solves listed in *(i)* will be saved to the solve table in the database.
  - **Load Cube State**: this will open a load dialog, asking the user to choose the file containing the cube state. Once the file has been chosen, the cube in *(ii)* will change state so that it corresponds with the file chosen. If the file is invalid, then the cube will be reset and the user will be informed.

- **Load Solve Information**: this will open a load dialog, asking the user to choose the file containing the solve information. Once the file has been chosen, the information in the file will be extracted and the solve will be added to *(i)*. If the file is invalid, the user will be informed.

- **Exit Program**: when clicked, the program will close.

- **Edit menu**: this will have three menu items:
  - **Add Solve**: when clicked, the form shown on page 82 will be shown, and the user can then enter the information about the solve. If the information is submitted, the solve will be added to *(i)*.

  - **Edit Selected Solve**: when clicked, the solve form will be shown and the user can edit the information about the current solve.

  - **Delete Selected Solve**: when clicked, the selected solve in *(i)* will be removed from the list.

- **View menu**: this will have six menu items:
  - **Show Time Graph**: when clicked, it will make the time graph (shown on page 87) become visible.

  - **Show Current Scramble**: this will be a checkbox menu item. When the box is checked, the scramble *(iii)* will be shown, otherwise it will not be shown.

  - **Show Scramble List**: when clicked, it will make the scramble list (shown on page 84) become visible.

  - **Show Algorithm List**: when clicked, it will make the algorithm list (shown on page 89) become visible.

  - **Show Solve List**: when clicked, it will make the solve list (shown on page 90) become visible.

  - **Show Statistics**: the bottom panel *(iv)* can switch between statistics-view, worked solution-view, and tutorial-view, so this menu item provides the user with a method of displaying the statistics if they are not currently being shown.

- **Options menu**: this will have two menu items:
  - **Preferences**: when clicked, it will make the preferences window (shown on page 86) become visible.

  - **Use Scrambles in List**: this will be a checkbox menu item. When the box is checked, the scrambles in the scramble list will be used to scramble the cube. It will use the first scramble, then the second scramble etc. until it has used the last scramble, then it will begin to use the first scrambles in the list again. When the box is not checked, a random scramble will be generated and this will be used to scramble the cube.

- **Tools menu**: this will have six menu items:
    - **Preferences**: when clicked, it will make the preferences window (shown on page 86) become visible.
    - **Cancel Solve**: this cancels any time-related activities including the current solve, real-time solution displays, inspection time etc., and stops the timer.
    - **Clear Stickers**: when the user is painting a custom state, they can select this menu item to clear all of the stickers on the cube, i.e. all stickers become grey. If the user is not painting a custom state, then this menu item will be disabled, i.e. the user won't be able to select it.
    - **Paint Custom State**: this will be a checkbox menu item. When the box is checked, the color selection window (shown on page 88) be visible, otherwise it will be hidden. If the user tries to exit the custom painting mode and the cube is not in a valid state, a warning will be shown asking the user whether they want to continue painting or reset the cube.
    - **Solve Cube**: when clicked, the program will generate a solution to the current state of the cube, display the solution in *(iv)*, and perform the moves on the cube at a rate specified in the preferences.
    - **Solve Piece**: this will be a checkbox menu item. If the box is checked and the user clicks on a cubie in *(ii)*, the program will show the solution for how to solve that cubie in *(iv)*. The user can specify (in the preferences) whether they wish to see a warning message when they enter this mode; the warning message will have this text, "Click on a piece to view solution".
    - **Apply Random Scramble**: when clicked, a random scramble will be generated and applied to the cube. This scramble will be shown in *(iii)*.
- **Tutorial menu**: this menu will contain five sub-menus and two menu items. Each sub-menu will contain a number of menu items. When a menu item in a sub-menu is clicked, the corresponding tutorial will be opened and the instructions will be displayed in the panel, *(iv)*, and the buttons will change to *(vi)*. The five sub-menus will have the following labels: "Cross", "Corners", "Edges", "Orientation", "Permutation". When the program enters tutorial mode, the label *(iii)* will change to 'Tutorial' . The other two menu items will be:
    - **Load Tutorial from File**: this will open a load dialog, asking the user to choose the file containing the tutorial information. Once the file has been chosen, the information the file be extracted and the tutorial mode will be activated so that the user can work through the tutorial.
    - **Exit Tutorial**: when clicked, the program will exit tutorial mode and the user can use the program as normal. The label *(iii)* will change back to 'Scramble:', and the buttons will change back to *(v)*.
- **Help menu**: this will contain only one menu item – 'Documentation'. When this item is clicked, help documentation will be shown.
- **Reset Times button**: when clicked, the list, *(i)*, will be cleared.

- **Start New Solve button**: when clicked, the cube will be scrambled using either a random scramble or a scramble from the scramble list (determined by the 'useScramblesInList' checkbox), then the inspection timer will start, and then the user can start the timer by pressing spacebar.

- **Reset Cube button**: when clicked, the cube will reset itself to a solved state and all timers will stop.

- **Description button**: when clicked, the panel, *(iv)*, will show the description of the current sub-tutorial.

- **Hint button**: when clicked, the panel, *(iv)*, will show the next hint for the current sub-tutorial.

- **Reset button**: when clicked, the cube will reset back to the initial state of the current sub-tutorial, and any recorded moves will be cleared.

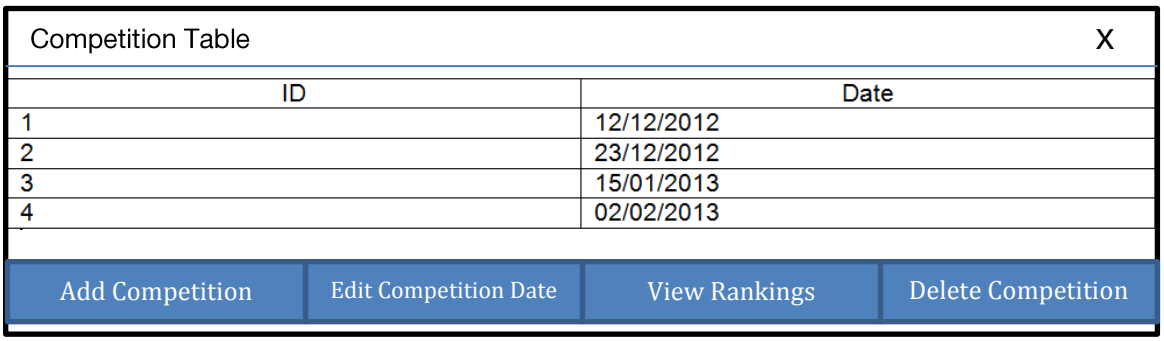
- **Solution button**: when clicked, the panel, *(iv)*, will show the solution for the current sub-tutorial.

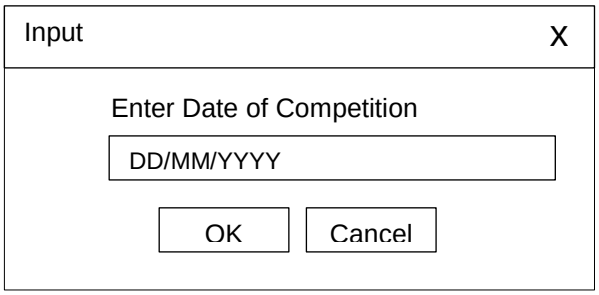
- **Back button**: when clicked, the previous sub-tutorial will be loaded and the panel, *(iv)*, will show the description of the sub-tutorial.

- **Next button**: when clicked, the next sub-tutorial will be loaded and the panel, *(iv)*, will show the description of the current sub-tutorial.

Competition Table Window

– This window shows all of the competition data stored in the competition table.

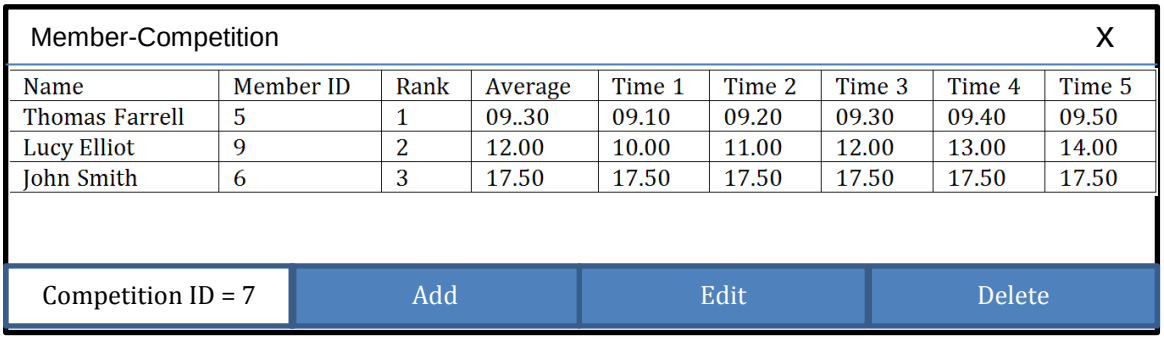


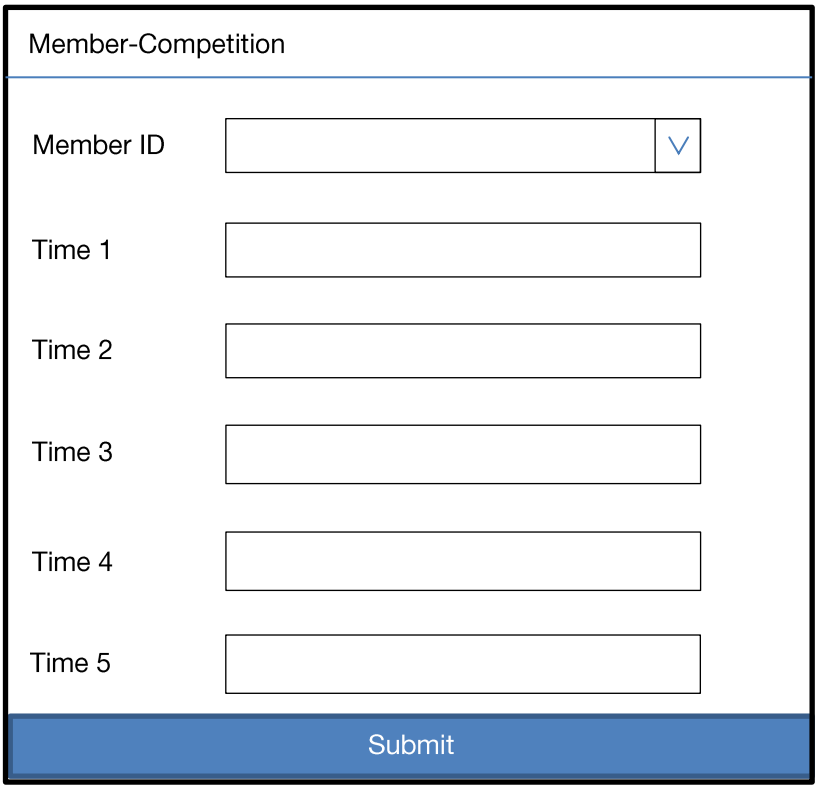


- Add Competition button: when clicked, a row will be added to the table with an automatically generated competitionID and a blank cell for the date.
- Edit Competition button: when clicked, the input window will be shown, allowing the user to enter a date for the competition. If they press 'Cancel' or close the window, the data will be unchanged.
- View Rankings button: when clicked, the rankings for the selected competition will be presented in the table shown on page 98.
- Delete Competition button: the selected rows will be removed from the table and from the database. The user will be prompted with a warning asking if they wish to proceed.

Member-Competition Table Window

- This window shows the rankings for a selected competition.
- The user can enter, edit, or delete information about the selected competition.
- If a user is deleted from the database, their times will still be stored for the corresponding competitions.
- The rows will always be sorted by average, then time 1, then time 2 etc.





- **Add button**: when clicked, the member-competition form will be shown. The Member ID combo box will hold items representing the available member ID's. For example, if there are four members in the database with IDs 1, 2, 3, 4 respectively, then the combo box will have four items – 1, 2, 3, 4. After a row is added to the table, the corresponding ID will be removed from the combo box. For example, if a row is added with Member ID = 3, then the next time the user clicks the add button, the combo box will have three items – 1, 2, 4.

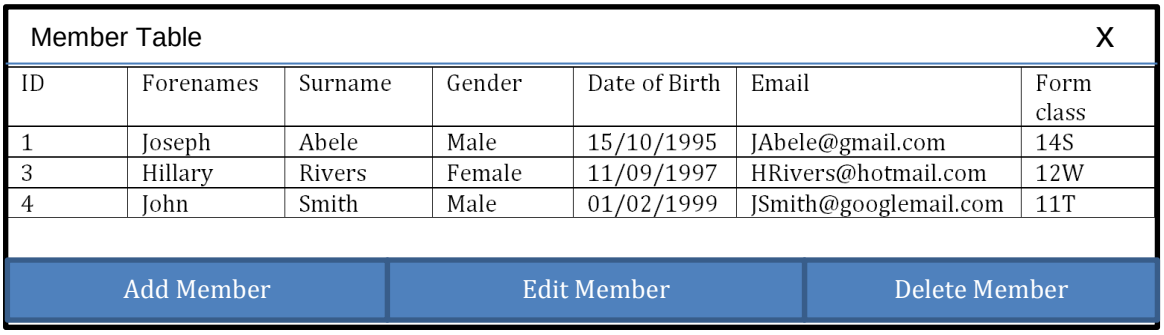
- **Edit button**: when clicked, the member-competition form will be shown and the fields will be populated with the values in the selected row in the table. For example, if the third row of the table shown is selected, when the edit button is pressed the Member ID combo box will have the selected value as ‘6’, and the other values will be 1, 2, 3, 4, 5; time1Field will have the text ’17.30’; timeField2 – ’17.40’; time3Field – ’17.50’; time4Field – ’17.60’; time5Field – ’17.70’.

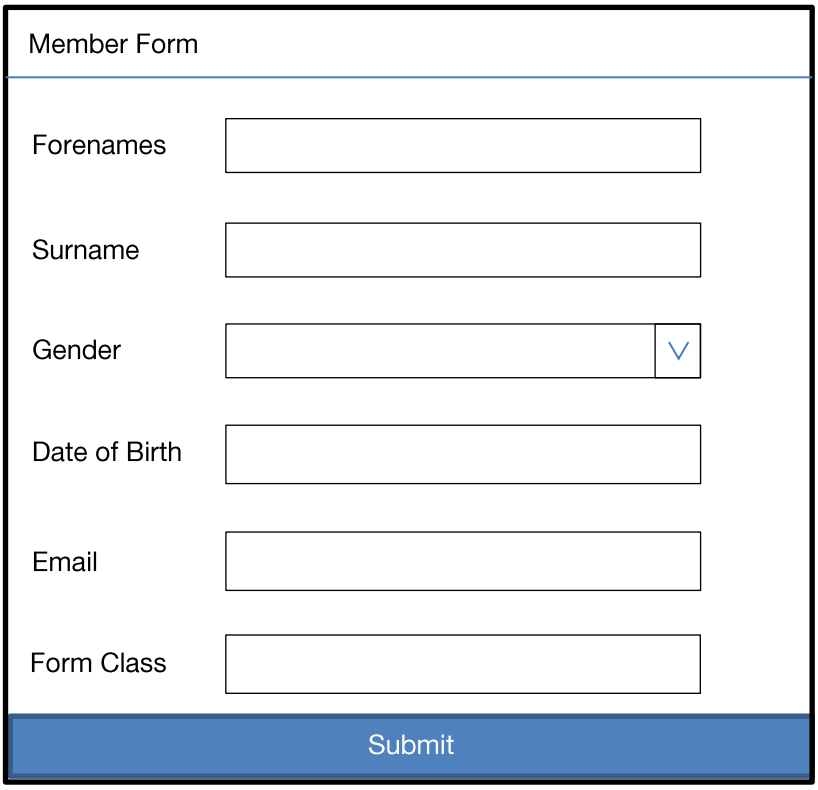
- **Delete button**: the selected rows will be removed from the table and from the database. The user will be prompted with a warning asking if they wish to proceed.

- **Submit button**: when clicked the data entered in the form will be passed through validation checks; each time field will be checked to see if the time entered is valid (and not empty).

Member Table Window

- This window shows the rankings for a selected competition.
- The user can enter, edit, or delete information about the selected competition.
- If a user is deleted from the database, their times will still be stored for the corresponding competitions.
- The rows will always be sorted by average, then time 1, then time 2 etc.





- **Submit button**: when clicked, the data entered in the form will be passed through validation checks. If an error if found, the user will be prompted.
- **Gender combo box**: the combo box will contain two items – 'Male' and 'Female'.

Overall Test Strategy

I will test the system throughout (integration testing) for validity and bugs so that as the number of modules working together accumulates, the system should still be stable.

My overall test strategy consists of the following:

Test Series	Purpose of Test Series
1	Test control flow (top-down testing). Test if the menu items and buttons go to the correct part of the system.
2	Test data validation (bottom-up testing). Test if the data input by the user is accepted or rejected appropriately.
3	Test sorting, calculations etc. (white-box testing). Test if selections and iterations carried out correctly with accurate results for calculations.
4	Test saving of data (system testing). Test if the specified data is updated in the corresponding tables, files, or in-program lists.
5	Test whether or not results of background processes are displayed correctly to user (black-box testing).

I will test normal, boundary, and erroneous data for data input.

After testing the system myself, I will perform acceptance testing with my end-user. I will ask him to provide input for the system to ensure that it operates in the manner he expects.