Contents

Analysis

Design

Technical Solution

Testing

System Maintenance Guide

Evaluation

Appendix

User Guide

*This is located at the very back of the document (after the Appendix) and uses its own table of contents and page numbers (which are in **red** rather than **black** and start with the letters **UG**).*

Analysis

1 Introduction

1.1 Background to the problem

Writing music is an art form that many young people are striving to take up. The problem is that very few people are naturally talented at doing it, yet it is a highly competitive field where you need to work as quickly and as productively as possible to gain your music products a good market share.

Usually, songwriting takes years of study, practice, networking and failures to make a successful product. The aim of my problem is to speed that process up by giving musicians ideas of melodies to use. It will not be designed to take away the art of songwriting; it will simply guide. I would argue that no computer program can fully replicate the art abilities of humans.

Musicians who are struggling to write their own music do so as they may be unmotivated by the complexities of songwriting and music theory. My program aims allow musicians who have knowledge of their musical style but limited experience of theory to write convincing melodies and harmonies to incorporate into their compositions quickly and effectively.

1.2 Problem Definition

The program allows the user to create songs, which are composed of clips of notes.

The user will be able to log into an account, the details or which are stored in a database, so that they can save clips and whole songs to it. The user's details are stored in one table in the database, another stores saved clips and another stores saved songs.

The program will allow the user to insert clips into a sequence of length determined by the user; they can choose how long they would like their clip to be, be it a bar, 2 bars, half a bar, quarter of a bar, etc. For each project, they can choose the key, time signature, and whether the song is major or minor. For each clip, the user can choose what range of octaves they would like their clip to have, what range of notes to have (from 1, being just one note, to 8, being the largest range across an octave), what syncopation they would like, how much syncopation they would like, and whether the clip is at the start, middle or end of a phrase (different note intervals are used at the start of a song compared to the end to signify changes). They can also save clips to use later, and insert saved clips. The clips will be displayed and edited using a sequencer in the centre of the user interface. There will be menu strips showing key commands for other features, such as saving the sequence, user account settings and inserting clips.

For the purposes of simplicity, my program will only generate notes in the treble clef. It is then up to the user to transpose as they wish.

2 Investigation into the Problem

2.1 Introduction

This section will be discussing two other software packages available with similar functions, how they are useful, and how my new program could help to improve upon where the packages meet their limits. I will also be demonstrating the data flow and features of the existing software compared to my proposed software.

2.2 Description of the Current System

Various music composition packages exist, though many purely exist for the creation of music notation.

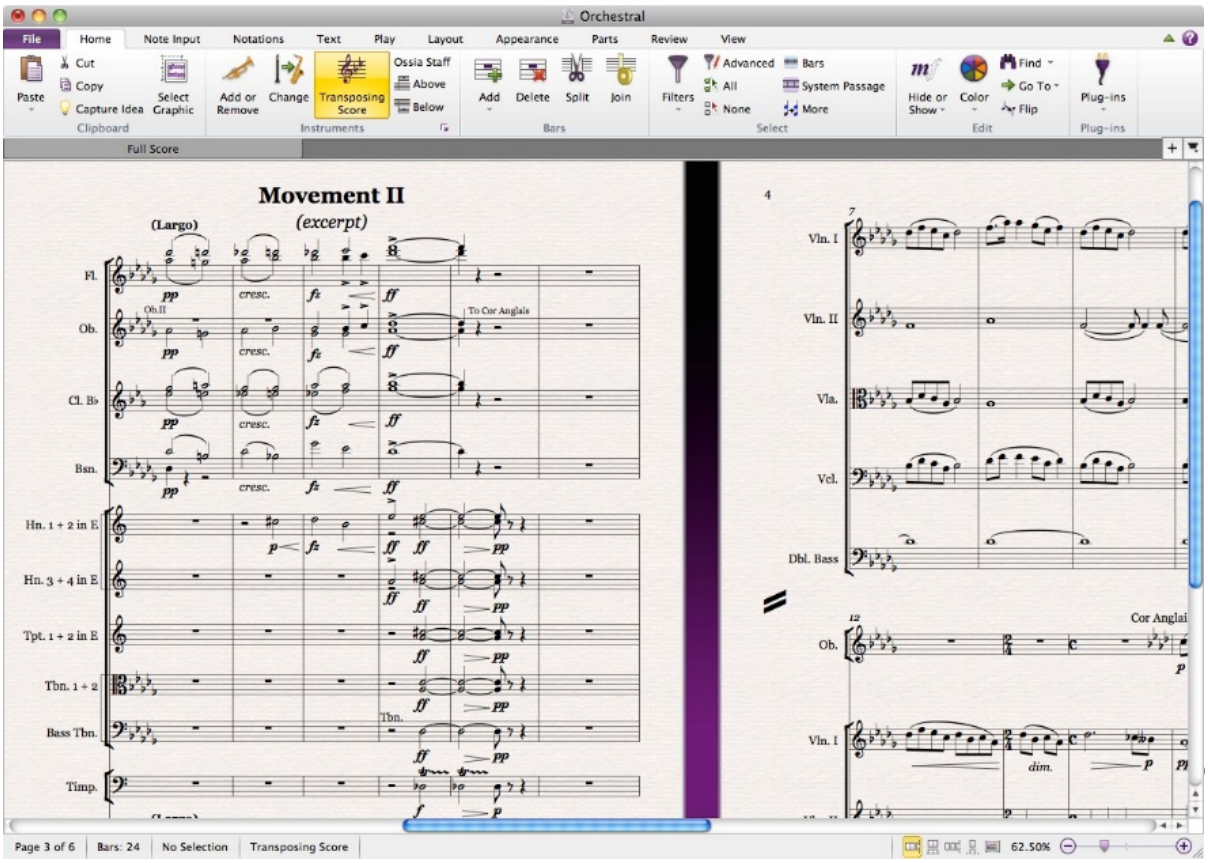
2.2.1 Sibelius

Developed by Avid Technology, Sibelius is one of the leading composition packages in the world. It allows users to write notes onto a virtual sheet of music. It's designed for users who have little experience with musical instruments and applies correct notation, with correct note and rest values.

You can have multiple staves and clefs for different instruments in a score and add lyrics, and simulate the score being played by an ensemble using a sample library; you can mix the level of the different instruments.

In **Figure 1.1**, note the ribbon at the top of the program where all of the tools are contained for editing the parts. In this example, you have the basic tools, such as adding or removing parts for the different instruments and changing a part to a different instrument, which is necessary as each instrument needs a particular type of clef (whether it be treble, bass, alto, percussion, etc.). The Bars section in the ribbon can add or delete new bars for all of the parts, and you can split one bar of a part into two, or join multiple bars into one.

Figure 1.1 *– a typical score arrangement within Sibelius.*



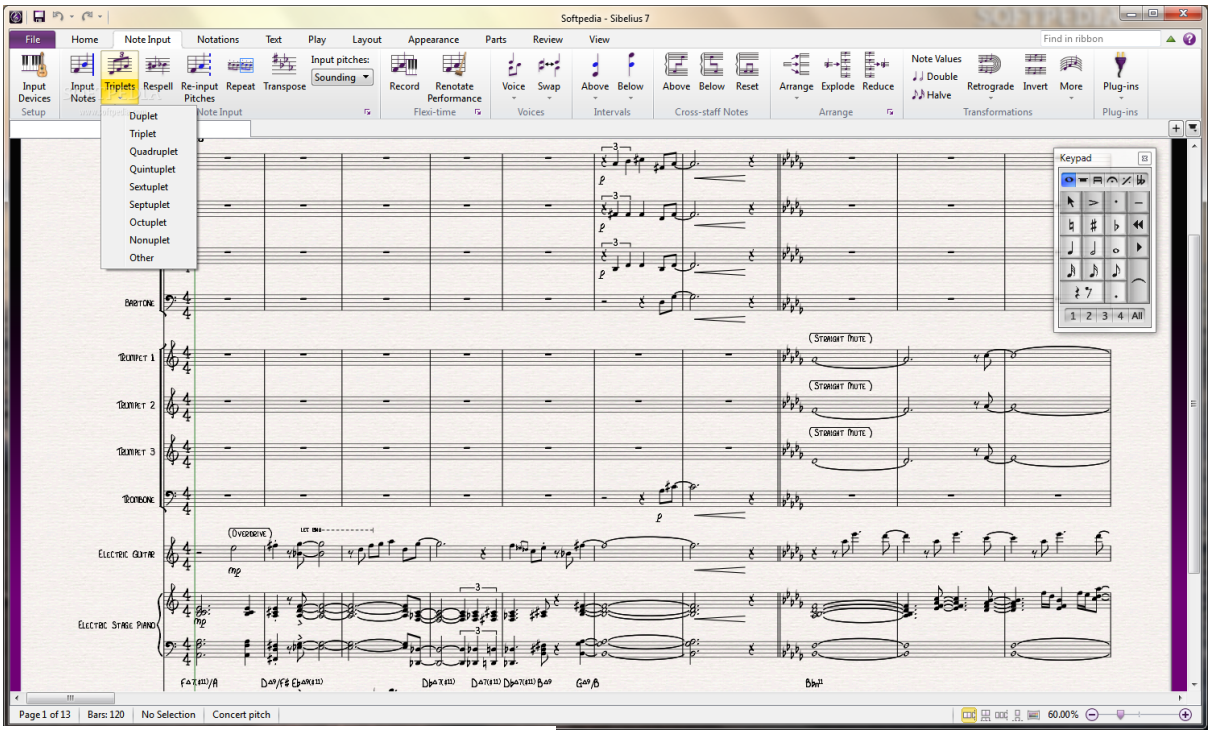
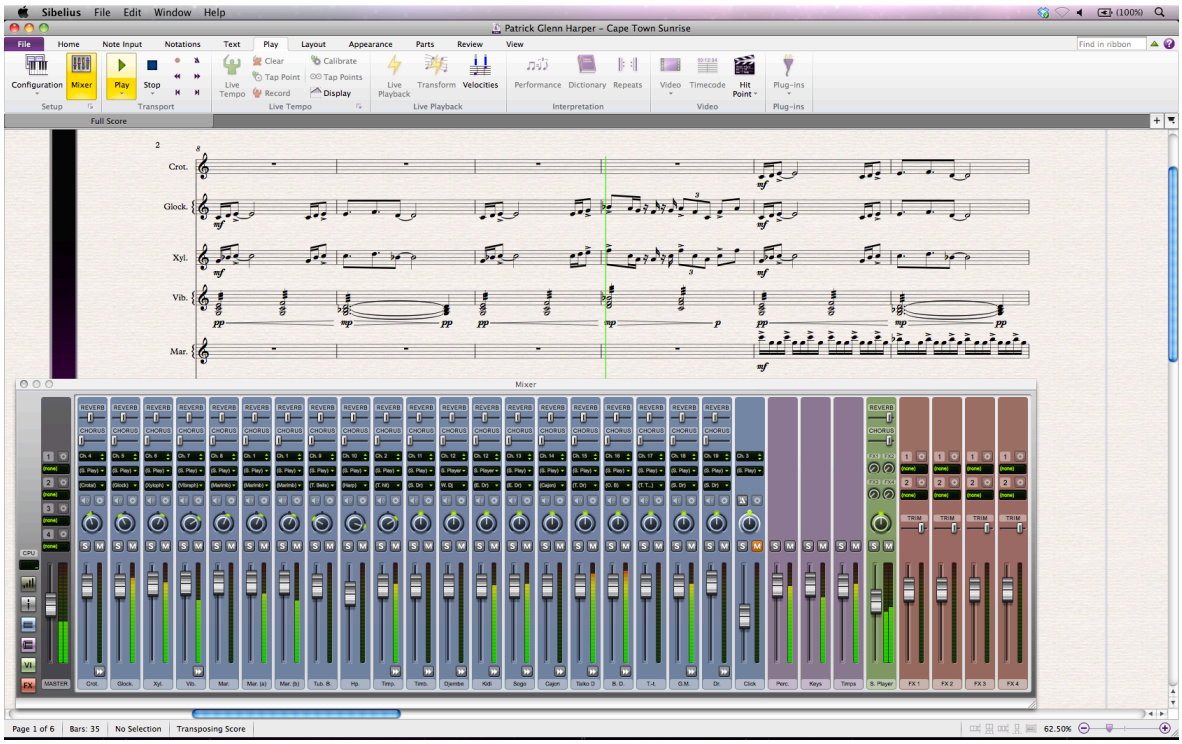


Figure 1.2 *– a look at the note insertion in Sibelius.*

Figure 1.2 shows the features of the Note Input ribbon in Sibelius. The 'Input Devices' button lets you set up an external device that you can use to input notes, i.e. a MIDI keyboard. You can also input notes by using the computer's keyboard. The Keypad on the right allows you to choose a note's value (whether it be a crotchet (quarter note), minim (half note), etc.) and you can press a key on the keyboard corresponding to the note letter and it will input the note – the program will automatically adjust any rests or other notes that you have in the bar so that they are the correct values. The green line you can see on the score is the current

Figure 1.3 *– showing the Play ribbon.*

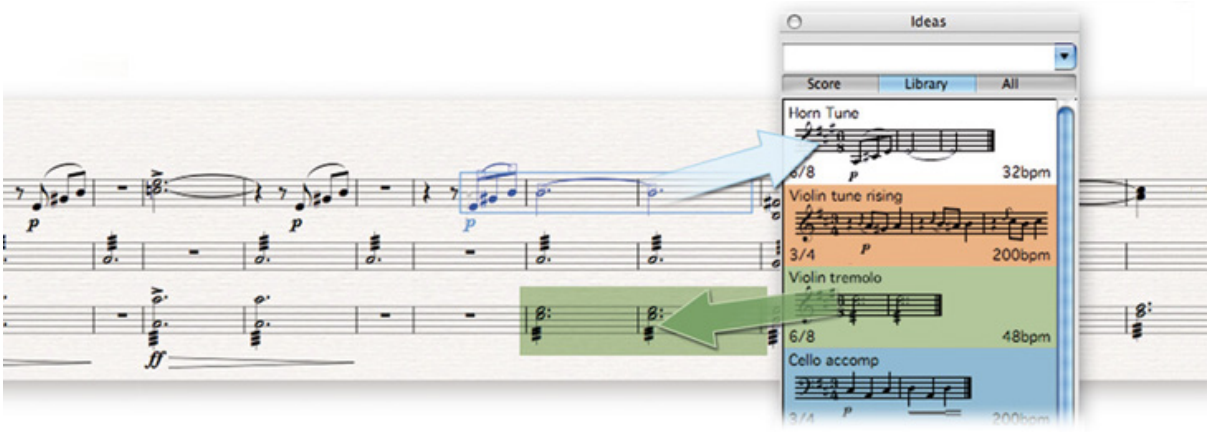


position where a note can be input; if you were to input a note, it would appear on this line on the part selected. The Triplets menu will change the notes that you insert into triplets, depending on what degree of triplets you select. You can also add accidentals (sharps or flats) by clicking on any of them in the Keypad, selecting the notes you wish to apply them to and pressing a key combination to apply them.

Once you have input your notes for the different parts, you can play them back to test them by using the Play ribbon controls at the top of the window shown in **Figure 1.3**. The Mixer button brings up this mixer interface in the bottom half of the screenshot. These control the individual parts within your score while playing them back. The sounds are generated using MIDI and a sample library. For each channel, at the top, are audio sends to reverb and chorus effects. The small knob in the centre is the pan control – this controls how far left or how far right the instrument is in the stereo field. Below that are S (solo) and M (mute) buttons – these can be used to only play a certain part or mute it (so you can't hear it). Below that is the level fader, i.e. the volume control for the part.

There is also an 'Ideas Hub', a library that contains clip of notes of various changes in pitch and dynamics, suitable for various instruments and parts of the score, which the user can insert into their score if they have run out of ideas; the program then transposes the clip into the key of the user's score. The Ideas Hub window, as shown in **Figure 1.**4, has over 1000 pre-written clips that you can insert into your score. The search box at the top searches for the names of the clips; each clip name is specific to the instrument that it is designed for, so if you searched for 'drum', you would get drum rhythms back. Double-clicking on a clip within the ideas window lets you preview the sound of it. The Score tab at the top will show any idea clips that you have created within your own score – you can do this by highlighting some notes and pressing a key combination to store them in the list, along with all of the note and rest values, any notation and dynamics. You can drag and drop any idea into the score by dragging and dropping.

Figure 1.4 *– showing the Ideas Hub*



While Sibelius's workflow is efficient and intuitive regarding its ability to input notes by using a keyboard, its versatility in composing large scores and preview either the whole piece or each individual part, it is quite overwhelming for any non-musicians, as if you want to create a melody, you have to enter all of the notes manually, and knowing what sequence of notes to use to give the correct note intervals for a given key requires music theory knowledge, something which not all musicians, particularly those who are not classically trained, have. That being said, while you can

insert pre-built ideas to ease your creative process, there is no provision for the creation of randomly-generated notes.

2.2.2 MuseScore

While Sibelius is premium, off-the-shelf software, MuseScore, developed by Werner Schweer, is an open source alternative. It provides support for multiple types of notation for multiple genres, e.g. jazz scores are different to classical scores. It also separates different instruments into their respective genres and categories.

The score wizard, as shown in **Figure 2.1**, opens when you create a new score in MuseScore. For your arrangement, you can select from a wide variety of instruments appropriate to various genres and arrange which instruments go on which staves on the score before you start inserting notes. I argue that this is more organised than Sibelius since it is clearer how you wish to arrange your notation when you load up the project.

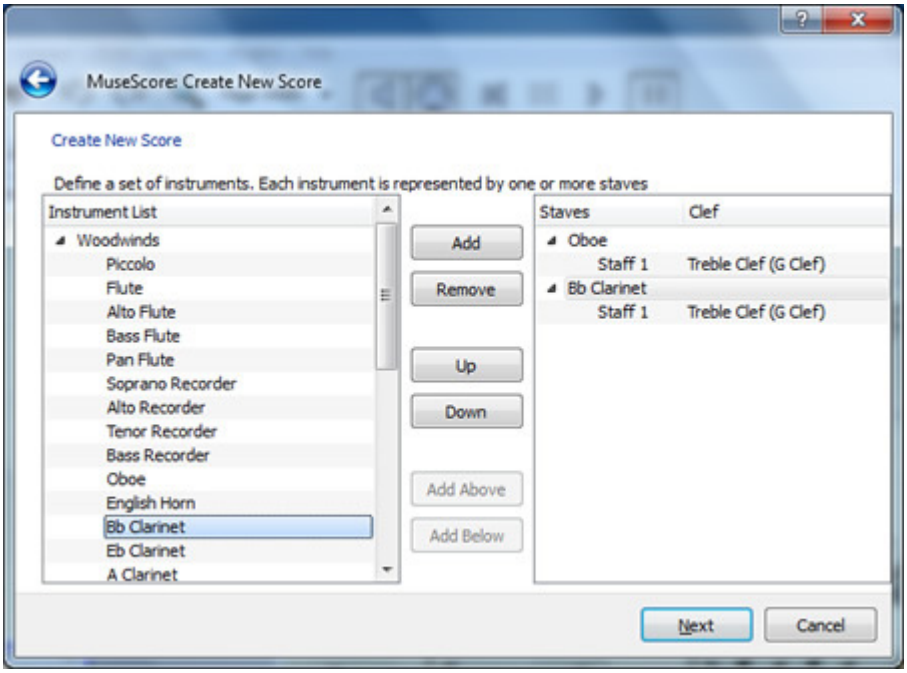


***Figure 2.1** – the score wizard in MuseScore*

Figure 2.2 shows a typical arrangement. Note how all of the options for inserting any notation are on the left hand side – while Sibelius also keeps notation organised within one tab, with this arrangement, it is much easier and faster to get to different types of notation. All of the notes that you can insert are along the top ribbon.

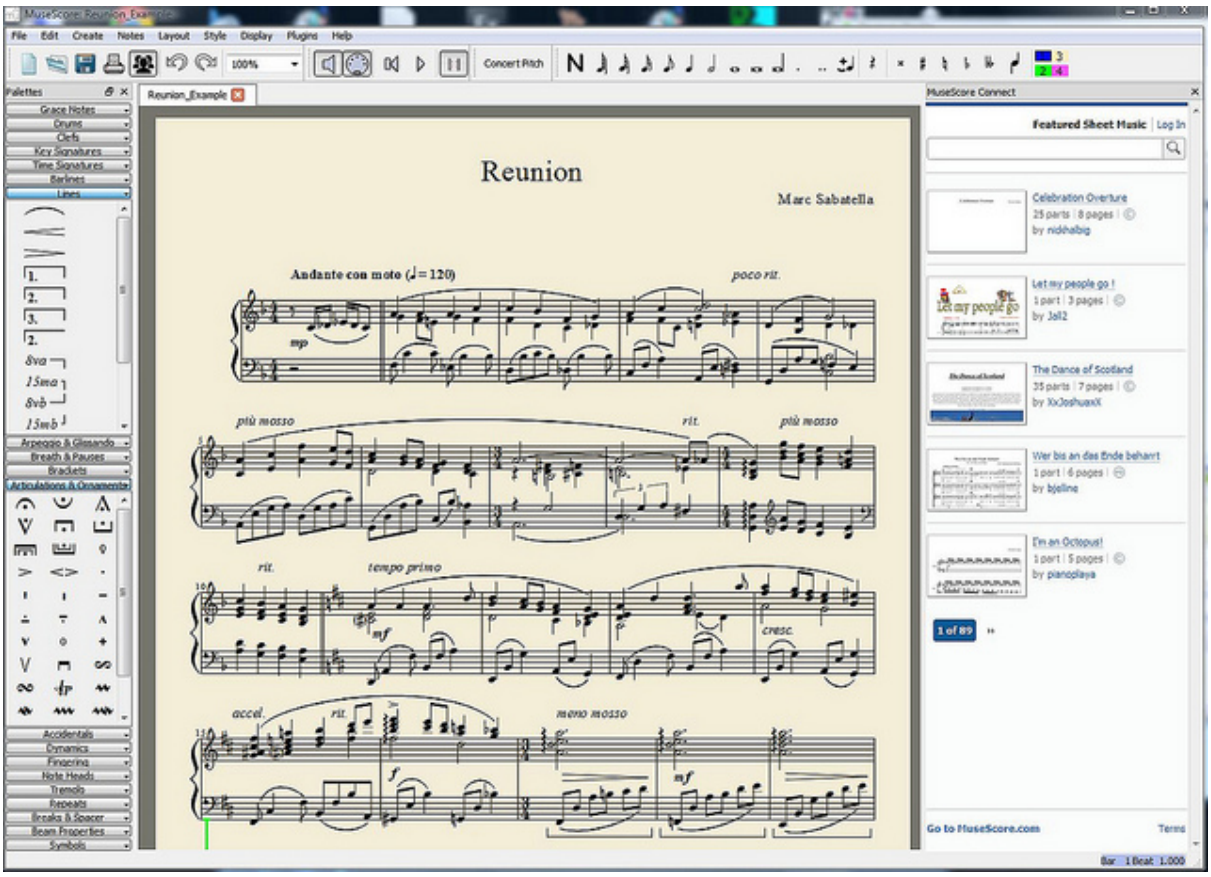


Figure 2.2 *– an arrangement in MuseScore*

Just like with Sibelius, you can use a MIDI keyboard to enter notes. While composing, you can add a variety of note tablature, including normal notes, slashes, guitar fret diagrams, tabs and more. However, that does not help a musician who is struggling with what notes to write down and what notation to use for effect because they have no inspiration; this problem is even more prominent in MuseScore because unlike in Sibelius, there is no Ideas Library. This is the program that I aim to solve with my program.

With both packages, you can print your score, so that it can be played by musicians.

2.3 Identification of Prospective Users

My program should have a wide range of possible users, although I will primarily focus on young musicians who don't have very much experience with composition software like Sibelius and MuseScore; many musicians who are not classically-trained also don't need to write complex scores.

2.4 Identification of User Need and Acceptable Limitations

I have extracted some potential user requirements from an interview with my client, who is a musician looking for inspiration, suggested by what she said and I have collated them below:

2.4.1 Acceptable Requirements

- She prefers the idea of using a database-driven program with multiple accounts rather than an offline program with one user. Since this would require no additional creation of data compared to storing the program files locally, this would be achievable.
- The program's work flow needs to be fast and efficient by not including any unnecessary features and by allowing for fast shortcuts – she says that she wants to improve her creative process, although she doesn't have much time for creating music as other tasks take priority.
- The program should allow for composition with various keys that can be chosen by the user when entering details for each clip – she wants to compose music with varying style influences.
- The program must be able to use clips of notes with fast retrieval from the database, i.e. clips are only downloaded if necessary – she often finds it easy to write songs by creating rough ideas before forming the final idea, but as she has limited time, there must be no delay caused by unnecessary select statements going to the database. With this approach, the user can still select clips from a menu to download by the program only selecting the names of the clips from the database, and selecting the note data from them if the user chooses to download a clip, using different select statements.
- She wants the program to be able to create chords as well as monophonic melodies – when creating a new clip in the sequencer, the user can choose whether the clip generates a chord or a melody; different note-generating protocols and options available to the user are used depending on which option is chosen.

2.4.2 Unacceptable Requirements (Limitations)

- She stated that she wants the program to real-time record and manipulate sounds. In its regular form, Microsoft Visual Studio, which I will be writing the program in, does not accommodate the manipulation of sound files – you would need a third party plug-in. Additionally, my program is not about editing sound – it is about generating note data.
- She also likes how, in other music-making applications, like Steinberg's Cubase, you can be creative and make up new arrangements. This would suggest that she would like the ability to custom-write your own notes or edit existing notes in my program – this would detract the program from its purpose as a music writing assistant rather than a complete music writing package, hence I will not include a feature to change notes; this will probably not be possible in the time that I will have available to complete this project.
- She wants to compose with many different genre influences. I will not allow the user to select from various musical styles (such as jazz, classical or pop) to be generated, as this, to me, is an unnecessary feature and would take too long to program for this project, although I should be able to allow for different musical keys.
- She has emphasised that other programs that she has used have user-friendly environments; this would suggest that a suitable solution would be a graphical user interface, such as with a Windows Form Application. She also stated that she wants the environment to contain colour coding for different parts of the program; e.g. the menu strips could be a different colour to the clip sequencer. With the time that I will have available, writing a Windows Form Application would not be achievable; it is also hard to colour-code different areas of a Form Application without it looking tacky. Therefore, I will create a console-based

application and colour-code different areas of the text. I will take input from the arrow keys on the keyboard to let the user select clips within the arranger screen – the highlighting of clips can be done by changing their colour.

- She stated that she would like the program to be able to work with chords as well as monophonic melodies. This could be implemented, but the algorithm used to generate notes will be complicated enough; generating chords of notes would be exponentially more difficult, as it would have to generate multiple notes in key and time signature, while ensuring that the intervals between the multiple notes are suitable. I am aware that I may not have the time or ability to implement this feature. I will only develop the program to work with melodies.

2.5 Data Sources and Destinations

No data is stored on local secondary storage apart from the program's executable file. All other data is stored in one database with three tables.

The first table holds the users' account details, such as usernames and passwords, the second holds any saved note clips belonging to a user, stored as strings, and the third holds any saved songs belonging to a user, again stored as strings.

2.6 Data Volumes

In a real life scenario, the database would be stored remotely on a public server. For the purposes of this project however, as hiring a public server would be expensive and unnecessary, I will run an EasyPHP DevServer client running MySQL off of my memory stick, connected locally to the computer, on localhost (127.0.0.1).

All data will be stored as text, except for the record IDs, which will be integers. Storing the users' saved arrangements will use my own encoding format using codes made up of single characters. Therefore, the size of the database will be relatively small.

Each user can have many arrangements and each arrangement could contain thousands of notes, but as I'm encoding the notes as text, each note shouldn't be larger than about 10 bytes in size depending on how I arrange the encoding in the design. Each note clip, depending on the length and if the users store any, shouldn't be more than about 300 bytes. Each arrangement will have additional data, which should be no more than about 100 bytes. Each user record should be no more than about 200 bytes.

2.7 User Data Dictionary

This is a brief set of tables describing the data that the user will input, and what the program will output, in different sections of the program. This is specifically focusing on the perspective of the program from the user rather than the programmer.

The different variables are arranged according to the sections of the program to which they will appear:

Section: **login screen**			
Variable	Datatype	Input (I) /Output (O)	Purpose
Prompt to enter details or create new account	Console output	O	Instruction on what to enter.
Choice of creating new account	Key input	I	User can choose whether to input a username and password to log in, or to press a key to send them to the new account screen.
Username	String	I	User can input the username of their account to log in.
Password	String	I	User can input the password of their account to log in.
No entry error message	Console output	O	If the user did not enter a username or password, an error is thrown.
Incorrect detail message	Console output	O	If the user has entered an invalid username and/or password, an error is thrown.
Processing to Take Place This is the first screen presented to the user when they run the program. The username and password entered by the user are checked against the accounts table in the database to see if 1. The username exists and 2. The password is correct.			
Outcome of Processing The program can either display an error message if the login details are incorrect, lead the user to the main menu, or to the new account screen if that's what they choose.			

Section: **new account screen**			
Variable	Datatype	Input (I) /Output (O)	Purpose
Prompt to enter details	Console output	O	Instruction on what to enter.
Username	String	I	User needs to enter a username for their account.
Password	String	I	User needs to enter a password for their account. This will be hashed before being stored in the database.
No entry error message	Console output	O	If the user did not enter a username or password, an error is thrown.
Invalid username error message	Console output	O	If a user has chosen a username that has already been stored with another account, an error is thrown.
Invalid password error message	Console output	O	If the password does not contain more than 8 characters, it cannot be saved and an error is thrown. This security feature will be discussed amongst other security features in more detail in the design section.
No match error message	Console output	O	If the password and confirm password don't match, this is

			thrown.
Processing to Take Place The user is prompted to enter details. The two passwords that they enter are checked if they match. If so, the password is checked if it's more than 8 characters. If so, the password is hashed before both the username and password are stored in a new record in the accounts table.			
Outcome of Processing Either the new account is created or an error message is shown.			

Section: **main menu**			
Variable	Datatype	Input (I) /Output (O)	Purpose
Prompt to choose a menu option and option list	Console output	O	Instruction on what to choose.
Option choice	Key input	I	User chooses menu option by pressing option's number on the keyboard.
Processing to Take Place The option number that the user enters runs the function belonging to that option. If they enter a different number to those listed, the program does nothing.			
Outcome of Processing The user's option choice is displayed or an error message is shown.			

Section: **new arrangement wizard**			
Variable	Datatype	Input (I) /Output (O)	Purpose
Instructions	Console output	O	Showing what details to enter and where.
Key tonic note	String	I	The user must determine the key of the project and one of two parts of determining the key is the tonic note, such as A, B ♭, C♯, etc…
Major/minor	Integer	I	The other part to determining the key if it's major/minor. It is an integer, because this program will only be able to support simple time signatures (divisible by 4) and the user will only be able to enter how many beats they'd like in a bar.
Time signature	Integer	I	The notes in the project must be aligned to a time signature.
Invalid input error message	Console output	O	If any of the fields are either blank or invalid
Processing to Take Place If the inputs are valid, the details given by the user are encoded into a single string of characters representing the details.			

Outcome of Processing
A new arrangement record is created in the arrangement table in the database, containing the encoded characters. If the details are invalid, an error is shown.

Section: **arrangement loader wizard**			
Variable	Datatype	Input (I) /Output (O)	Purpose
List of arrangements belonging to the user	Console output	O	The user must choose one arrangement to load into the arranger screen.
User's choice	Integer	I	The user enters the number (in the list) of the arrangement they'd like. It is read as a key input and converted to a
Processing to Take Place The program takes the key input from the user and runs a select query on the arrangements table using the chosen number as the arrangement ID. Once the data is downloaded, the encoded string is split into its different objects.			
Outcome of Processing The decoded data is displayed to the user in an arranger. If an invalid number is entered, no action is taken.			

Section: **change password wizard**			
Variable	Datatype	Input (I) /Output (O)	Purpose
Instructions	Console output	O	Showing what details to enter and where.
Old password	String	I	The user must confirm their old password.
New password	String	I	The password to change to.
Confirm new password	String	I	They must enter it again to ensure they typed it correctly.
Invalid password error message	Console output	O	If the password does not contain more than 8 characters, it cannot be saved and an error is thrown.
No match error message	Console output	O	If the new password and confirm new password don't match, this is thrown.
Processing to Take Place If the password is valid, it is sent to the accounts table using an update query.			
Outcome of Processing Either the password is successfully updated and the user is returned to the main menu, or an error message is shown.			

Section: **help screen**			
Variable	Datatype	Input (I) /Output (O)	Purpose
Help information	Console Output	O	Displays my email address to the user so that they can ask for help, and so I can maintain the program.
Key input	Console key read	I	When the user has finished reading the help info, they can press any key to return to the main menu
Outcome of Processing The help information is displayed. Once a key is pressed, the main menu is shown.			

Section: **arranger screen**			
Variable	Datatype	Input (I) /Output (O)	Purpose
Current arrangement name	Console output	O	The user can know which project they're working on.
Bar count	Console output	O	Above each clip is a number representing which bar they fall into within the piece.
Clip information	Console output	O	The user can see all of the clips and their status (saved/unsaved, shown by colour) and their names (if they're saved, they will have a name; if they're not, they'll have a bar number).
Task instructions	Console output	O	Text showing which keys you need to enter to perform the various tasks on the bar currently selected, such as inserting a new clip, saving one or deleting one. There is also an option to choose to display the notes within the clip.
Clip selection	Console key input	I	The user can use the arrow keys to move between clips on the arranger before choosing to perform a task.
Task choice	Console key input	I	Once the user has highlighted the clip they'd like, they can press another key to perform a task on the clip selected.
Processing to Take Place The clips within the arrangement are displayed in a space on the console. They are all given a name, and will be one colour to represent if it's saved, and another colour if it's not. Below the clip space are printed a number of options			
Outcome of Processing The next screen depends on which task the user chooses.			

Section: **new clip wizard**			
Variable	Datatype	Input (I) /Output (O)	Purpose
Instructions	Console output	O	Showing what details to enter and where.
Range of octaves	Integer	I	This tells the algorithm over how many octaves may the notes span.
Range of notes	Integer	I	This tells the algorithm how many notes across the range of one octave the notes may span.
Syncopation randomisation	Integer	I	The user may enter a range from 1, meaning that the notes include little syncopation, to 10, where the notes can have the most syncopation. Syncopation is where the notes' rhythm changes. The more syncopation, the less straight the rhythm.
Processing to Take Place With the given details, the note generating algorithm generates a series of notes, whose note values and rhythm correspond to the details.			
Outcome of Processing The notes generated are created in a list and are displayed to the user and inserted into the arrangement object.			

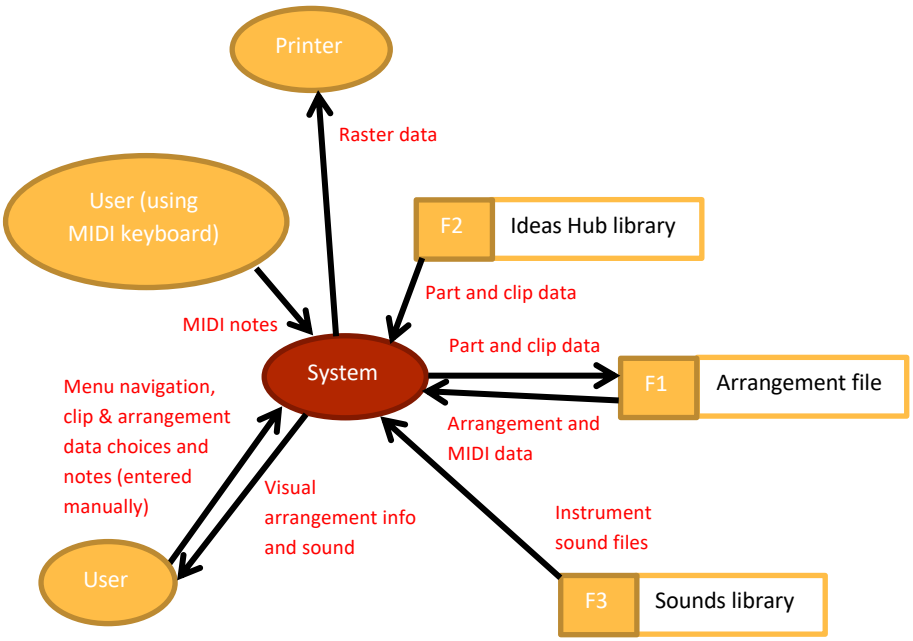
Section: **clip loader wizard**			
Variable	Datatype	Input (I) /Output (O)	Purpose
List of clips belonging to the user	Console output	O	The user needs to be shown in a numbered list which clip they can choose to load.
User's choice	Integer	I	The user the number of the clip they'd like to load
Processing to Take Place Once the user chooses a clip to load, the choice number is used as an ID and the encoded data is queried from the clips table in the database. These are loaded into the program and the notes within them are decoded, and it is inserted into the arrangement.			
Outcome of Processing The new clip is displayed within the arrangement to the user.			

2.8 DFDs of Existing System

This is a basic overview of the processes within Sibelius and MuseScore. There are many features within the programs, although I will only be listing the main ones which I have identified in the investigation into the problem section.

Data going to the arrangement file is done so assuming that the user saves their work.

2.8.1 Level 0

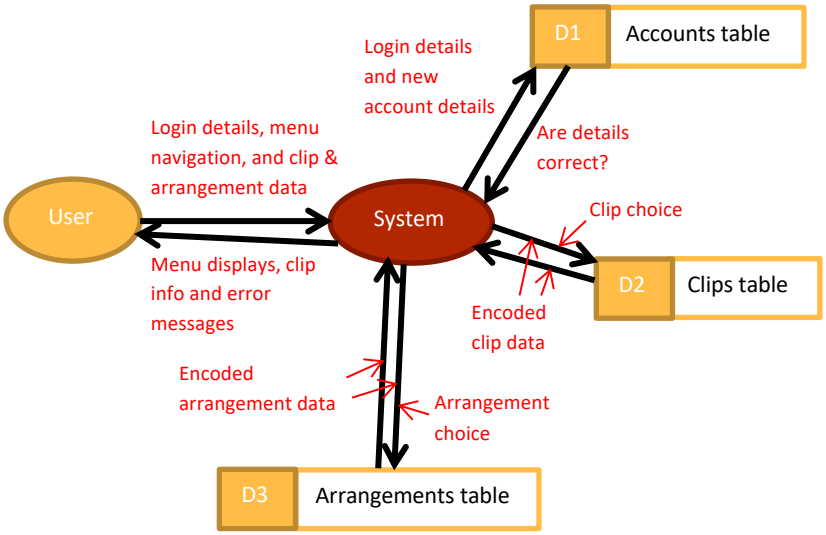


2.8.2 Level 1


Menu navigation and keyboard characters
User (using mouse and keyboard)
5.1. Input notes using onscreen keypad and keyboard
User (using MIDI keyboard)
MIDI input
User
Menu navigation
Part data
5.2. Input notes using MIDI keyboard
Displayable arr. info
4. Add/delete/split/join bars
Part data
Part data
1.1. Create new arrangement
Arr. data
2. Display arrangement
3.1. Add/remove/edit parts
F1
Arrangement file
Arr. visual info
Decoded arr. data
Encoded arr. data
Clip data adapted to arrangement
Project details
1.2. Load arrangement
MIDI data
Clip choice
5.3. Insert Ideas clip
User
Arr. name
3.2. Arrangement playback (Sibelius)
User
Clip data
F2
Ideas Hub library
Mixer adjustment
Mix data
Instrument sound files
F3
Sounds library
6. Applying a mix
Sound
User (speakers)
3.3. Print arrangement
Raster data
Printer

2.9 DFDs of Proposed System

2.9.1 Level 0



2.9.2 Level 1

EM represents an 'error message'.

EM1: Data entry invalid or not all fields entered.
EM2: Account username already exists.
EM3: Old password not correct.
EM4: New password & confirm new password don't match.

3 Objectives of the New System

3.1 Introduction

This section will outline the objectives of my problem, which I will solve by writing the program. These may be solved by custom algorithms, functions and/or custom classes. These objectives are inspired by both what I intend to solve with my program and by what my client has suggested.

3.2 General Objectives

The default screen of the program will be a login screen; the user can then log into an account stored in the database or create a new one. Once logged in, a main menu will be displayed, giving the option to create a new arrangement of clips, load a previously-saved one, change their details, display contact information or log out. Once the user has created a new arrangement or loaded a saved one, the program will display a clip sequencer in a console window (as they can be more easily formatted compared to a form application), in which clips can be inserted and edited. On the same screen, a menu is provided underneath the sequencer giving the user options to insert new or saved clips, to save a clip, to save their sequence or to log out.

When creating new clips, once the user gives the details of their clips, the program will randomly generate a sequence of notes using an algorithm. Each clip will be stored as an object; the notes within each clip are stored as an array within each clip object. The size of each array will be determined by the note values of the notes to be stored, which is determined by the user. The notes will be defined as a custom class.

3.3 Specific Objectives

These objectives are SMART objectives; they correspond to small, individual areas of the new program. SMART means:

- Specific – why the objective relates to specifically one area.
- Measurable – whether or not the objective is measurable given roughly how long the objective will take to execute within the program given its complexity. In cases where a piece of code may execute instantly, that would make it non-measurable, as it would not be possible or practical to necessary to measure a response time.
- Achievable – whether or not the objective can be carried out in the time available and why.
- Relevant – evaluating whether or not the objective is realistic given how important the task is to the working of the program, and/or if my client has requested it, or has very closely suggested requesting it, in the interview. See the appendix for the question responses.
- Timely – whether or not the task can be achieved in a reasonable timeframe and why.

I have given each objective a number so that it can be easily referred-to later on in the project. I have also specified if the objective is regarding the user's experience or the system (where operations are hidden from the user).

3.3.1 Specific Objective 1

User	To allow the user to log in
S	This objective is not specific because it relates to the user entering their details into the login page, and them being looked up to the database, which are two separate tasks.
M	This is measurable as connecting to the database to validate the login could take some time depending on the speed of the query's execution. The time spent by the user entering the details is negligible.
A	This is achievable as it is a simple task and should not interfere with other parts of the project, although querying the database will need testing.
R	This is realistic as without logging into an account, the users of the program cannot save and protect their work. My client has also asked for data to be stored in accounts in a database.
T	This is timely, though not by much, as it would take no more than 15 minutes to code.

3.3.2 Specific Objective 2

System	To be able to hash a password (both for storing the password and for hashing a value which the user enters as a password when they log in to be compared with the stored password)
S	This is specific because the hashing algorithm is one part of storing the password in the database, or matching the user's input with the stored value.
M	This is not measurable as, since this is only referring to the hashing algorithm and not the lookup to the database, the code should run instantly.
A	This is achievable because hashing involves replacing the normal characters in a string with nonsense characters, something which can easily be done with string manipulation and loops.
R	This is realistic as hashing a password for a database complies with basic de facto standards of database security.
T	This is timely, since I will need to look up how to write a suitable hashing algorithm and test that the characters are manipulated correctly. This could take 40 minutes to write and test.

3.3.3 Specific Objective 3

User	To allow the user to create a new account
S	This is not specific, since this is referring to both the entering of the data for a new user and storing the data in the accounts table in the database.
M	This is measurable, as it could take some time for the user details to be stored in the accounts table in the database. The time spent by the user inputting the data is negligible.
A	This is achievable as it just needs some SQL queries to insert data into the database. I will however need to look up the correct syntax for doing so.
R	This is not completely relevant, since some of the attributes that I originally wanted to contain for each account are not necessary to the working of the program, such as country, email address and phone number. My client has not requested that extra information is stored about users; not following this would be breaking the Data Protection Act. Usernames and passwords will however still be used to allow the accounts to at least work.
T	This is timely as there may be many queries and testing involved; it could take about 50 minutes to code.

3.3.4 Specific Objective 4

User	To display the main menu
S	This is specific, as displaying the menu to the screen is a single task.
M	This is not measurable, as taking the input from the user and converting it into an integer should run instantly.
A	This part should be easily achievable, as all it is doing is writing text to the console, giving menu options and prompting them to input a choice.
R	This is relevant, since the user needs to choose between the different options available in the program.
T	This is mildly timely; it should take only 2 minutes to code.

3.3.5 Specific Objective 5

User	To allow the user to choose an option from the main menu
S	This is specific because taking input from the user is a single task.
M	This is not measureable as the code should display immediately.
A	This is achievable as displaying a menu options is well within my programming capability. To make it straightforward, I will make the user enter the number of the menu item that they would like to use and press enter. Other methods may be too timely to implement.
R	This is relevant since, without it, the user cannot access the different options in the program.
T	This is timely as it should not take more than 4 minutes to code the menu options using an IF statement.

3.3.6 Specific Objective 6

User	To allow the user to input details for a new arrangement
S	This is specific because it deals only with the user input.
M	This is not measurable, since taking the multiple inputs from the user and converting them for use in the appropriate variables should run instantly.
A	This should be achievable, since it easy to convert user input into data types. There may be some format exceptions, but these can be debugged.
R	This is relevant because without it, the user would not be able to work on their arrangement; the program needs to know what musical information to use in the project for the notes to be generated correctly to the user's needs.
T	This part is mildly timely and shouldn't take more than 8 minutes to write because, just like inputting from the main menu, this just involves writing and reading to/from the console.

3.3.7 Specific Objective 7

System	To be able to connect to the database
S	This is specific because this is dealing with the connection object only and not the data itself.
M	This will be measurable, since establishing a connection could take a good few seconds depending on the efficiency of the EasyPHP DevServer.
A	This is achievable because this is the easy part of the using the database, though I need to anticipate connection problems.
R	This is relevant because, without connecting to the database, we cannot access any of the data storage.

T	This is timely and could take 20 minutes to write and debug. I will complete this objective at the start of the code writing, as it is one of the most important parts of the program.

3.3.8 Specific Objective 8

System	To be able to send queries to the database (regarding all queries in the program)
S	This is specific because this deals with the SQL query strings sent to the database after the connection has been established.
M	This is measurable, as it could take a few seconds for a query to execute.
A	This is achievable, as the queries can be written, although I will need to write them in order of category of each database-reliant area of the program to ensure each part is working.
R	This is relevant as, without queries, the database cannot be accessed from within C#.
T	This is timely because, although the queries should be relatively simple, there will be a lot of them. This could take 30 minutes to write and test. I will write all of the queries after I have written the parts that do not rely on the database to maintain work efficiency.

3.3.9 Specific Objective 9

System	To be able to format the select query responses so that they fit the format required within the program
S	This is specific as this is focusing on extracting data from the response strings sent back from the database after it has been queried.
M	This is measurable, as it could take a few seconds for the data to be returned and formatted for the use in the program.
A	This is achievable since building the query strings just needs string functions to extract the right information into the variables required.
R	I would need to look up how to assign SQL queries to variables. This is relevant as, without it, the data from the database cannot be used in context.
T	This is timely because it would involve assigning different parts of a query into multiple variables. In some cases, multiple queries may be needed. To ensure that every query is assigned to variables correctly in every part of the program where they are required, this objective could take 30 minutes to meet.

3.3.10 Specific Objective 10

System	To create a new arrangement record in the arrangement table in the database
S	This is specific because this deals with creating the record with data for a new arrangement, while the data has already been input.
M	This is measurable as it could take a few seconds for the data to be sent to the database and the record created.
A	This will need some basic debugging, but should still be achieved easily, because the SQL is quite simple (just an Insert statement).
R	This is relevant since the data for an arrangement needs to be stored in the database.
T	This is timely since it would take about 5 minutes to write and test.

3.3.11 Specific Objective 11

User	To display a blank arrangement page to the screen

S	This is specific because it just deals with displaying the space for an arrangement to be loaded into, along with details of function keys.
M	This is not measurable, as displaying a blank arrangement does not use any data sources, and so should be drawn to the screen instantly.
A	This is achievable as it just requires console statements.
R	This is realistic, since without an arrangement page, the user cannot work on their arrangement. This is the most important screen within the program.
T	This is timely as it will require some experimentation to ensure that the information is displayed correctly, hence this will be a complex task. It could take 40 minutes to write.

3.3.12 Specific Objective 12

User	To let the user choose a previously-saved arrangement (from a list)
S	This is specific because this deals with displaying the different options of arrangements belonging to the user for them to load, to let the user choose and input their choice.
M	This is measurable since taking the names of the arrangements available from the database and formatting them into a list could take a few seconds.
A	This is achievable because it involves easy select statements.
R	This is relevant as the user could not load their saved work without it.
T	This is timely as I will need to make sure that the select statements run correctly. This could take 10 minutes to code and test.

3.3.13 Specific Objective 13

System	To load a previously-saved arrangement
S	This is specific because this deals with the querying and loading part of the user choosing an arrangement to load.
M	This is measurable as it could take a few seconds to query the database to get the correct arrangement data, and for the arrangement to be loaded from the table into a string variable.
A	This should be achievable because it uses simple SQL queries.
R	This is relevant as the data for an arrangement needs to be loaded from the database.
T	This is timely as ensuring that the data is stored in the program correctly could take a few tries. This could take 30 minutes to write and test.

3.3.14 Specific Objective 14

User	To display the previously-saved arrangement to the screen
S	This is specific because this deals with the formatting of the arrangement data and displaying it to the screen rather than the loading of it.
M	This is measurable, since an arrangement could be very long, and it that means that a large number of characters would be searched and decoded. It could take a few seconds to run.
A	This should just involve the use of string functions, so this task is achievable.
R	This is relevant because the user needs to be able to see the arrangement's clips on the screen.
T	This will be timely as I will need to add many conditions for each encoded character in the arrangement data. There could be initial encoding conversion issues when it is first run, and there be formatting issues when the clip data is displayed. Therefore, it could take 50 minutes to write and debug.

3.3.15 Specific Objective 15

User	To allow the user to input a new password.
S	This is specific because it deals with one section in the main menu, and this part of it is dealing with the input rather than the storing of the new password in the database.
M	Apart from the time taken for the user to input the data, the code shouldn't be measurable because it should run instantly.
A	This is easily achievable because it's only taking user input from the console.
R	This is not relevant as it is an additional feature that, although it is a de facto standard to be able to change your password for an account-based program, it is not necessary for the working of this program. My client has not specified how they would like the accounts system to work, only that it should be account-based.
T	This would not be very timely – it would only take about 3 minutes to write. Although it is not relevant, I will add this section anyway, because it's non-intrusive to other areas of the program and is fast to write.

3.3.16 Specific Objective 16

System	To save new user details
S	This is specific because this deals with the storing of the new password into the database after the input has been done.
M	This is measurable since it may take some time for the database record to be updated. It could take a few seconds.
A	This is achievable, as it is one of the simplest update SQL queries used in the program.
R	This is relevant, as with the last objective, the changed password would need to be stored in the database.
T	This is only mildly timely, since it only involves one update statement. Writing it and checking that the data is sent to the database correctly could take 10 minutes to do.

3.3.17 Specific Objective 17

User	To display the help screen
S	This is specific because this deals with just displaying text to the console.
M	This is not measurable as it should execute instantly.
A	This is achievable as it just uses a Console.WriteLine() function.
R	This is relevant as without it, maintenance of the program (the user can send me an email outlining problems) would not be possible.
T	This would not be very timely as it would take about 2 minutes to write.

3.3.18 Specific Objective 18

User	To be able to log out
S	This is specific because logging out is one action from the user.
M	This is not measurable as the code should execute instantly.
A	This is achievable because it just involves clearing variables relating to the current user logged in.
R	This is relevant as otherwise, other users won't be able to use the program; only one user can

	be logged in at a time.
T	This may be timely, since I would need to ensure that all of the variables that need to be cleared have been cleared. This could take 6 minutes to write and test.

3.3.19 Specific Objective 19

User	To be able to insert a new clip into an arrangement
S	This is specific because inserting a clip is a single task for the user, as it deals with them inputting the data for a new clip.
M	Apart from the time spent by the user to input the clip data, this won't be measurable, since the input functions would run instantly.
A	This is achievable because it is just uses a few input functions.
R	This is relevant as, without it, the user cannot work on an arrangement.
T	This will be timely because I will need to ensure that all of the data is input and validated correctly. This could take 10 minutes to write and test.

3.3.20 Specific Objective 20

System	To be able to generate a melody sequence for a clip using the details given by the user
S	This is specific because this deals with the generation of the notes for a clip after the user has input the details for the clip.
M	This is measurable, since this is a relatively complex custom algorithm. It may be inefficient, so it could take a good few seconds to run.
A	This is achievable because I know roughly what pseudocode to use at this stage.
R	This is relevant as this is part of the core functionality of the program.
T	This is timely, since the algorithm will have to be able to generate notes at random but in accordance with the user's input guide and so that they are in the correct key and time signature. This will require the most debugging out of any section in the program, and I will need to write some test runs before the fully-featured generator. Therefore, this could take in excess of 90 minutes to write and test.

3.3.21 Specific Objective 21

System	To be able to generate a chord sequence using the detail given by the user
S	This is specific because, just like in Objective 20, this deals just with the generation of notes.
M	Just like Objective 20, this is a complex custom algorithm, hence it could take a good few seconds to run.
A	I think that this is not achievable in the time available as I don't feel that I will have time or the ability to implement this. Generating a monophonic melody as in Objective 20 is going to be hard enough; generating a series of chords is going to be even harder.
R	This is relevant as my client has suggested that the program's note generation algorithm should be able to generate chords as well as monophonic melodies.
T	This would probably take in excess of 120 minutes to write and test due to the complexity.

3.3.22 Specific Objective 22

System	To ensure that any notes generated in either melody or chord generation are in the correct key and time signature to the project

S	This is specific because this refers to one section in the note-generating algorithm, as describes above, which checks the notes and adjusts them accordingly.
M	This is measurable because the code could take a few seconds to run.
A	This is achievable because I understand how I can implement music theory into an algorithm.
R	This is relevant because this forms the basis of generating a series of musically-correct notes.
T	This is timely due to the number of logic errors that could occur. It could take 60 minutes to write and test, and will form part of Objectives 20 and 21.

3.3.23 Specific Objective 23

System	To be able to save a clip to the database
S	This is not specific because this would refer to the encoding of the note data within a clip and then sending it to the database; that's 2 parts.
M	This is measurable because it could take a few seconds to encode the clip data and then send it to the database.
A	This is achievable because the encoding just involves string functions and inserting the data into the table uses some simple SQL.
R	This is relevant, as my client has suggested that data can be saved, so that they can work on their ideas at a later date. Being able to save the individual clips as well as the whole arrangement makes the program's workflow more flexible.
T	This will be mildly timely as the encoding and inserting will need to be taste; there could also be data inconsistencies, so this could take 40 minutes to write and test.

3.3.24 Specific Objective 24

User	To be able to load a saved clip into an arrangement
S	This is not specific because this deals with both the loading of the clip data from the database and the decoding of the data to be used within the arrangement.
M	This is measurable because it could take a few seconds for the clip to be loaded and decoded.
A	This is achievable because this uses straightforward SQL queries and string functions.
R	This is relevant because, if the user has saved clips to the database, they need to be able to load them. All of the clip features fit in with my client wanting the program to be able to record and draw up quick ideas.
T	This is timely since it may take some time to make sure that the data is decoded correctly. This may need about 40 minutes for writing and testing time.

3.3.25 Specific Objective 25

System	To be able to save an arrangement to the database
S	This is not specific because this would need to deal with both the encoding of the arrangement data and sending it to the database.
M	This is measurable because it may take a few seconds for the data to be inserted into the database.
A	This is achievable because this just involves string functions and basic SQL.
R	This is relevant, since the user would need to save their data if they want to work on it again later. This complies with my client's suggestion of wanting to come back to ideas later.

T	Just like in Objective 24, I need to ensure that the data is encoded correctly, so I will need about 40 minutes to write and test this.

3.3.26 Specific Objective 26

User	To be able to exit an arrangement
S	This is specific because exiting the arrangement is a single task.
M	This is not measurable because the main menu should be displayed instantly.
A	This is easily achievable because it just involves drawing the main menu to the screen and emptying any variables relating to the arrangement being worked on.
R	This is relevant, since the user may wish to work on another project when they are bored with the current one. My client said that they would like to be able to make quick ideas and leave them for another time.
T	This would be only mildly timely, as it would take about 5 minutes to code.

3.3.27 Specific Objective 27

User	To be able to colour-code different areas of the display
S	This is not specific, because this could be referring to changing the colour of any part of the program's interface.
M	This is not measurable because each part of the console window which needs to be coloured will be coloured instantly.
A	This is achievable because it is possible to change the console writing colour in C#.
R	This is relevant because my client has asked for different areas of the program to be colour-coded.
T	This would be timely to code, ensuring that all console colour functions are correctly implemented, which could take 30 minutes.

4 – The Solution

4.1 Introduction

This section outlines various options for types of forms that the user can interact with, why they are appropriate/inappropriate, and why I chose the solution that I will be using.

I will be using Microsoft Visual Studio, using C#, to program this project, as that is what I have been provided with by my college and is what we have been trained with. The use of classes and functions will be based around C#'s dynamics.

4.2 Appraisal of Alternate Solutions

There are three main solution methods to this program when developing in Microsoft Visual Studio: a Windows form application, a web form application or a console application.

A Windows form application would be useful since that is the de facto standard for non-web applications, at least on Windows machines. This is because there is a variety of form controls available which can be formatted to a professional-level design rationale. You can also give different fonts to different sections. However, it can only allow for a fixed number of purposes, since the

design of the form controls can be limiting for a project like my clip arranger, which uses custom-formatted data and drawing. It would be too time-consuming to format a series of clips containing musical notes to fit the design of form controls.

A web application can take away much of the processing from the user's computer, due the provision for server-side scripting. It is also more flexible at arranging graphics; loading graphical content into divs is more dynamic than using a set of fixed image boxes. Websites are also the standard for account-based applications, as they can provide the user with a lot of information in a short space of time. However, even though my program does use some complex custom algorithms, they do not require enough power to warrant the use of server-side scripting which, even if I didn't use that, web programming takes a lot longer to code than writing an offline program, as I would need to edit the HTML code as well as the C# scripts. Formatting and management of variables is also much more time-consuming in a set of web forms than in an offline program. Additionally, the relatively-small number of features of my program means that a website would be too overkill for displaying a small amount of information.

In terms of development, it would be bad practice in the case of this project to use a waterfall development model, where once a phase, such as analysis or design, is completed, it cannot be revisited. This is because my client may have been ambiguous in what they need out of a piece of music software, and my failure to interpret what those needs are would lead to an inconsistent analysis. Also, as I am acting both as the analysist and developer, and since the waterfall model is designed for workflows where the developer talks to the analysist, and the analysist talks to the customer, it would be inappropriate and inflexible to lock down areas with the project progression.

4.3 Justification of Chosen Solution

I will be using a console application for my program. It is the simplest method in which to convey my chosen design, as you can freely arrange all of the controls. It is also possible to write some parts of the console in colour; my client suggested that the program contained colour, and it is harder to produce blocks of colour in a form application. Although it means that all of the processing is done on the client's computer, this should not be a problem, as the hardware should be powerful enough. I am using encoding and hashing algorithms, but they are not as demanding as encryption algorithms that would require a server, such as the RSA algorithm.

This means that I will be able to easily display the clips and the notes within them, and the different sections, in different colours, without being limited to the dynamics of form controls. It is also easy to take key input from the user; much of the program will contain key shortcuts.

For development, I will use the spiral model, which allows me to go back a step, such as from the design to the analysis, in case my client or I are unhappy with how the program in its state is meeting the client's needs. As I write this, I have started on a prototype program, which I will develop and show my client before finishing the design; I can also make changes to this analysis section if necessary. Of course my client may have no significant issue with the program, but as they were slightly ambiguous when answering the interview questions, perhaps due to the fact that I conducted them to a casual standard, I need to be able to cover myself for any desired changes.

5 Abstraction Models

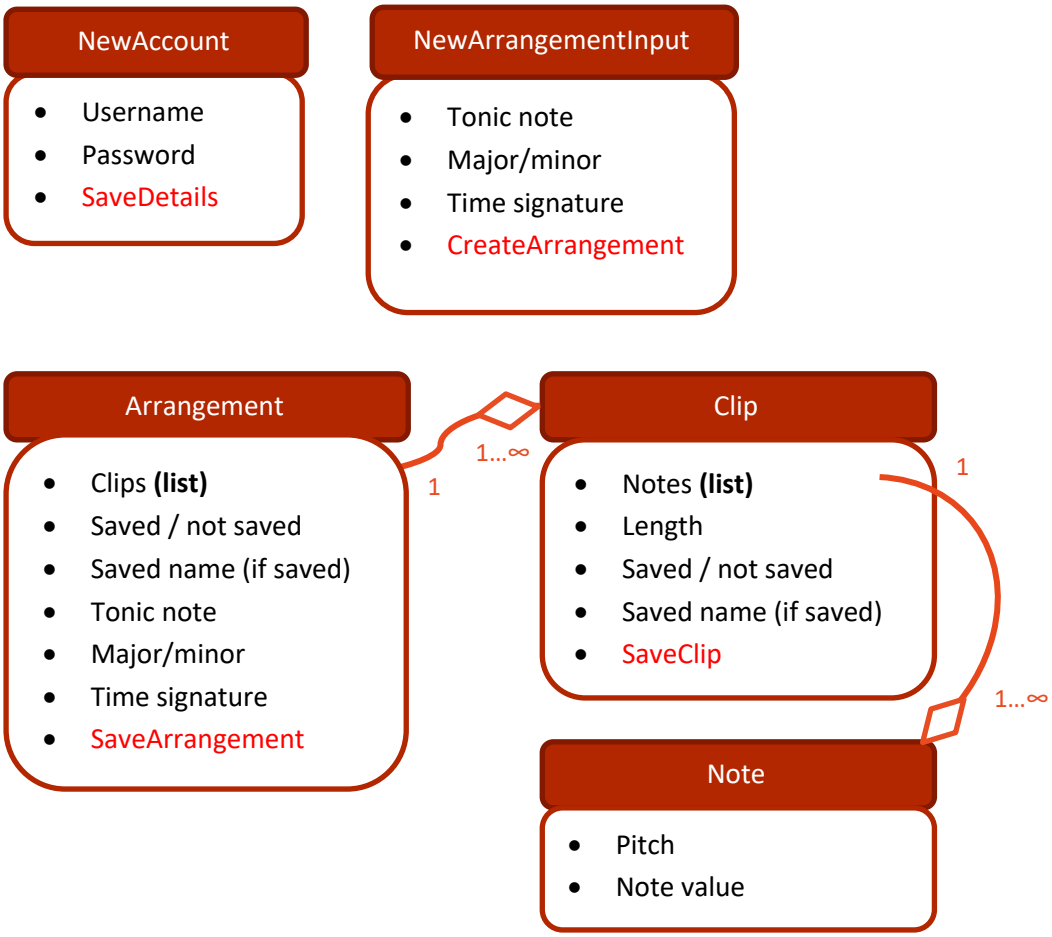
5.1 Introduction

This section gives a basic outline of the relationships between any database entities or objects used within the program. As explained in section 4.3, these will be subject to change throughout the design section, pending the outcome of the development of the prototype program, which I will work on at the same time as doing the design.

5.2 OOAD Analysis

This program will be dealing with objects that contain multiple attributes, hence we must analyse these. They will all be defined as classes for consistency.

The initial relationship of the objects to be created in my program is as follows, along with the attributes contained within them. Black text represents variables and red text shows methods:



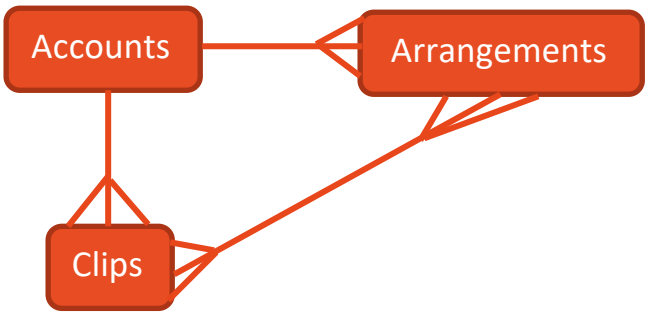
To summarise, the *NewAccount* is used for taking in input from the user for a new account, and saving the details to the database. *NewArrangementInput* deals with the input of data for a new arrangement and the creation of a new arrangement in the database. *Arrangement* is used for the storage of an arrangement that the user is using; it can save the arrangement to the database. *Clip* is used for storing data about a clip; it can save the clip to the database. *Note* stores the pitch and note

value of a single note. Many notes can be aggregated in one clip, and many clips can be aggregated in an arrangement.

The organisation of the objects may change in the design section.

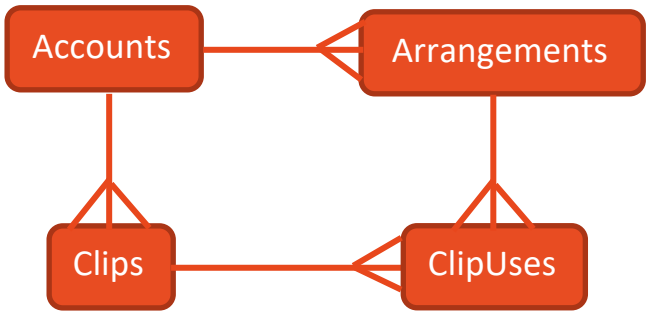
5.3 ERD Model

The initial cardinality of the entities in the database would be as follows:



It is only possible in my system for one user to use many arrangements. Additionally, many clips can be saved to one user account. However, as the arrangement data needs to know which saved clips (if any) are in an arrangement, there must be a relationship between the arrangement and clips entities. It is possible for many arrangements belonging to a user to contain one clip and it is also possible for many clips belonging to a user to be used in one arrangement. This creates a many-to-many relationship, which creates a circular reference, which is bad, as it leads to data inconsistency.

Therefore, we need an extra entity between Clips and Arrangements to separate the use of the clips from the arrangements, and the clips and arrangements themselves. I will call it ClipUses:



Now, it is possible for a clip to be used multiple times, and for multiple uses of clips to be used in an arrangement.

The *Accounts* table stores usernames and passwords. *Arrangements* stores the property data about the arrangements stored by the users, which users they belong to, and any unsaved clip data within them. *Clips* stores any saved clip data (containing notes and rests) by users, and the users that they belong to. *ClipUses* stores the uses of any clips within any arrangements; it stores which clips are used and which arrangements they belong to.

Design

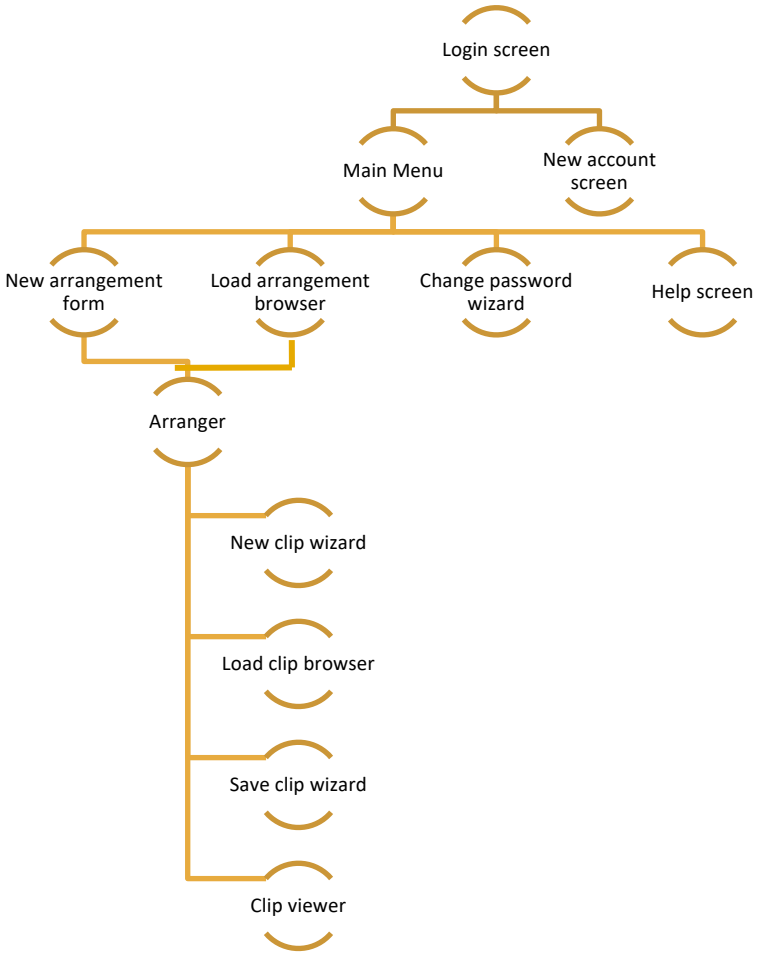
1 Overall System Design

1.1 Introduction

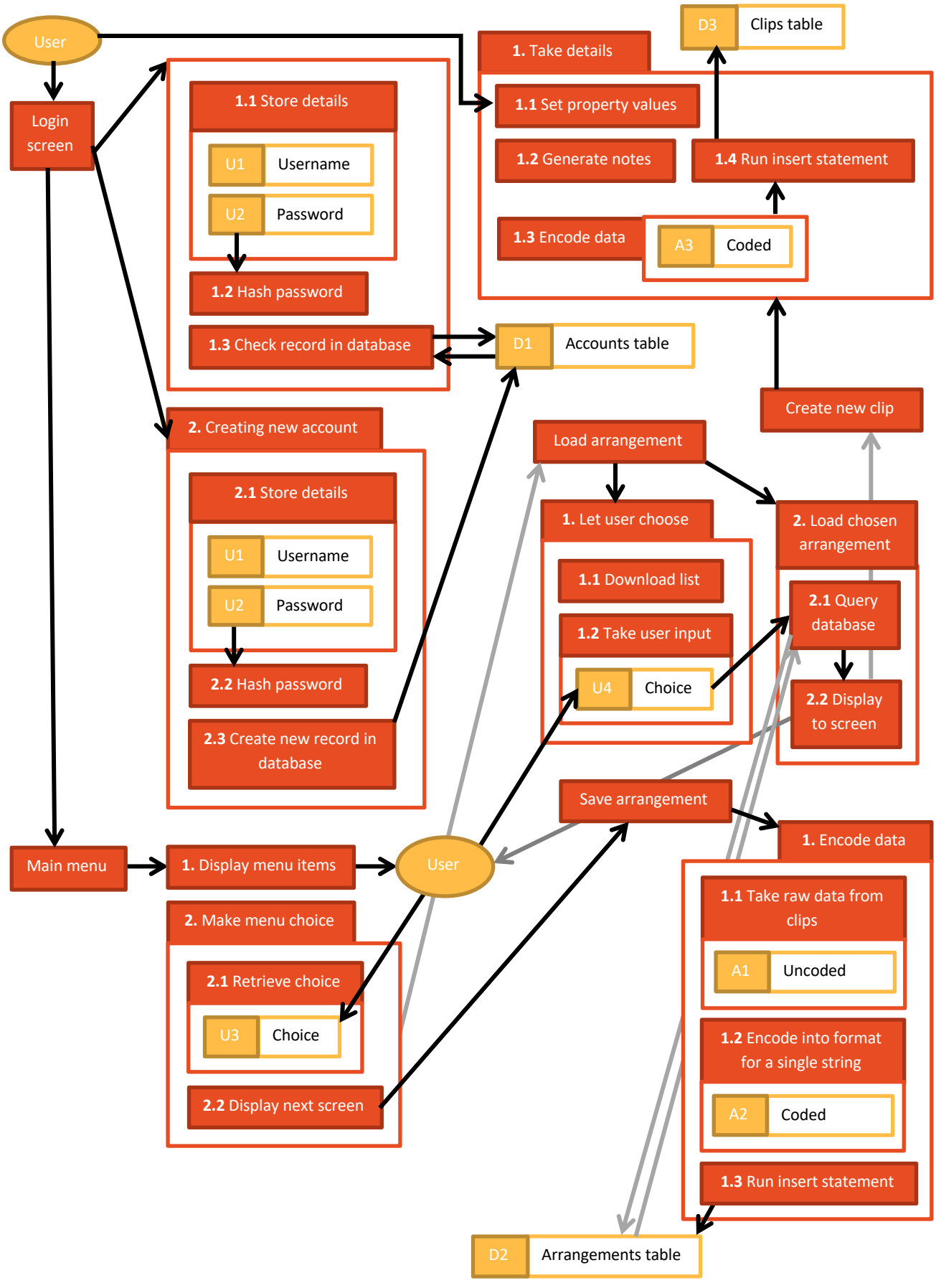
This section deals with how the data is organised within the system, with how the different methods relate to each other and how the data flows from one method to the other.

1.2 Modular Construction of System

As the system is designed to emulate a graphical user interface, I will be displaying the information in various forms. The navigation between the forms available is shown here:



1.3 DFD to level 2 of Proposed System



2 Data Requirements

2.1 Introduction

2.2 Programmer Data Dictionary

This section deals with the different variables used in the different classes within the program, with the class **Program** being the class which is run by the compiler first.

While the user data dictionary in the analysis dealt with the variables from the perspective of the user, this section shows the variables with respect to how I use them within the program. This means that we need to consider scope with these variables, or how accessible within the class and to other classes.

These variable lists are based on the class definitions in **Section 3.4** and the user data dictionary in the analysis.

I will name all public variables in capital case and private variables in camel case.

Class: **Program**			
Variable	Datatype	Scope	Purpose
SqlConnection	MySqlConnection	public static	The object allowing for connection to the database. This is public as it is accessed in all classes.
CurrentUserID	int	public static	The ID of the current user account being accessed.
newAccountChoice	console key	private	When logging in, if the first key that the user inputs is F1, this is a prompt to the new account screen. Without this variable, the new account screen can't be shown.
usernameInput	string	private	Used for logging in.
passwordInput	string	private	Used for logging in.
newUsername	string	private	Used for creating a new account.
newPassword	string	private	Used for creating a new account.
confirmNewPassword	string	private	Used for creating a new account, where password needs to be entered again for validation.
optionChoice	console key	private	Given by user on the main menu.
tonicNoteInput	string	private	Used for creating a new arrangement. Inputs key note.
majorMinorInput	string	private	Used for creating a new arrangement. Inputs whether the key is major or minor.
timeSignature	int	private	Used for creating a new arrangement. User inputs number of beats in a bar.
arrangementIDChoice	int	private	Used when loading an arrangement, where user chooses

			ID number of one they'd like to load.
oldPassword	string	private	Used for changing password, where existing password is validated.
newPassword	string	private	Used for changing password, where new password is entered.
confirmNewPassword	string	private	Used for changing password, where new password is confirmed.
rangeOfOctaves	int	private	Used for creating a new clip – user inputs range of octaves for the clip to include.
rangeOfNotes	int	private	Used for creating a new clip – user inputs
syncopationRandomisation	int	private	Used for creating a new clip – user enters a score from 1 to 10 based on how much they'd like their syncopation to change. See **Section 3.3** for more info on what this means.
clipIDChoice	int	private	Used for loading a clip – user chooses ID of clip they want to load.

Processing to Take Place

SqlConnection establishes a connection to the database. The user starts with the option to go to the new account screen by pressing F1. The first key they press is stored as newAccountChoice. If it's F1, then the new account screen is shown. If not, then the user enters their username and password into usernameInput and passwordInput. These are checked against the database to see if they're correct; if they are, the user is shown the main menu, and if they're not, then an error message is shown.

If the user is creating a new account, then the user enters input for newUsername, newPassword and confirmNewPassword. confirmNewPassword must be the same as newPassword, otherwise an error is shown. If the details are OK, a new record is created in the Accounts table.

When the main menu is shown, the user makes a choice as to what option they'd like next; this is stored in optionChoice. The appropriate screen is then shown.

Outcome of Processing

These are the outcomes from the main menu:

1. Create new arrangement
2. Load an arrangement
3. Change password
4. Help
5. Log out

The Help just prints an email address, so that the user can contact me with any questions.
If Log out is chosen, the user is returned to the login screen.

Class: **NewAccount**

Variable	Datatype	Scope	Purpose
username	string	private	For storing new username to be saved
password	string	private	For storing new password to be saved

Processing to Take Place

This class is used for taking in and saving the details for a new account. The SetDetails method (see **Section 3.4** for class definitions) sets the values of username and password, with input coming from the Program class. The SaveDetails method then checks if the details are valid.

Outcome of Processing

In SaveDetails, confirmPassword must equal password; if it doesn't, an error is returned; if it does, then an insert statement is run on the Accounts table in the database, so the account has a new record created.

Class: **Arrangement**

Variable	Datatype	Scope	Purpose
clips	list (of type Clip)	protected	Contains the clips within an arrangement.
saved	bool	protected	To flag whether or not the arrangement has been saved.
tonicNoteNumerical	int	protected	The numerical equivalent of the string tonic note which was input by the user. Read **Section 3.3.1** to see what the numerical values are for the different notes. I am using numbers as they are easier to process than characters.
majorOrMinor	bool	protected	Determines if the key is major or minor.
timeSignature	int	protected	Determines the number of beats in a bar.
Sharps()	Bool function	public	Determines whether or not the key signature uses sharps or flats.

Processing to Take Place

The properties are used for the creation of the arrangement. The SetDetails method (see class definition in **Section 3.4**) sets the input values from the Program class. The CreateArrangement method changes the input of the key signature tonic note, which will be a letter, into its numerical equivalent (see **Section 3.3.1** for more info on what this means), and stores it in tonicNoteNumerical. majorOrMinor and timeSignature are just set without processing.

If saved is false, if a user tries to exit the arranger, a message is displayed asking if they want to save changes.

The Sharps function is used within the clip creation algorithm when sharpening/flattening notes.

Outcome of Processing

The properties are used to create a new arrangement record in the Arrangements table (with the CreateArrangement method). They can also be used to load properties from the database (with the LoadArrangement method) and save the properties of an existing arrangement (with the SaveArrangement method).

When an arrangement is created, it is created with the specified key signature and time signature, hence why the properties are needed.

Class: **Clip**

Variable	Datatype	Scope	Purpose
rangeOfOctaves	int	private	Stores the range of octaves chosen by the user.
rangeOfNotes	int	private	Stores the range of notes within the octave chosen by the user.
syncopationRandomisation	int	private	Stores the randomisation factor for the syncopation of the notes, chosen by the user. See **Section 3.3** for more on what this means.
notes	list (of type Note)	private	Stores the notes within the clip.
accidentals	int[]	private static	Stores the numerical values of the notes which need to be changed depending on how many notes there are in the key signature. See **Section 3.3**.
syncopationRandomiser()	bool function	private	Decides whether or not the note value needs changing.

Processing to Take Place

rangeOfOctaves, rangeOfNotes and syncopationRandomisation are used for the creation of a clip. The values come from input via the Program class, using the SetDetails method (see **Section 3.4**). When notes are randomly-generated within the clip (see **Section 3.3** for full details on how this is done), their pitches and note values (lengths) are checked against these properties. If they don't fit, then the note is generated again. The accidentals array stores the pitch values of which notes need to be sharpened or flattened depending on the key signature, where the index corresponds to the number of accidentals in the key signature. If a note is created that needs sharpening or flattening, then its numerical pitch value is changed.

Outcome of Processing

This class is used for the creation of a new clip (given entered details), it can save a clip that's currently being used to the Clips table in the database, or it can load the details of an existing clip from the Clips table.

Class: **Note**			
Variable	**Datatype**	**Scope**	**Purpose**
isRest	bool	private	Stores whether or not the note object represents a note or a rest. If it's true, it's a rest. If false, it's a note.
pitchNumerical	int	private	The numerical value of the note's pitch. See **Section 3.3.1** to see what the numerical values are for the different notes.
noteValue	int	private	The number code for the note value (length) of the note. See **Section 3.3.2** for what the different codes are**.**
GetStringNoteValue()	string function	public static	Returns the written value for the note's pitch, instead of its numerical value which is used for processing. See **Section 3.3.1**.
Processing to Take Place If isRest is true, it indicates that the only property that needs to be used is the noteValue (length), since rests only have a length and not a pitch. GetStringNoteValue will take the pitchNumerical value and run it against an if statement to check what lettered note value it is before returning it.			
Outcome of Processing GetStringNoteValue is used to return a lettered note, such as D, instead of its numerical value, which is 2. This function will be called in Program when the note's pitch needs to be displayed to the user in a format they can understand.			

2.3 Description of Data Records

The database will be stored using MySQL and will contain 4 tables. I was originally going to use 3, but as shown in the analysis, using 3 would create a circular reference.

PK means primary key and *FK* means foreign key.

The first table store user accounts. From the program, options will include either logging in to a previously-stored account or creating a new one.

Account details in the first table will be arranged in the **Accounts** table as follows:

Field	Data Type
PK_AccountID	Integer
Username	Varchar(30)
Password	Varchar(50)

The username and password will be fixed varchars, since it is unlikely that they will be very long.

The password will be hashed by an algorithm within the program to prevent hackers hijacking users' accounts. This is similar to the de facto standard of storing passwords used by large commercial websites. I will demonstrate this segment in Section 3.3

The second table, **Clips**, stores clips of notes saved by the user for future use in projects. The schema of this table is as follows:

Field	Data Type
PK_ClipID	Integer
ClipName	Varchar(50)
NoteData	Text
FK_AccountID	Integer

The ClipName field will store the name of the clip once it has been first saved by the user. NoteData stores data for the notes that it contains.

The third table, **Arrangements**, stores arrangement data saved by the user, including the name, its details, and clips it contains (in ArrangementData), including references to saved clips in the Clips table and any unsaved clips, which will be stored in the ClipContentsData field.

Field	Data Type
PK_ArrangementID	Integer
ArrangementName	Varchar(50)
ArrangementData	Text
ClipContentsData	Text
FK_AccountID	Integer

The Text data type is to be used for the encoded data fields, as Text can support long strings, whilst Varchar is the de facto standard for shorter strings.

As in the second table, the ID acts as a foreign key to the other tables, so that data can be linked and retrieved, and the songs and the clips will be specially-encoded, just like the saved clips.

The fourth table is called **ClipUses**, which will link the clips with their uses in any arrangements:

Field	Data Type
PK_ClipID	Integer
PK_ArrangementID	Integer

No unnecessary data is stored, and when a user decides to delete their account (after they have emailed me first, as you cannot delete the account within the program), all data associated with it must be deleted straight away; this is to comply with the Data Protection Act 1998.

2.4 Data Validation

Below is a list of validation scenarios for when the user is entering their data and any error messages associated with them (for if the condition is not met). All of these variables are within the **Program** class, as this is where all of the data input takes place.

Validation ID	1
Variable	usernameInput, passwordInput, newUsernameInput, newPasswordInput *and* confirmNewPasswordInput

Condition	Cannot be blank
Error Message	Please don't leave the <field name> blank

Validation ID	2
Variable	usernameInput *and* passwordInput
Condition	Must be a username/password that exists
Error Message	The username/password is invalid

Validation ID	3
Variable	tonicNoteInput
Condition	Must be a valid note (such as C, Eb or F#)
Error Message	You have not entered a valid note

Validation ID	4
Variable	majorMinorInput
Condition	Must be either "major" or "minor"
Error Message	You must enter either "major" or "minor"

Validation ID	5
Variable	timeSignature, rangeOfOctaves, rangeOfNotes, syncopationRandomisation, arrangementIDChoice *and* clipIDChoice
Condition	Must be an integer
Error Message	You must enter a whole number

Validation ID	6
Variable	rangeOfNotes
Condition	Must be between 1 and 8
Error Message	Please enter a number between 1 and 8

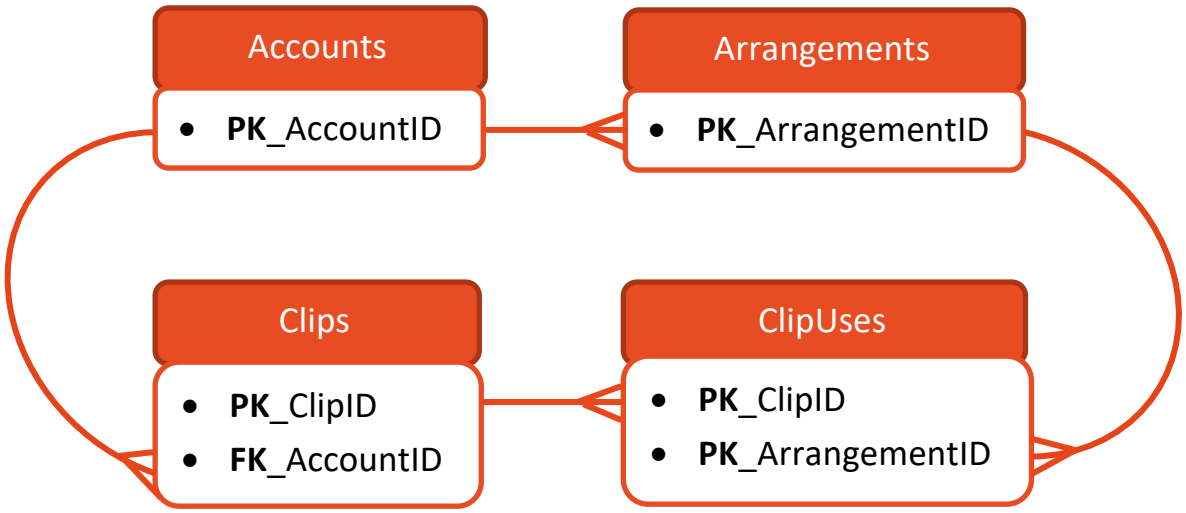
Validation ID	7
Variable	syncopationRandomisation
Condition	Must be between 1 and 10
Error Message	Please enter a number between 1 and 10

Validation ID	8
Variable	arrangementIDChoice *and* clipIDChoice
Condition	Must be a valid arrangement/clip ID
Error Message	Please enter a valid ID number

Validation ID	9
Variable	confirmNewPassword
Condition	The confirm password must be the same as the first password
Error Message	Your two passwords do not match

2.5 Database Design – ERD

Below is a layout of the cardinality between each of the keys in the tables:



To summarise, one account can have access to have many arrangement, which can contain many clip uses, where each use belongs to one clip, and one account can have many clips.

2.6 Database Design – Normalisation Process

Below is an example of what the database would look like if all of the fields were in one table with no normalisation. The table includes one user, who has one arrangement, which contains two saved clips.

Username	Password	Arrangement	Clip
Ross	Duncan	ArrangementOne0061C24A1010BE1F0020 2210422410012021100203211222210E2F0 0202210422410012021100203211222210	ClipOne002022104224100 12021100203211222210
Ross	Duncan	ArrangementOne0061C24A1010BE1F0020 2210422410012021100203211222210E2F0 0202210422410012021100203211222210	ClipTwo002032114225100 12041100206212322410

This table is not in first normal form. To begin with, for each clip, the arrangement and account details are repeated. We separate this one table into two to prevent this:

Username	Password	Arrangement
Ross	Duncan	ArrangementOne0061C24A11BE1F002022 1042241001202110020321122210E2F002 0221042241001202110020321122210

Clip
ClipOne002022104224100 1202110020321122210
ClipTwo002032114225100 1204110020621232410

This now reasonable. However, let's consider if a user had multiple arrangements saved under their account:

Username	Password	Arrangement
Ross	Duncan	ArrangementOne0061C24A11BE1F00202210422410012021100 203211222210E2F00202210422410012021100203211222210
Ross	Duncan	ArrangementTwo <data here>

The account data is still repeated. Therefore, we need to split again:

Username	Password
Ross	Duncan

Arrangement
ArrangementOne0061C24A1010BE1F00202210422410012021100 203211222210E2F00202210422410012021100203211222210
ArrangementTwo <data here>

Clip
ClipOne00202210422410012021100203211222210
ClipTwo00203211422510012041100206212322410

It is no longer necessary to repeat Ross Duncan. It is perfectly possible on the other hand for there to be multiple Ross Duncans, as can we have multiple entities called ArrangementOne and ClipOne. Hence, each record needs a primary key (new or altered keys and fields throughout are shown in red):

PK_AccountID	Username	Password
1	Ross	Duncan

PK_ArrangementID	Arrangement
1	ArrangementOne0061C24A11BE1F00202210422410012021100 203211222210E2F00202210422410012021100203211222210
2	ArrangementTwo <data here>

PK_ClipID	Clip
1	ClipOne00202210422410012021100203211222210
2	ClipTwo00203211422510012041100206212322410

Now, each record is unique. However, considering the encoding that I've made for the arrangements and clips (see **Section 5.2.2** for more detail on how they work), they are not atomic. The code for ArrangementOne can be split into multiple parts.

The Arrangement field can be split up into its saved name, the data about the arrangement itself and the unsaved clips it contains.

The Clip field contains the entire clip data, including both the clip's saved name, the data about the clip itself and the notes it contains.

We need to therefore split up the Arrangement and Clip fields:

PK_AccountID	Username	Password
1	Ross	Duncan

PK_ArrangementID	ArrangementName	ArrangementData	ClipContentsData
1	ArrangementOne	0061C24A11BE1FE2F	E1F00202210422410012021100203211222210E2F00202210422410012021100203211222210
2	ArrangementTwo	<data here>	<data here>

PK_ClipID	ClipName	NoteData
1	ClipOne	00202210422410012021100203211222210
2	ClipTwo	00203211422510012041100206212322410

The ArrangementData field details the details about the arrangement and references to any saved or unsaved clips. The E1F and E2F codes in the ArrangementData field refer to their corresponding clip data in ClipContentsData.

As there are no partial-key or non-key dependencies, our tables are now in second and third normal form automatically.

Now, the tables need to be related together using foreign keys. The relationship between the foreign keys are described above in **Section 2.5**.

We will also need the fourth ClipUses table to link the clips and arrangements together.

These are the final tables:

PK_AccountID	Username	Password
1	Ross	Duncan

PK_Arrangement ID	Arrangement Name	Arrangement Data	ClipContentsData	*FK_AccountID*
1	Arrangement One	0061C24A11B E1FE2F	E1F0020221042241001 2021100203211222210 E2F0020221042241001 2021100203211222210	1
2	Arrangement Two	<data here>	<data here>	1

PK_ClipID	ClipName	NoteData	*FK_AccountID*
1	ClipOne	00202210422410012021100203211222210	1
2	ClipTwo	00203211422510012041100206212322410	1

FK_ClipID	*FK_ArrangementID*
1	1

FK_ClipID and FK_ArrangementID form a compound primary key. It is impossible for two different clip ID and arrangement ID combinations to be the same.

2.7 SQL

Data will be retrieved and inserted into the database using SQL statements. I cannot list all instances of these statements but I can give these examples.

The first statement would be used for creating a new account:

insert into Accounts (Username, Password) values (<username>, <password>)

Soon after, the login function needs to retrieve account information, used for checking if the username exists and if the password is correct:

select PK_AccountID, Username, Password from Accounts where Username = <username>

Once logged in, if the user needs to load a list of existing arrangements to choose one to load, we should retrieve the names of arrangements belonging to that user:

select PK_ArrangementID, ArrangementName from Arrangements where FK_AccountID = <current user ID>

When the user chooses to load the arrangement from the table, they will choose the ID number:

select ArrangementData, ClipContentsData from Arrangements where PK_ArrrangementID = <chosen arrangement ID>

When the user creates a new arrangement, only the chosen name is placed into a new record:

insert into Arrangements (ArrangementName) values (<chosen name>)

If the user wants to save an arrangement after they have chosen a name, the rest of the details are inserted:

update Arrangements set ArrangementData=<arrangement data to upload>, ClipContentsData=<data for any unsaved clips used> where PK_ArrangementID = <current arrangement's ID>

If the user wants to load a list of clips available to use, this is used:

select ClipName from Clips where FK_AccountID = <current user ID>

Where a clip needs to be retrieved, this would be used:

select NoteData from Clips where PK_ClipID = <chosen ID from user>

Where a user would like to save a new clip, this would be used:

insert into Clips (NoteData, FK_AccountID) values (<encoded clip data>, <user ID that clip belongs to>)

As an Easter egg, I will also allow the user to delete a clip from the list of clips, where the deletion is done by this statement:

delete from Clips where PK_ClipID = <chosen ID from user>

2.8 Identification of Storage Media and System Specs

Please note that, as of this stage, I have not tested the full program, I can only give an estimate as to what specs I think *may* be suitable for the program.

2.8.1 The Server

As this is only a database-driven console application, this program will only require a database server.

The program is designed so that several thousand users can access their projects at once; therefore, a hard drive in the server would not suffice, as they are too slow. A better option would be a solid state drive with either a hard drive or a tape drive to back up the data.

All the data is stored in the database as varchars, text or integers; however, consider the following string, which stores data about one clip (see **Section 3.3** and **5.2.2.1** to see what this means):

00202210422410012021100203211222210

This registers as 35 bytes when saved as a .txt file, but this clip data only contains 2 notes and 1 rest (see **Section 3.3**). Imagine a clip which contains 10 times more items than this, which would bring the size up to 350 bytes. Imagine if a user has 100 clips saved to their account; this would make it 35 kilobytes. Triple this to compensate for any arrangement data (maybe the user has 100

arrangements), which themselves can contain unsaved clip data: 105 kB. Now, imagine that there are 1,000 users, so multiply that number by 1,000: 105 MB. And then add 50 kB to compensate for about 1,000 usernames and passwords: 105.05 MB. And this is assuming that the user has about 100 arrangements and 100 clips, and that there are 1,000 users. There could be a lot more.

Therefore, I would suggest for contingency to allow for 2 GB of database storage at the absolute minimum. Because a large amount of data could be accessed at any one time, the data bus needs to be fast; one of the latest and fastest buses available is the SATA Express (based on SATA 3.2), or SATAe, released in 2013, which allows for up to 1,969 MB/s.

Multiple SQL queries would need to be handled at once, so a multi-core processor like the Intel Core i7-47xx. This model has an 8 MB L3 cache and 4 cores; there are more advanced models of the i7, but as we're only processing text instead of, say, multimedia, we don't need to worry too much about the cache size. The 4 cores however will aid in processing multiple queries at a time.

The amount of RAM is additionally important, due to the amount of data being transferred from the network to the secondary storage. The latest version of Windows Server is 2012 R2, which requires at least 512 MB of RAM, but I would reserve 4 GB for the OS for speed. Then, I would triple that to give 4 GB for the database server software and 4 GB to handle any network connections, giving 12 GB of RAM.

To start, I would have a 128 GB solid state drive for the operating system and any software, and a separate 128 GB SS drive for the database storage. I would then have a 3 TB external hard drive for backup.

To summarise, here are my specs for the server:

- Intel Core i7-47xx @ 3.4 – 4 GHz (depending on the sub-model)
- 12 GB RAM
- 2x 128 GB SSD on SATAe bus
- 3 TB hard drive on USB 2.0/3.0
- Windows Server 2012 R2
- SQL Server or MySQL software

2.8.2 The End User

Since this is a console application, my program is not particularly demanding; however, the network connection and network card need to be fast to minimise delay; my client says that they want to get their work done quickly.

Most modern Intel chipset network adapters should suffice, but the program itself could probably run on a Windows 98-era machine: it has minimal graphics requirements, but due to its database connection, it probably wouldn't run very quickly on a DOS-era machine. Many computers of the Windows 98 era had about 128 MB of RAM and a 233 MHz Pentium II processor, and were more than capable of displaying web pages with graphics, so there would be no problem with strings being transferred over a network.

2.9 File I/O

Other than the program's executable file, all data relating to this program are stored in the database, so I will not be using any local files to store any data.

3 Programming Constructs

3.1 Introduction

This is an outline of any algorithms used within my program and how they are implemented, as well as describing how I intend to use object-orientated programming. All of the algorithms will be described in plain English to give a better understanding.

Firstly, I must disclaim that explaining music theory terms to someone who doesn't understand music, or doesn't play a musical instrument, is very difficult, as it is such an advanced field that people who study it get used to, just like computer science; therefore, some of my explanations of how my algorithms work may be vague.

3.2 Identification of Standard Data Processing Algorithms

I am not intending to use any algorithms from the AQA syllabus, though there are other standard algorithms within C# that I will be using.

3.2.1 Hashing

When logging in to a new account or creating a new one, I need to hash the password input. To ensure that the hash cannot be reversed, I will use a one-way function to store the value in the database when creating a new account, then apply the function on any input when the user logs in.

C# has its own cryptography library called 'System.Security.Cryptography'. Within it is the HashAlgorithm class. From the HashAlgorithm class are inherited various subclasses for different hash algorithms, including *MD5*, *RIPEMD* and *SHA*; an object for this must be created before the functions can be used. The function to use is ComputeHash(byte[]), which returns a byte array containing hashed bytes using input from another byte array given as a parameter. The password will be converted to the byte array by extracting each of the characters and converting them into Unicode, using the function Encoding.UTF8.GetBytes(password).

The hash algorithm to use is evaluated in **Section 5.2 (page 69)**.

3.3 Identification of Custom Data Processing Algorithms

3.3.1 Key Signatures

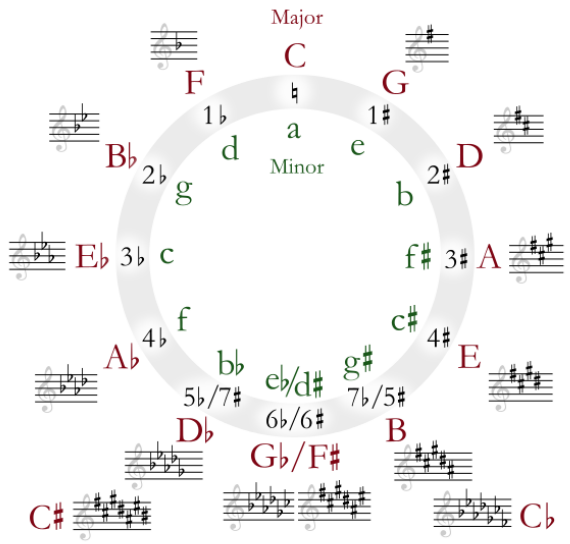


Figure 5.1 – circle of fifths

The main custom algorithm will be the note generation algorithm. This will take in the data provided by the user for a clip, being the range of octaves, range of notes across the octave and the syncopation randomisation. Additionally, the data about the arrangement is used, being the tonic note, whether it is major/minor and the time signature.

In the Western musical scale, there are 12 key signatures – each key signature represents which notes in the scale played need to be raised or lowered one semitone (1 note on the keyboard) in order to fit the key. Raising a note is sharpening it and lowering a note is flattening it, represented by a sharp (♯) or flat (♭) symbol. **Figure 5.1** shows the circle of fifths, a representation of how the tonic (main) note of a key can be raised 5 tones (5 main notes), or a fifth, to transition to the next key, which has one more sharp or flat, depending on whether you ascend or descend the keyboard. If you ascend from C major, you increase the number of sharps, and if you descend from C major, you increase the number of flats.

Each major key also has an equivalent minor, whose tonic note is 3 tones, or a third, below the major key's tonic note.

We can represent the different key signatures in a public 2D integer array, where each row represents a key signature.

We can represent the value of notes in our program by using unsigned integers.

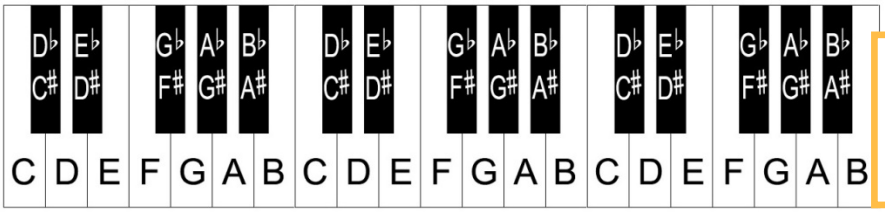


Figure 5.2 – a keyboard representation showing the notes of the Western scale

Our array of key signatures will have the following fields:

1. The tonic note
2. Whether sharps or flats are used, shown by a single character
3. The number of sharps or flats

I will treat C as the first note in the scale. Hence, I will represent the note C numerically as 1. I will represent each white note as a single-digit whole number in ascending order, so the note above C, D, will be 2, E will be 3, F will be 4, up to B, which will be 7.

If I am representing a black note, I will add an extra digit, with 1 representing a sharp and 2 representing a flat. Therefore, the note C♯ will be stored as 11. The note B ♭ will be 27.

In my Note class, I will contain a function called GetStringNoteValue(int noteIntValue), which will return the written value of the note given its numerical value, so putting in 15 will give G♯.

As the major keys can be translated into minor keys by shifting the tonic note by a third, and vice versa, we only need to store an array of the major keys, and if the user chooses their arrangement to be in a minor key, then the major key of the tonic note that they have selected can be shifted down by a third.

Sharps or flats are known in music theory as accidentals. In the key signature array, if a key contains sharps, a 0 is stored in the 3rd column; if it contains flats, a 1 is stored in that column.

The array will be arranged like this (this is in reference to **Figure 5.1**):

Note (written)	Note (numerical)	Accidentals	Sharps/flats
C	1	0	*null*
G	5	1	0
D	2	2	0
A	6	3	0
E	3	4	0
B	7	5	0
C♭	21	7	1
F♯	14	6	0
G♭	25	6	1
D♭	22	5	1
C♯	11	7	0
A♭	26	4	1
E♭	23	3	1
B♭	27	2	1
F	4	1	1

The written forms of the notes (in the first column) will not be stored in the array – they are in the table to show which notes are which. C's sharps/flats is null because C major doesn't contain any accidentals.

Another integer array will store the accidental notes that can be applied to notes within the scale depending on which key the arrangement is in; there are up to 7 accidentals that can be applied. In a major scale, the accidentals are applied to notes in ascending order, so if a key has 2 sharps, they would be F and C sharpened. In a minor scale, the accidentals are applied in descending order, so a

key with 2 flats would have B and E sharpened. The purpose of this array is for the algorithm to look up which notes need to be sharpened/flattened when generating notes.

In minor keys, the 7th note of the scale (in a C scale, this would be B) is raised by 1 semitone.

This is how the accidentals array will be stored as:

Note ID	Note (written)	Note (numerical)
0	F	4
1	C	1
2	G	5
3	D	2
4	A	6
5	E	3
6	B	7

Please note that only the numerical note field is what will be used in the one-dimensional array. The first two columns are here just to show the index of each note and which note is which. The Note ID starts at 0, because indexing in arrays always starts from 0.

3.3.2 Generating Notes

3.3.2.1 How It Should Work

As well as having a pitch, each note has a note value, which shows how long the note lasts for in relation to a bar.

In my program, I will only support simple time signatures, where a semi-breve (whole note, which fills a whole bar) represents 4 beats. There is also compound time, where a whole note represents 8 beats, but I will not implement that for simplicity's sake.

These are the note values that I will support:

- Semibreve – 1 bar
- Minim– 1/2 bar
- Dotted minim – 3/4 bar
- Crotchet – 1/4 bar
- Quaver – 1/8 bar
- Semiquaver – 1/16 bar

A rest is a space where there are no notes; just like a note, a rest can have a length, known as a rest value.

Any note/rest values are represented as the following codes:

Note/Rest Value (relative to 1 bar)	Code
1	0
3/4	1
1/2	2
1/4	3
1/8	4
1/16	5

Change between note/rest values is called syncopation. The syncopation randomisation variable stored about a clip, on a scale of 1 to 10, show how often the values should change.

1 means that no note/rest values change, whereas 10 means that it would change on every note/rest. 4 means that every 10 notes, 4 have their note value changed. The notes which may be changed are determined by 2 counters – one which counts how many notes have been created, and one which counts how many have been changed. Depending on the randomisation choice, the program chooses whether or not to change a note's syncopation based on how many have been changed already and how many need to be changed.

The clip object, in which notes are contained, will contain integer variables representing the range of notes, range of octaves and syncopation randomisation. The clip object also contains the CreateClip method, which contains this algorithm, and a list of notes (containing Note objects).

The range of notes shows how many notes outside of the pitch of the tonic note the algorithm is allowed to generate. For example, if it is 1, only the tonic note will be generated; if it is 2, the note above and below the tonic note can be generated, as long as it is in the range of octaves.

The range of octaves shows how many octaves the clip may contain, 1 being only the middle octave of the keyboard, where middle C resides. 2 means that the octaves above and below the middle octave can be included. The middle octave is represented scientifically as octave 4; therefore, the note C in octave 4 is represented as C4.

The CreateClip method will randomly generate a note value integer and it will be added as a note into the notes list, as long as it meets the criteria, as specified by the properties, and fits within the clip. If it doesn't, the method generates another random note value, and will keep doing so until a valid note is given. This random generation is done by randomly selecting an integer between 1 and 7 which, as shown earlier, represents the notes within an octave.

The algorithm must also randomly choose between including a note and including a rest. This is done by randomly choosing a number; if this number is divisible by 2, a note is created. If not, a rest is created.

3.3.2.2 Working Example

The key signature of an arrangement is E major and the user inserts a new clip. The octave range is 1, the note range is 5 and the syncopation randomisation is 3.

The algorithm first generates a note with a pitch of D2 and a value of 1/4. As the octave range is 1, the clip can only contain notes in octave 4, so the note cannot be used.

The algorithm then makes a note with a pitch of A4 and a value of 1/2. As the note is in octave 4, the octave is OK, and as the note range includes A4 (the tonic note being E4), the note can be used. The syncopation randomiser decides that this note doesn't need to have its syncopation changed.

The algorithm decides to make a note with pitch F4, using the note value of the previous note, which is 1/2 (the algorithm will only use the value of the previous note if it has been included in the clip). The note is in octave 4 and in the range. However, because the arrangement is in E major, and F is sharp in the key signature, the pitch is changed so that it is F♯4. The syncopation randomiser decides not to change this note's value.

The algorithm then decides to make a note with pitch B4 using the value of the last note, which is 1/2. The note is in the right octave and range, and doesn't need flattening/sharpening. However, the syncopation randomiser decides that this note does need its value changing, so it randomly selects a new note value, say 1/4. The note is now a 1/4 bar in length, or a crotchet.

The algorithm then successfully generates 7 more notes and some rests up until the last 1/16th of the bar. It generates a note that is in range and octave, but the only value it can use is 1/8, as that was the value of the previous note. The syncopation randomiser has already changed the value of 3 notes, and as there have been 9 notes already created, and since the randomisation setting is 3, the 10th note cannot be changed. A 1/8 note (quaver) cannot fit into 1/16 of a bar, so the algorithm cannot include another note in this clip.

The clip is then completed, shown in the arranger, and the notes can now be displayed to the user. See how the notes are displayed in **Section 4.4 (page 67)**.

3.*3.2.3 Pseudocode*

The following pseudocode for the note generation algorithm is based on the example given above. This is an outline of some of the variables and methods within the Clip class, which the note generation algorithm resides in; I have not outlined all of the methods, because they don't all relate to the note generation algorithm.

The Clip class also needs to access the variables within the arrangement that it is used in, hence why I have included sections saying Arrrangement.<variable name>.

```
private int: rangeOfOctaves
private int: rangeOfNotes
private int: syncopationRandomisation
private list (of type Note): notes
private static int[] accidentals = {4, 1, 5, 2, 6, 3, 7}
private static

public method Constructor {No parameters}
```

All properties are private to prevent from unauthorised access

To instantiate a Clip object, we need a constructor. As all parameters are handled by other functions, we don't need parameters in the constructor.

This method acts as an access modifier; it gets properties given from the use of the function in the Program class and sets the private variables

```
public method SetDetails {Parameters = (int: RangeOfOctaves, int: RangeOfNotes, int: SyncopationRandomisation)}:
        rangeOfOctaves ← RangeOfOctaves
        rangeOfNotes ← rangeOfNotes
        syncopationRandomisation ← SyncopationRandomisation

public method CreateClip {No parameters}:
        int[]: octaveRangeAvailable ← {4 - rangeOfOctaves, 4 + rangeOfOctaves}
        int: highestNoteAvailable
        if Arrangement.tonicNoteNumerical = 1
                highestNoteAvailable ← 7
        else:
                highestNoteAvailable ← Arrangement.tonicNoteNumerical - 1
        int: numberAccidentals ← Number of accidentals in key signature
        bool: sharpen
        int: previousNoteValue
        static float[]: fractionNoteValues ← [1, 3/4, 1/2, 1/4, 1/8, 1/16]
        int[]: accidentalsToApply[numberAccidentals]
        if Arrangement.Sharps() = true:
                sharpen ← true
                for (int i ← 0 to (numberAccidentals – 1)):
                        accidentalsToApply[i] ← accidentals[i]
        else (key contains flats):
                sharpen ← false
                for (int i ← 6 (decreasing) to (7 – numberAccidentals)):
                        accidentalsToApply[i] ← accidentals[i]
        float: spaceLeft ←  Arrangement.timeSignature * 1/4
        infinite loop:
                int: pitch ← Random number between 1 and 7
                int: octave ← Random number between 1 and 10
                int: noteValue
                if syncopationRandomiser(syncopationRandomisation) = true:
                        noteValue ← Random number between 0 and 5
                else:
                        noteValue ← previousNoteValue
                if spaceLeft >= fractionNoteValues[noteValue]:
                        if (octave >= octaveRangeAvailable[0]) & (octave <= octaveRangeAvailable[1]):
                                if (pitch >= Arrangement.tonicNoteNumerical) & (pitch <= highestNoteAvailable):
                                        if accidentalToApply contains pitch:
                                                if sharpen = true:
```

If the range of octaves chosen is 2, the lowest octave permitted for notes is 2 (because it is 2 below octave 4) and the highest is 6

As there are only 7 notes available in 1 octave, if the key note is 1 (C), then the highest note available is 7 (B). Otherwise, it is whatever note is below the key note.

A function determined in the Arrangement class (which I am not sure how to implement at this stage) will return to this class the number of sharps/flats in the key signature of the arrangement, which is stored in this variable.

This section stores in an array the note values which need to be sharpened/flattened. The Boolean variable 'sharpen' determines whether the notes are sharpened or flattened. If the key signature needs sharps, then accidentals are grabbed from the 'accidentals' array at the start in incrementing order. If the key signature needs flats, the accidentals are grabbed in reverse order, since that is how key signatures are assigned in music.

Stores the amount of space in note values available in the clip, starting with all free space

If the syncopation randomiser function decides that the note value for the note needs changing, then a random note value out of those selected is chosen

```
                                        pitch ← pitch + 10
                                    else:
                                        pitch ← pitch + 20
                                previousNoteValue ← noteValue
                                notes ← New note
                                New note .SetDetails(pitch, noteValue)
                                spaceLeft ← spaceLeft – fractionNoteValues[noteValue]
                            else:
                                break
                        else:
                            break
                else:
                    break

private bool function syncopationRandomiser {int: syncopationRandomisation}:

    int: randomNumber ← Random integer from 1 to syncopationRandomisation:

        if randomNumber = 1:
            return true
        else:
            return false
```

This function will generate a random number between 1 and the set syncopation randomisation value. If the number is 1, then the syncopation in the note generation will be changed, as the function will return true. Otherwise, it will return false

3.4 Class Definitions

class NewAccount

- private
  - string username
  - string password
  - string confirmPassword
- public
  - method SetDetails(string Username, string Password, string ConfirmPassword)
  - method SaveDetails()

class Arrangement

- protected
  - list<Clip> clips
  - bool saved
  - string name
  - int tonicNoteNumerical
  - bool majorOrMinor
  - int timeSignature
- public
  - method SetDetails(string TonicNote, bool MajorOrMinor, int TimeSignature)
  - method CreateArrangement()
  - method LoadArrangement(int id)
  - method SaveArrangement()
  - bool function Sharps()

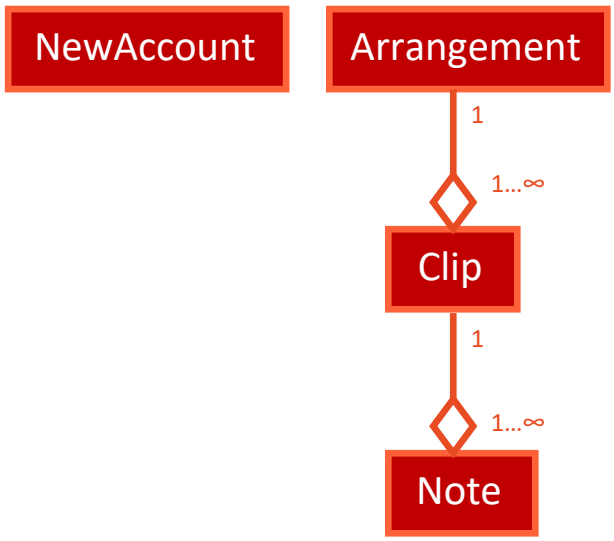
class Clip

- private
  - int rangeOfOctaves
  - int rangeOfNotes
  - int syncopationRandomisation
  - List<Note> notes
  - static int[] accidentals
  - bool function SyncopationRandomiser()
- public
  - method SetDetails(int RangeOfOctaves, int RangeOfNotes, int SyncopationRandomisation)
  - method CreateClip()
  - method LoadClip(int id)
  - method DisplayClip()
  - method SaveClip()

class Note

- protected
  - bool isRest
  - int pitchNumerical
  - int noteValue
- public
  - method SetDetails(bool IsRest, string Pitch, int NoteValue)
  - string function GetStringNoteValue()

3.5 Class Diagram



3.6 Class Behaviours

NewAcccount is used for when the user is entering details for a new record in the Accounts table in the database. The SetDetails method sets the values of the private username and password variables, before the SaveDetails method inserts the data into the database using SQL.

Arrangement contains all methods and variables concerning arrangements. One behaviour it can do is to create a new arrangement, where the user can set the details using the SetDetails method and an insert SQL statement is run in the CreateArrangement method. Another behaviour is to load an arrangement from the database, using a select statement and storing the data into the private variables. The third behaviour is to save an arrangement being worked on and all of the current clips stored within it, by running an update statement. *Arrangement* aggregates a list of *Clip* objects to store the clips.

Clip's main behaviour is to generate a clip of new notes using the properties given by the user, using the SetDetails and CreateClip methods. In doing so, a list of *Note* objects is aggregated. *Clip* can also load a clip from the database using a select statement and the LoadClip method, and save a clip to the database, using either an insert statement, if the clip is new, or an update statement, if the clip has already been saved, in the SaveClip method. Another behaviour of *Clip* is displaying the clip's contents to the screen, using the DisplayClip method.

Note just contains properties about a single note, using a SetDetails method to set the variables. The properties are protected, so that they can be called from the DisplayClip method in the *Clip* class.

4 – User Interface

4.1 Introduction

This section will be outlining how the user interacts with the program, how the inputs from the user and the outputs from the program will be formatted, and an analysis on why the designs are set out like they are.

Although I have been arranging the data in boxes here, the program will actually be arranged in a console window.

This section forms part of the high-level design, as processes are not described here.

4.2 UI Rationale

Although, fundamentally, this program will be a console application, I do not want to implement wholly an old-fashioned command line interface like a DOS or UNIX shell. My client has stated that they would like the program to be simple and user-friendly, and the command line systems, where all you were able to do was type in commands (which you had to know) have always been very unfriendly to use, hence why graphical user interfaces (GUIs) like Windows and Mac OS were developed: they're for the user who's not a geek.

While I will include some text-based entry, as that is the simplest form of input, I will include some elements that take inputs from the user's letter and number key input, to allow menu option choosing to be fast, as well as arrow key inputs to navigate blocks on the screen, to attempt to simulate a GUI.

At the bottom of most screens, I will include a dynamic text box showing the user messages such as which user is logged in, any error messages and to show if anything's loading; I will discuss this bit more later. This is similar to how many modern websites include a banner at the top of their pages to show information about the current user. Showing this on most of the screens keeps a form of consistency throughout the system.

Some areas of the program will also be colour-coded to make different sections easier to distinguish. My client also asked for sections to be coloured. Although this is easier to accomplish in a Windows form application, it will make certain items look tacky, as many form controls are not designed to be multi-coloured. Hence, colour would look more suitable in a console application.

4.3 Planned Data Capture Designs

In the diagrams below, all red areas show spaces showing where and how data can be input by a user. These may be either read lines or prompts to press a key (any red text is text that will be displayed).

All of the orange areas show brief descriptions of what any boxes or areas on the diagrams represent; a more detailed description is given below it.

A green arrow shows what screen the input section leads to when filled.

Any black text is text that will definitely be displayed.

You'll see the name Project Gracenote in these designs. This is a codename for the project and I have not yet decided on a final name.

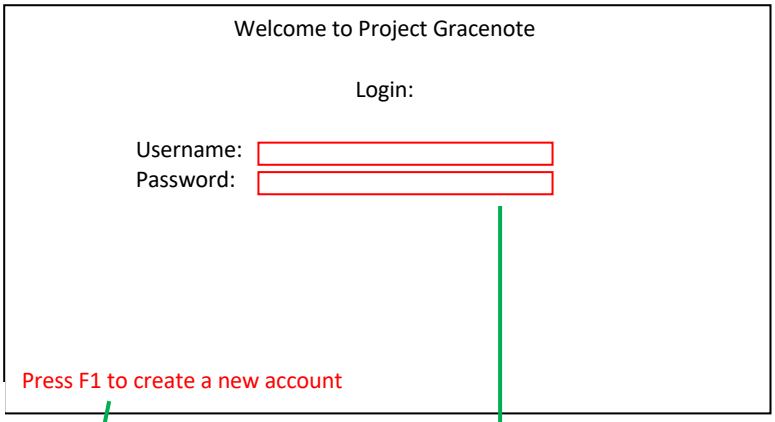


Figure 3.1 *– login screen*

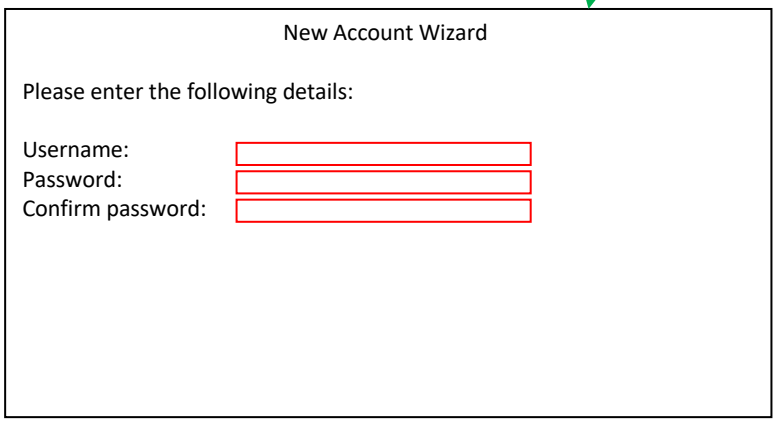


Figure 3.2 *– new account wizard*

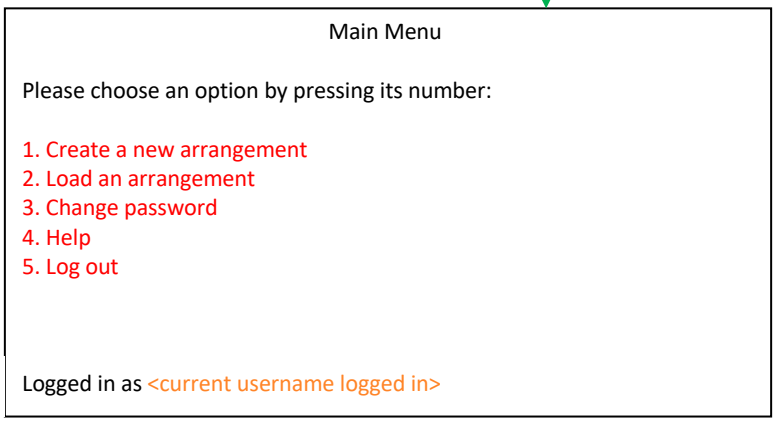


Figure 3.3 *– main menu*

As in **Figure 3.1**, the first input that the user will be prompted will be for their username. They must then press enter before they can enter their password. They then press enter to log into the program and get to the main menu.

If they wish to create a new account, the user can press F1, which will lead them to the new account wizard in **Figure 3.2**.

They must press enter after entering each detail, after which they will be returned to the login screen to log in with the new details. Once the user is logged in, they are displayed the main menu.

In **Figure 3.3**, the user presses the corresponding number on their keyboard to the option they'd like. If any invalid keys are pressed, nothing happens.

Figure 3.3

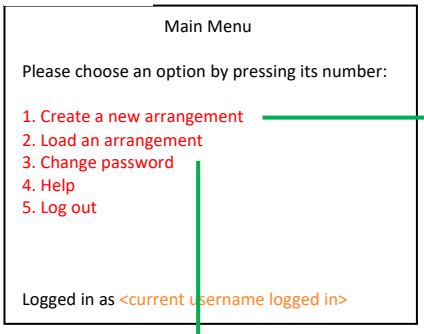


Figure 3.4 *– new arrangement screen*

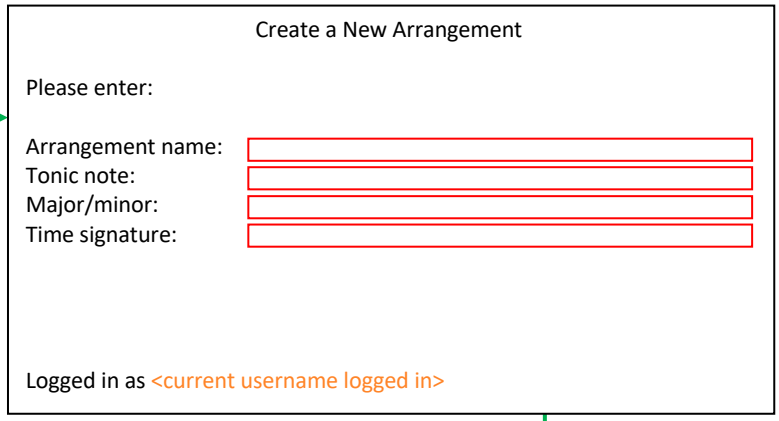
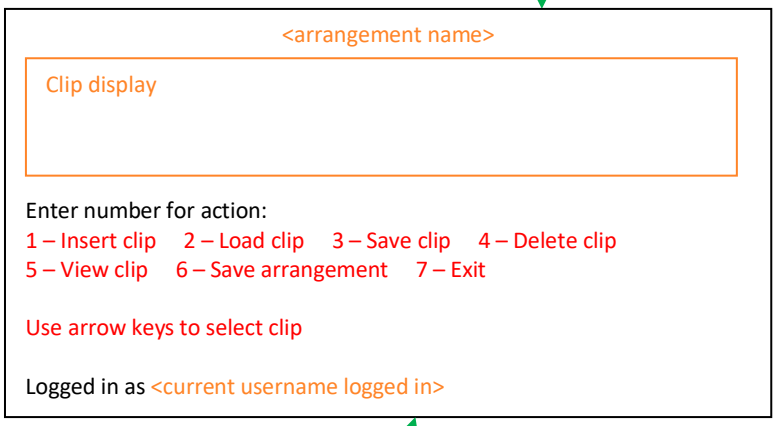


Figure 3.5 *– arranger*



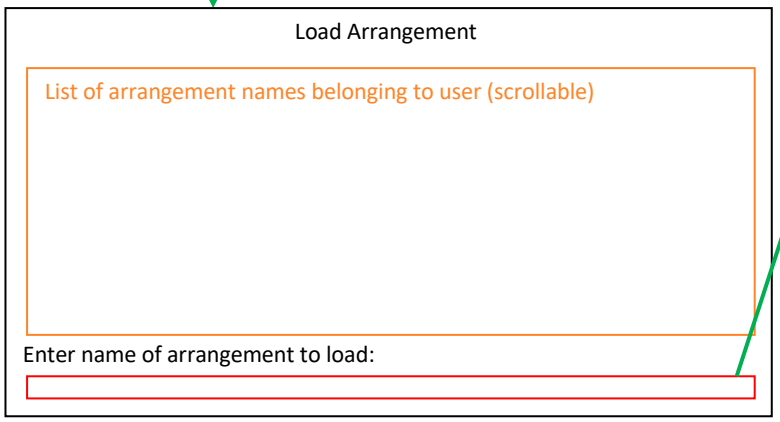


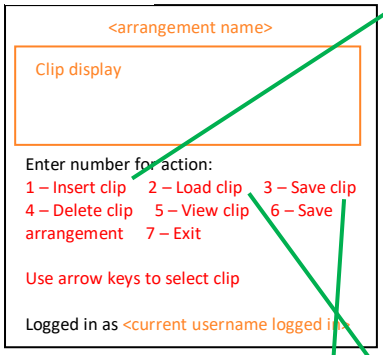
Figure 3.6 *- arrangement loader*

If the user chooses option 1, they are shown the new arrangement screen (**Figure 3.4**), which lets the user enter the details for a new arrangement. They must press Enter after filling in each detail to get the next one. Once they finish and they press Enter, a new record is created in the arrangements table in the database using the name that the user has entered as the name for the record. The user is then displayed with the arranger (**Figure 3.5**) with no clips in the clip display (the design of the clip output is in Section 4.4 on page 67).

If the user chooses option 2 and loads the arrangement loader (**Figure 3.6**), once they have entered the name of the arrangement to load, they are lead to the arranger, but this time, the data for this arrangement is retrieved from the arrangements table in the database and the clips are displayed in the clip display.

From the arranger, the user can select a clip using the arrow keys and it will be highlighted. On each clip, or empty space if a bar is empty (the clip display design is in Section 4.4), the user can then press a number to perform an action. These actions are displayed below:

Figure 3.5



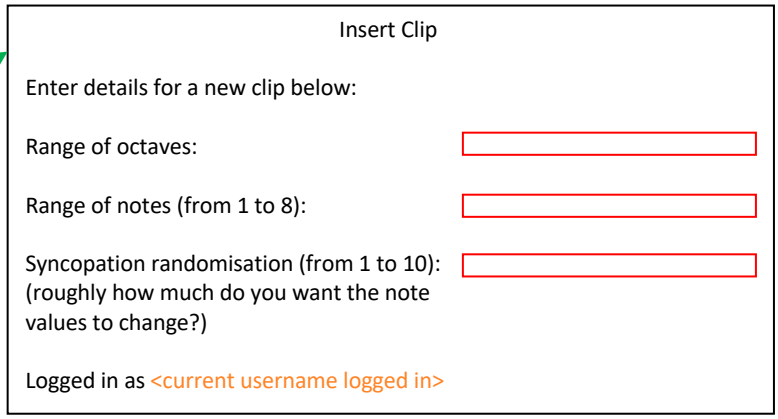


Figure 3.7 *– new clip wizard*

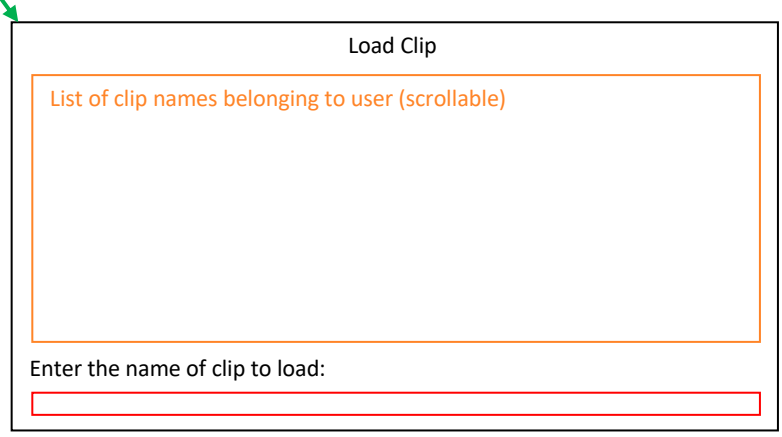


Figure 3.8 *– load clip wizard*

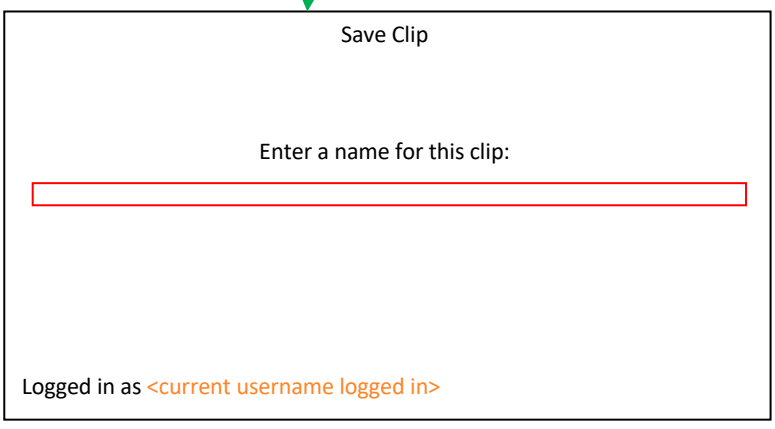


Figure 3.9 *– save clip wizard*

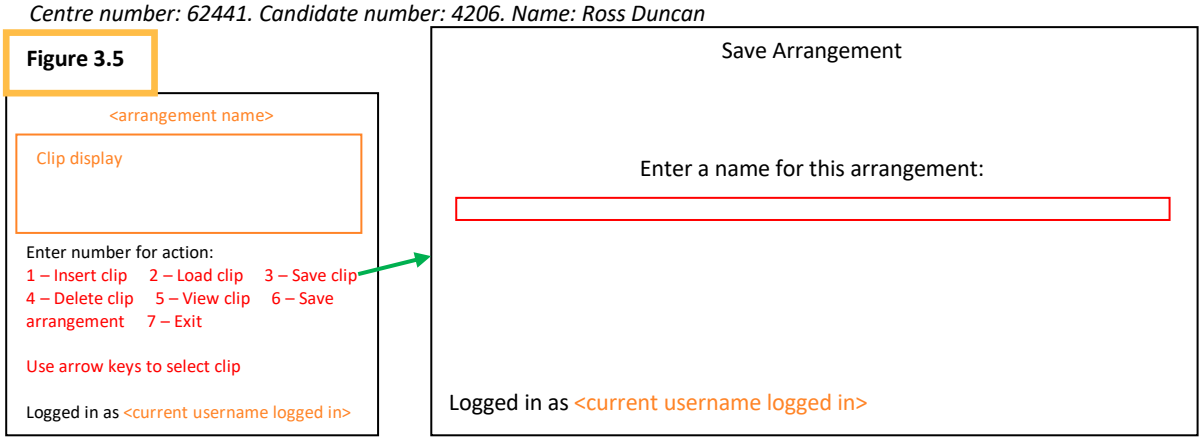


Figure 3.10 *– arrangement save wizard*

In **Figure 3.7**, the user must press Enter after entering each detail before moving on to the next one, before a new clip is inserted. In **Figure 3.9**, the user only needs to enter the one input before pressing Enter.

Figure 3.8 will be scrollable since a Windows console window can scroll if there is too much text to be displayed in a single window. This will be useful if the user has a large number of the clips saved. The same dynamic applies to **Figure 3.6**. As an Easter egg, I will allow the user to delete a clip rather than load one by pressing the F1 key. This is will change the title of the screen to Delete Clip, and the clip of the ID chosen by the user will be deleted from the database.

Figure 3.9 is only displayed if the clip selected by the user hasn't already been saved and given a name. If the user presses 6 if they have saved it before and the clip has a name, then the clip's record in the database is updated with no form to display.

In **Figure 3.5**, if the user presses 4 to delete a clip or 5 to view a clip, no input form is displayed. The screens displayed are in Section 4.4.

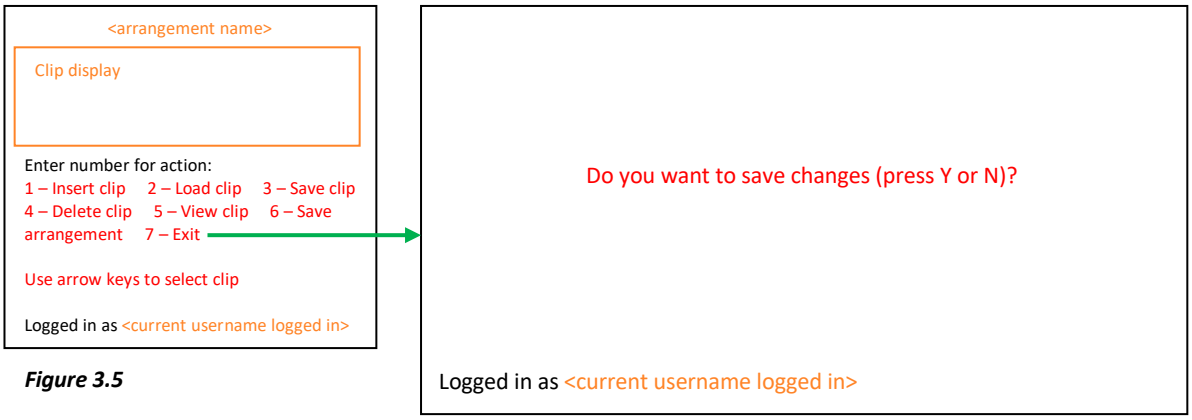


Figure 3.5

Figure 3.11 *– save changes warning*

Figure 3.11 is only displayed if the user has chosen to exit by pressing 7 and they haven't saved it yet.

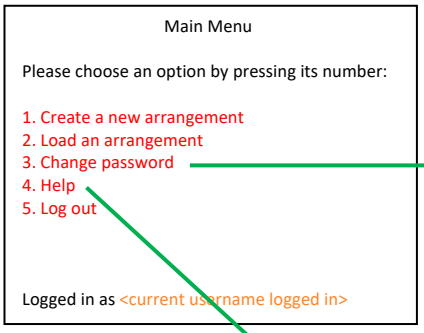


Figure 3.3

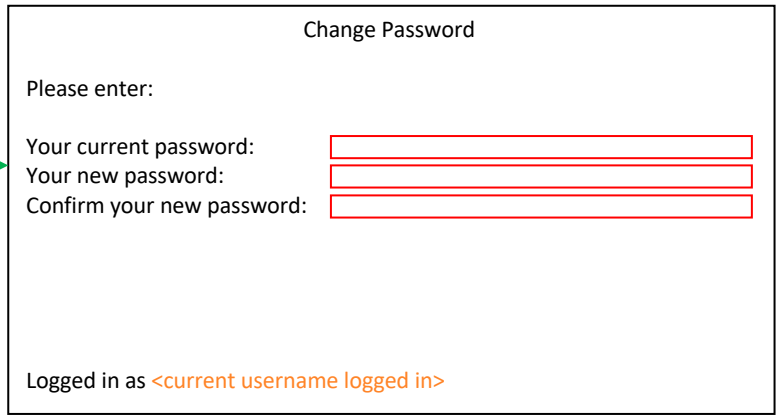


Figure 3.12 *– change password wizard*

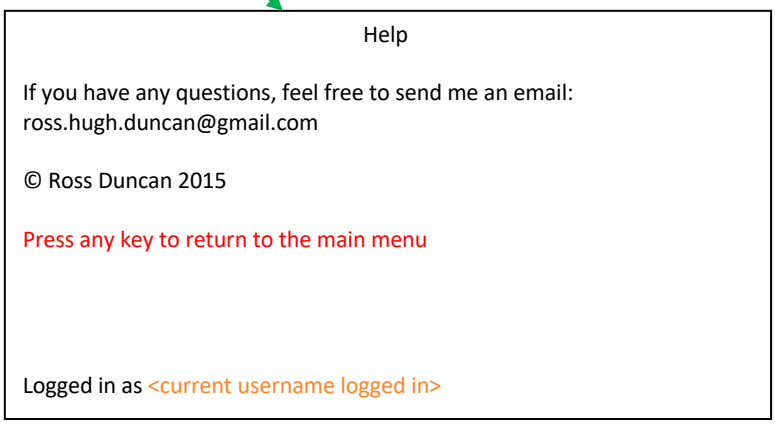


Figure 3.13 *– help screen*

Once the user has entered the details required and pressed enter in **Figure 3.12**, or pressed any key in **Figure 3.13**, they are shown the main menu.

In **Figure 3.12**, if there are no validation problems, the password is updated by running an update statement on the Accounts table in the database.

In **Figure 3.3**, if the user chooses 5 to log out, they are shown the first login screen.

4.4 Planned Data Output Designs

In any of the above screens where there is text at the bottom showing the username logged in, this will change to display 'Loading…' if the program is loading the next screen. It will change back to showing the username when finished loading. Additionally, if any error messages need to be displayed to the user, they are shown in this string of text as well; this text will be printed in yellow, while all other text in the console will be the default white, although that may change.

In **Figure 3.5**, the clips display section will show graphics of clips. Here is an example, where I have included three saved clips (called Clip1, Clip2 and Clip3), a space with no clip, and an unsaved clip. In this case, the first clip is selected.

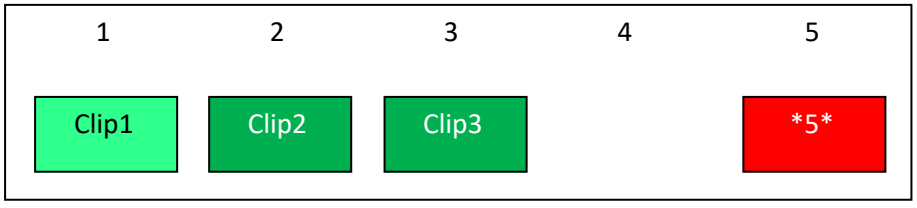


Figure 4.1

The numbers above the clip boxes represent the bar that the clip belongs to. All unsaved clips will be red with their bar number in asterisks as the text, and all saved clips will be green and contain the text that the user has given to them. The selected clip is displayed with a brighter colour than the other clips and black text, whereas all unselected clips will have white text.

If the user was to press their right arrow key to select the next clip along (Clip2), this is what would show:

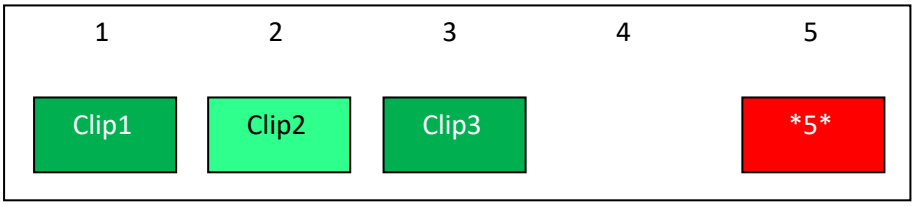


Figure 4.2

If the user was to select the blank space in bar 4, it would look like this:

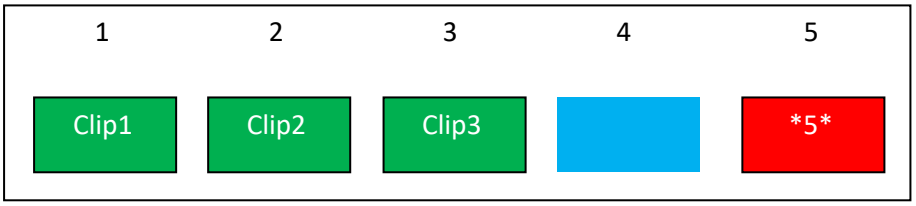


Figure 4.3

All selected blank spaces will be shown in light blue.

If the user selects the unsaved clip 5, this is shown:

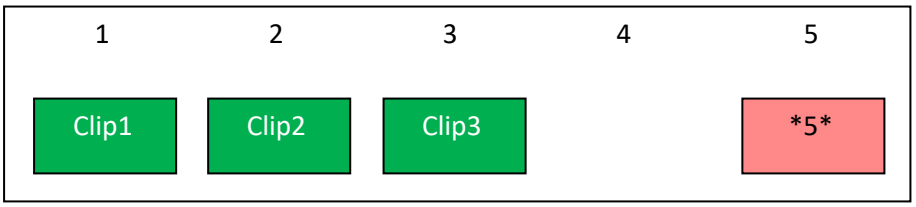


Figure 4.4

If the user chooses to delete clip 5, this shows:

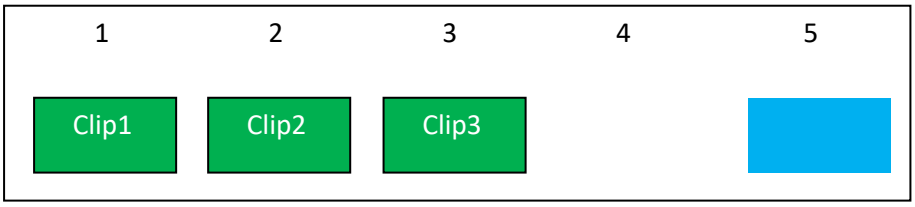


Figure 4.5

Bar 5 is still selected, but as there is no clip there anymore, it is a blank space, hence the blue box.

If, in **Figure 3.5**, the user presses 5 to view a clip, this is an example of what would be shown (if the clip's name was RandomClip):

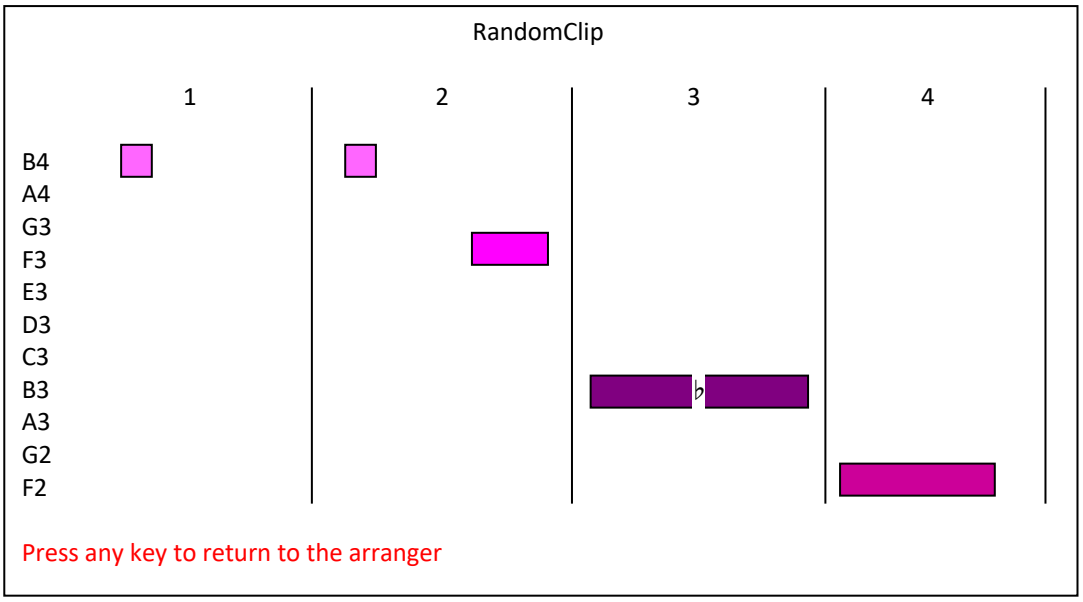


***Figure 4.6** – clip viewer*

The numbers above the notes represent the beats that they belong to. The lines separate the beats.

The display may contain any range of notes depending on what the algorithm has generated. If so, the display will be scrollable thanks to the console window. There may also be any number of beats depending on what time signature the user has selected, so it will be scrollable horizontally as well.

If any notes are sharp or flat, they will have a ♯ or ♭ symbol contained in the note block, as demonstrated in **Figure 4.6**.

Different note values will all be in different shades of pink and purple. The smaller note values will be the palest and the longest the darkest; the colour hex codes will be set in an array for each note value.

For this project, I will only support notes from a whole bar (a semi-breve) to a 1/16th of a bar (a semi-quaver). I will also for now only support time signature which are simple time, e.g. 4/4, 3/4 or 2/4. It is also possible to get compound time, whose beats are divided in 8th notes (quavers), but that is far more complicated to implement than simple time, and I may not have time in this project.

This screen can be used for the user to take ideas and perhaps play the notes on a music keyboard. It is up to them to then adapt the notes to their musical ideas, which is the purpose of this software.

5 System Security

5.1 Introduction

In this section, I will be identifying the appropriate security measures to put in place for the protection of the data used in the program and any functions or variables used in the code.

5.2 Data Security and Integrity

5.2.1 Password Hashing

The password must be stored in the Accounts table as a hashed value; however, there are several algorithms that we can choose from.

The MD5 algorithm generates a 128-bit hex digest that aims to be unique for each message. The system is suitable for password storing and validation, since it will always generate a fixed-length hash. However, it is not collision-resistant – in 2004, scientists Wang, Feng, Lai and Yu were able to create identical hashes from two messages of different hex data within an hour. Regarding computational efficiency, we will also probably not require compression for a password, as it is unlikely that a password would be the same size as a full document.

The RIPEMD algorithm can produce digests of multiple sizes, namely 128, 160, 256 or 320 bits. It is based on MD4 and, despite its ability to encrypt messages of any size, it is not very widely used – there is very little information online and no freely-available specification; the only useful information I could find about it was on forums. Compared to its more successful cousin, SHA, it takes more steps to compute.

SHA is a U.S. Federal Information Processing Standard (FIPS) algorithm. The latest standard, as of writing this, is SHA-3, published in August 2015.

In Visual C# 2012, which is what I am using to code, the latest and fastest version is SHA-256, which is a variation of SHA-2. This version of the system, unlike MD5, is almost impossible to perform a collision attack on; there are over 1,300 implementations of SHA-256 online validated by FIPS.

It would be infeasible to use the hash on other fields in the database, as a hash is designed to not be reversed, and the other fields, such as the arrangement data, need to be written to and read from.

For the simplicity of this project, the password is the only field that I will encrypt; other fields could be encrypted by using a cipher, although I will not implement that here.

5.2.2 Clip and Arrangement Encoding

As the data about the arrangements and clips are to be stored as text fields in the database, the data needs to be represented by certain characters, which can saved and extracted. This system can double as making the data more secure, because the data should not make sense to anyone hacking the database.

In order to make the data secure, I must use opcodes to represent what the data contains, rather than characters that can be easily guessed by a hacker. Only I would be able to work out what the opcodes mean because I created the key, which I keep private.

5.2.2.1 Encoding Clip Data

In the Clips table in the database, there are 2 fields regarding the clip data: *ClipName* and *NoteData*.

The *ClipName* field does not need to be encoded, as it is one piece of data.

5.2.2.1.1 Opcodes

The *NoteData* field contains all of the notes within one clip.

Each opcode representing the different pieces of data is made up of two digits, where each first digit has a set of subdigits. The opcodes in red are followed by operands.

First Digit	Second Digit	Meaning
0		Start of
	0	a note
	1	a rest
1		End of
	0	a note
	1	a rest
2		Note/rest data
	0	Note/rest value
	1	Pitch (only for notes)
	2	Octave (only for notes)

See **Section 3.3.2 (page 55)** for what the note/rest value codes are.

Pitches are represented using 2 numbers, as explained in **Section 3.3 (page 53)**. To recap, sharp notes start with a 1 and flat notes start with a 2. For the purpose of this encoding, any natural (non-sharpened or non-flattened) notes will start with a 0. The first digit is followed by the main numerical pitch value (C being 1, up to B, being 7).

5.2.2.1.2 Working Example

Below is an example of note data encoded into opcodes in a record in the database. All of the opcodes are in **bold green** and any data is in light blue. The data that is contained is the following:

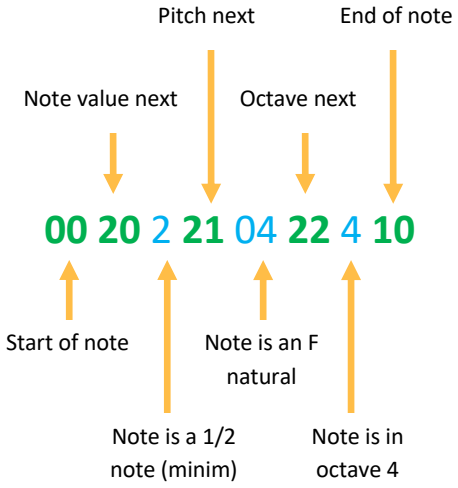
- A minim (1/2 note) with pitch F4
- A minim rest (1/2 rest)
- A crotchet (1/4 note) with pitch D♯2

This is the encoded note data (I have spaced the different sections out so it's easier to read):

00 20 2 **21** 04 **22** 4 **10 01 20** 2 **11 00 20** 3 **21** 12 **22** 2 **10**

First note Rest Second note

In case this doesn't make any sense, here's the first note's encoding closer up, showing what each section means:



In the database, the data would be stored like this:

00202210422410012021100203211222210

This should more than likely not make any sense to someone who doesn't know my keys.

5.2.2.2 Encoding Arrangement Data

Encoding for the arrangement data acts much the same way as with the clip data, but there is different data to be stored.

There are three fields in the arrangement table that refer to arrangement data: *ArrangementName*, *ArrangementData* and *ClipContentsData*. *ArrangementName* does not need to be encoded as it has one datum; the other two do.

5.2.2.2.1 Opcodes

The *ArrangementData* field contains general data about the arrangement which are useful for the creation of clips. These are the opcodes that it can contain:

Opcode	Meaning
0	Tonic note
1	Major/minor
2	Time signature
A	Start of a clip that has been saved
B	Start of a clip that hasn't been saved

I am going to use the letter C to represent major and D to represent minor.

A is followed by the compound primary key of the use of the clip in the ClipUse table, which can then be looked up, and the ID of the saved clip can be found, so the clip data can be loaded.

The reason that the last 2 opcodes are letters is so that they do not get mixed up with any compound primary key data, which is all numbers.

B is followed by an ID of an unsaved clip within the *ClipContentsData* field. This ID must start with a E and end with a F, to reserve a space for the ID.

The reason I am including the letters A to F is to try and trick a hacker into thinking that the code is a hex code, which includes numbers and the letters A to F.

5.2.2.2.2 Worked Example

Below would be the *ArrangementData* field for an arrangement which has the following details:

- The key is A major
- The time signature is 4/4

The arrangement includes a use of a saved clip which has a compound primary key in the ClipUse table of 1010, and an unsaved clip, whose ID is E1F, and another unsaved clip, whose ID is E2F.

0 06 **1** C **2** 4 **A** 1010 **B** E1F E2F

In the database, it would look like this:

0061C24A1010BE1FE2F

In the *ClipContentsData* field, this would be in it:

E1F <data> **E2F** <data>

In the <data> regions would be the same type of encoding as in **Section 5.2.2.1 (page 70)**, where the clips' notes and rests are represented.

5.3 System Security and Integrity

Part of ensuring that no data can be accessed without permission is by making the user log in, which we have already seen.

To ensure that the variables within the system are secure, most have been declared as private, so that they can only be used within their containing class.

Although many applications provide an admin option to allow an admin user to access more features than other users, in my program, there aren't enough features to warrant access permissions. All users will get to use all of the features. However, the current user's account ID is stored as a variable within the Program class, and this means that any of the SQL queries cannot look up any data that doesn't belong to that user's account.

When loading lists of arrangements or clips to load, only the ones belonging to that user will be displayed as options.

6 Testing

6.1 Introduction

This section outlines the different series of tests that I will run on the code, which each have a purpose for testing a specific area of the program.

These test series will be expanded in more detail in the test plan section.

6.2 Overall Test Strategy

Series	Condition to Test
1	The user is able to successfully navigate the program using all forms.
2	A new account can be created.
3	The input validation is all carried out correctly.
4	The fields in the database are updated with no errors.
5	Arrangement and clip browsers are displayed correctly.
6	An arrangement can be created, loaded and displayed successfully.
7	Navigation using the keyboard within the arranger works correctly.
8	A clip can be created, loaded, displayed and deleted successfully.
9	The user can change their password.

For all series, I plan to use white-box testing, except for series 8, for which I will use black-box testing. This is because the accuracy of the notes from the note generation algorithm within the CreateClip method is dependent upon the end result rather than how the result was formed; it is more straightforward to create a convincing series of notes using trial and error. In other words, it would not be practical to run full debugging on series 8, but it will be on the other series.

Technical Solution

1 Introduction

This section demonstrates the operation of the program showing its states. Each state that requires data is using test data entered by me, all of which is valid for the program. The insertion of invalid data and the respective errors are demonstrated in the **Testing** section (page 105).

2 Screenshots

What you will see next are a series of screenshots showing the progression of the states. Each screenshot has a figure legend, each of which contains red text, showing how the user moves to the next state.

Each subheading is accompanied by an introduction and the screenshots are shown with an explanation to the processes behind the program.

2.1 New Account

This shows creating a new account, adding it to a new record in the Accounts table in the database, and then logging in.

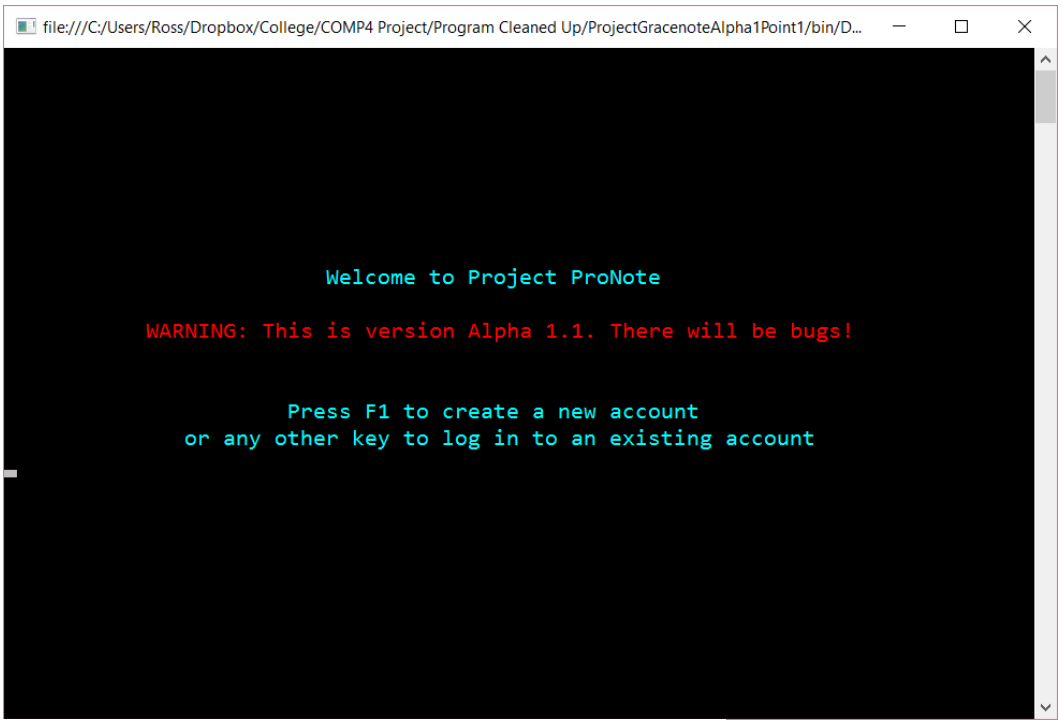


Figure 5.1 *– the welcome screen. Press F1 key*

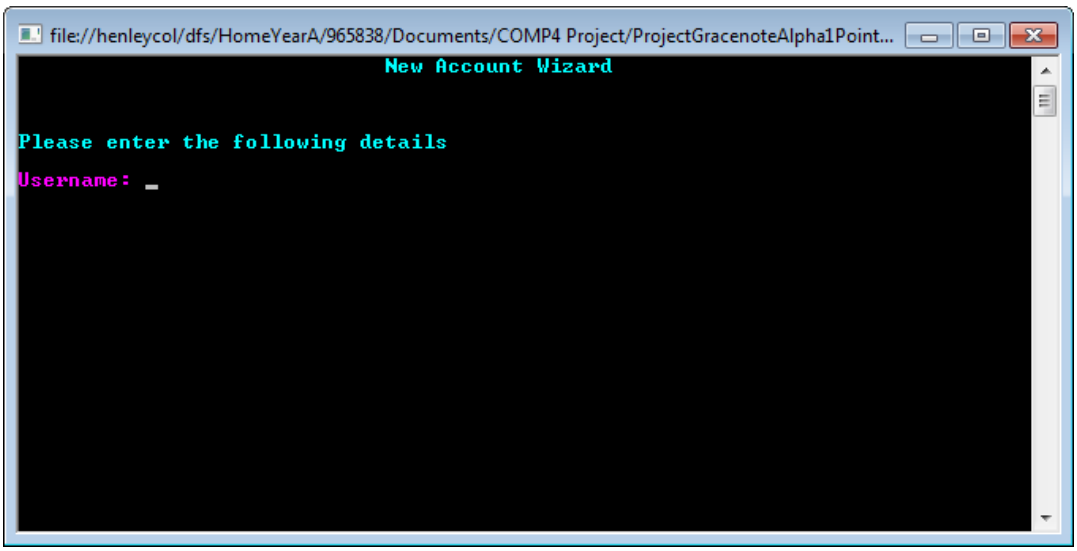


Figure 5.2 *– the new account wizard. Enter details*

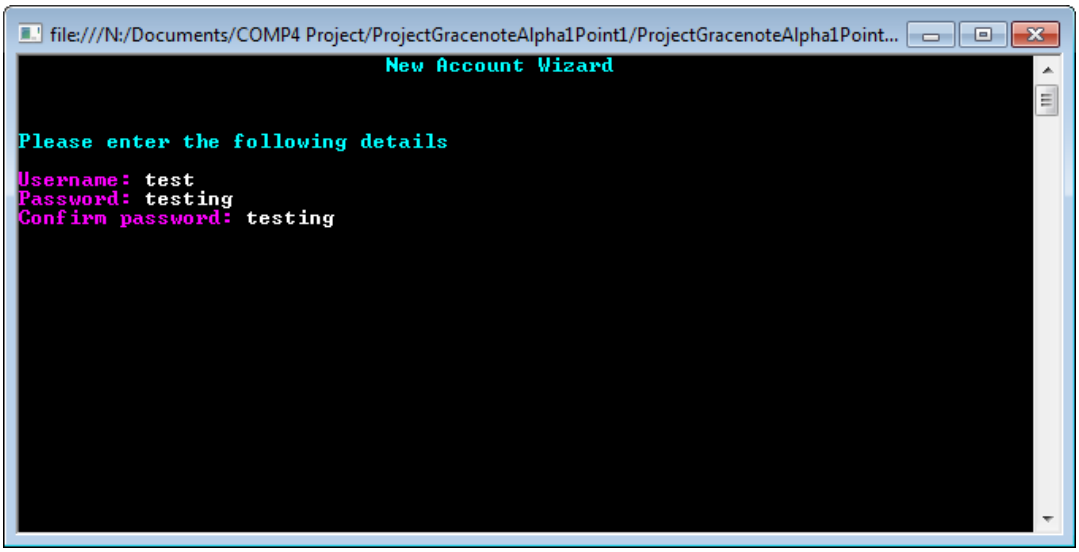


Figure 5.3 *– new account wizard with data. Once all data is entered, press Enter*

Once the user enters their data in **Figure 5.3**, the username and password needs to be sent to the Accounts table. The password is hashed using the SHA-256 algorithm before the insert query is run.

Before the account is created, the Accounts table is empty. After, we have the following data:

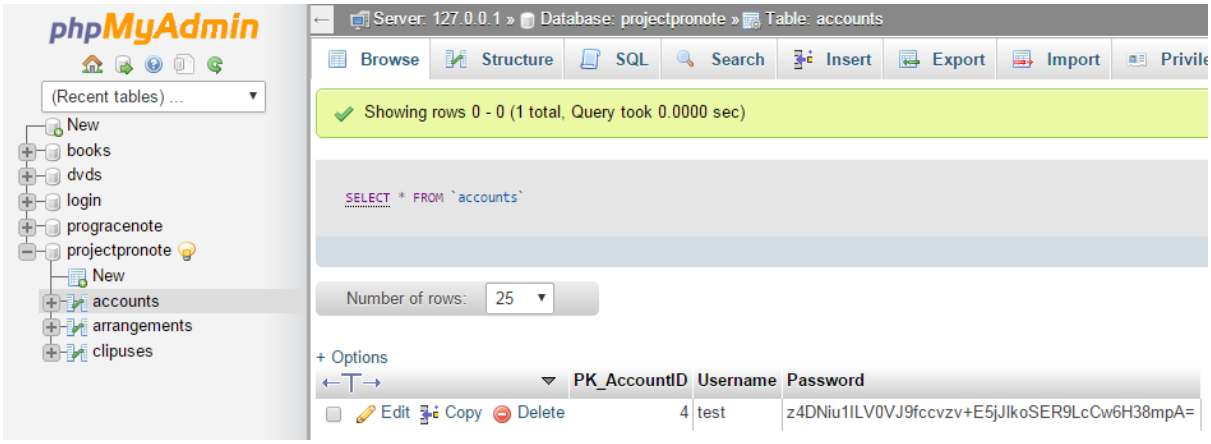


Figure 5.4 *– Accounts table*

The value for PK_AccountID is 4 because I created 3 accounts before this one in the same table during development, and the field is auto-incrementing.

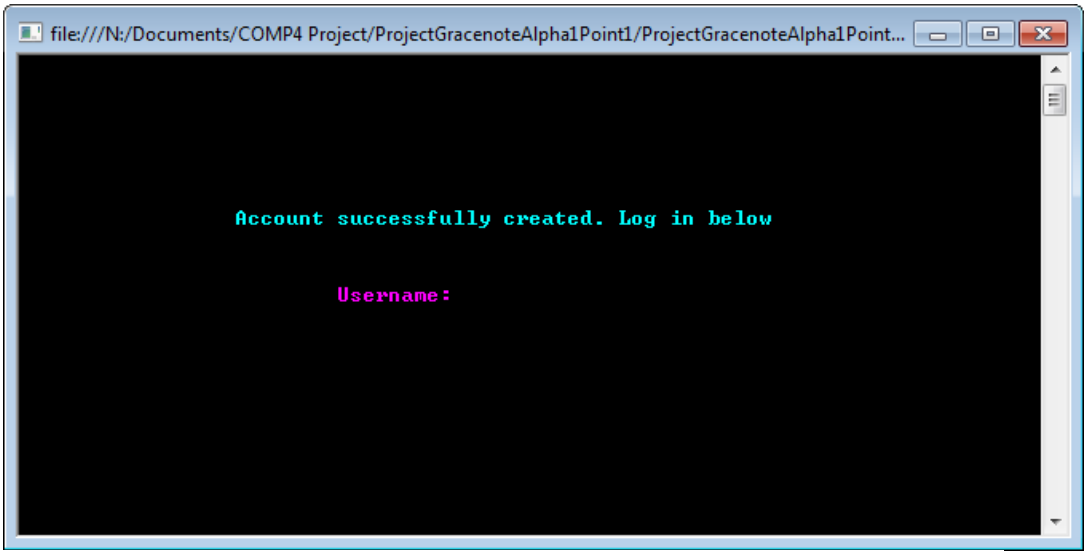


Figure 5.5 *– login screen after account is created. Enter username and password*

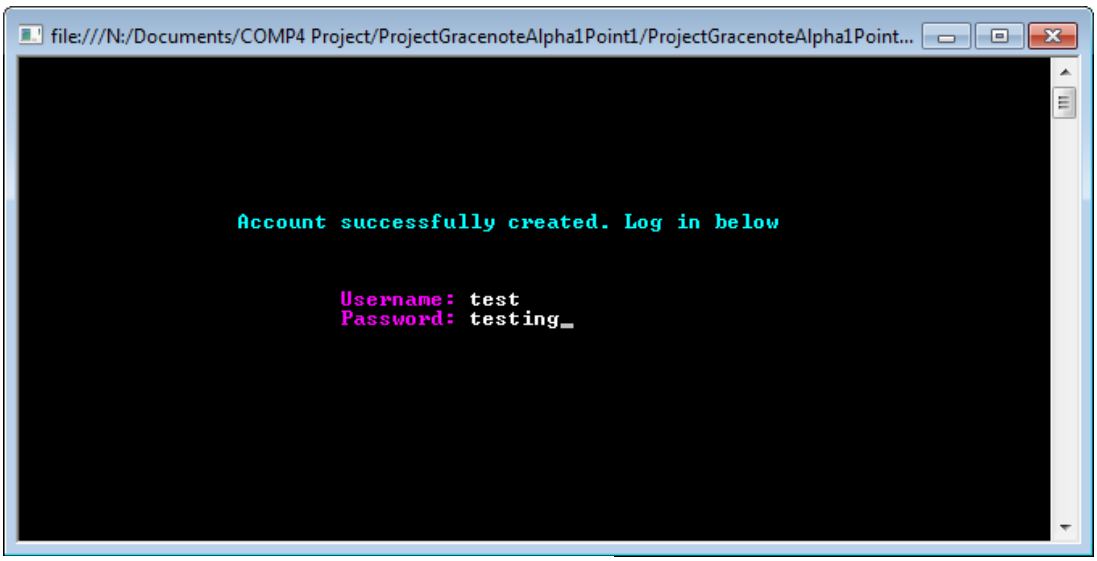


Figure 5.6 *– login screen with data. Press Enter*

When you enter the password on the login section, this is hashed, and is compared with the hashed password in the Accounts table.

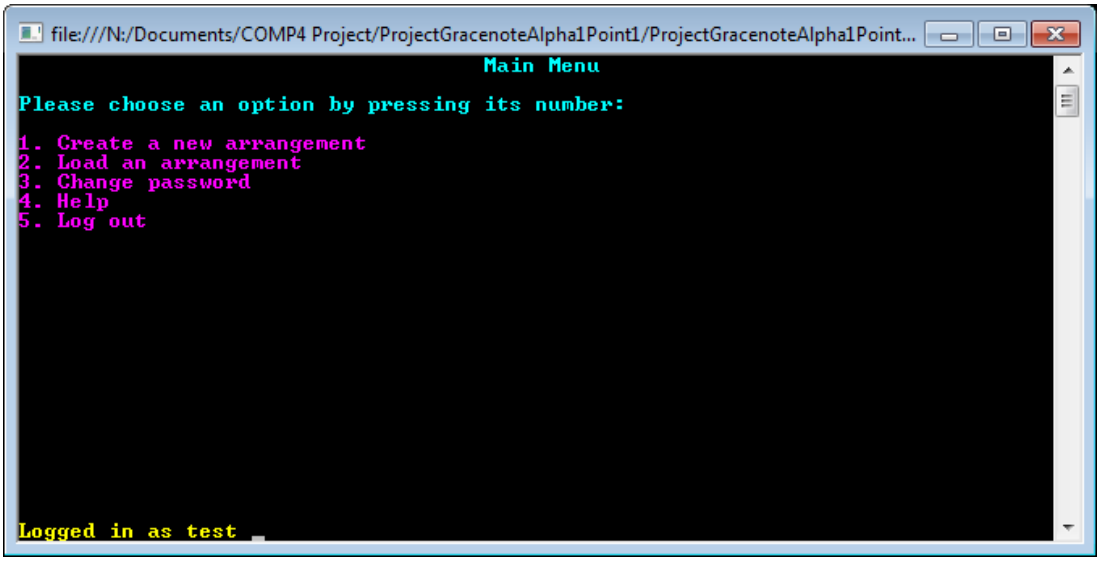


Figure 5.7 *– the main menu*

2.2 Logging into an Existing Account

This section shows how to log into an account that has already been created:

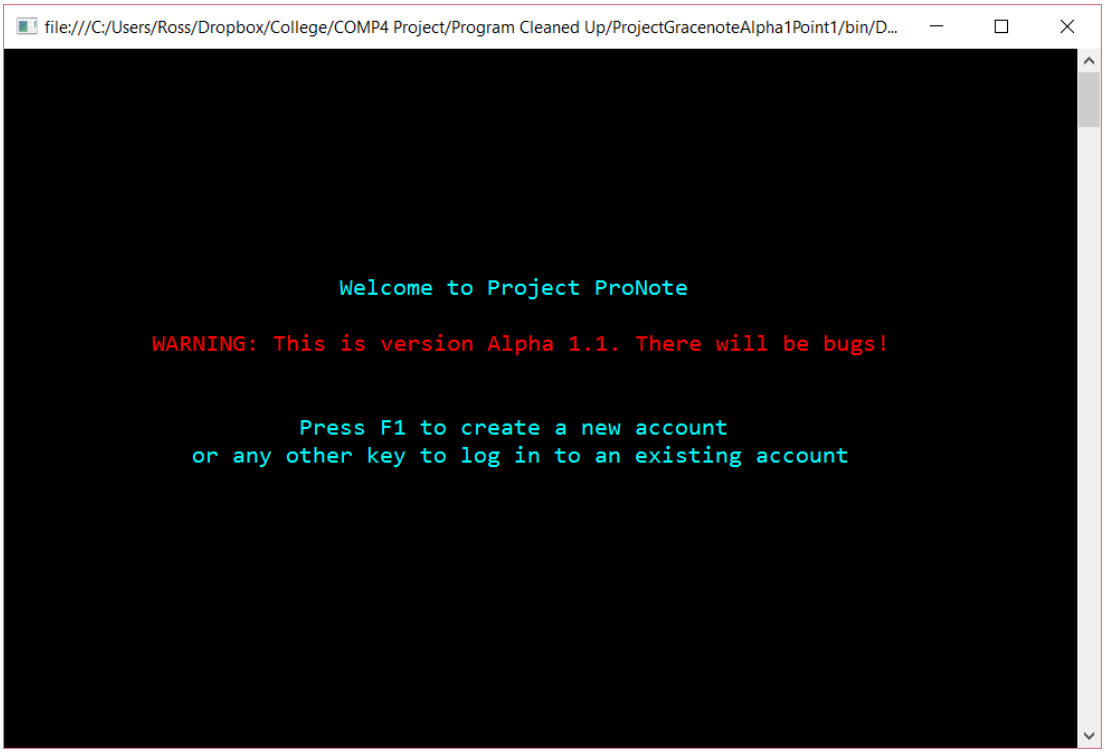


Figure 5.8 *– welcome screen. Press any key other than F1*

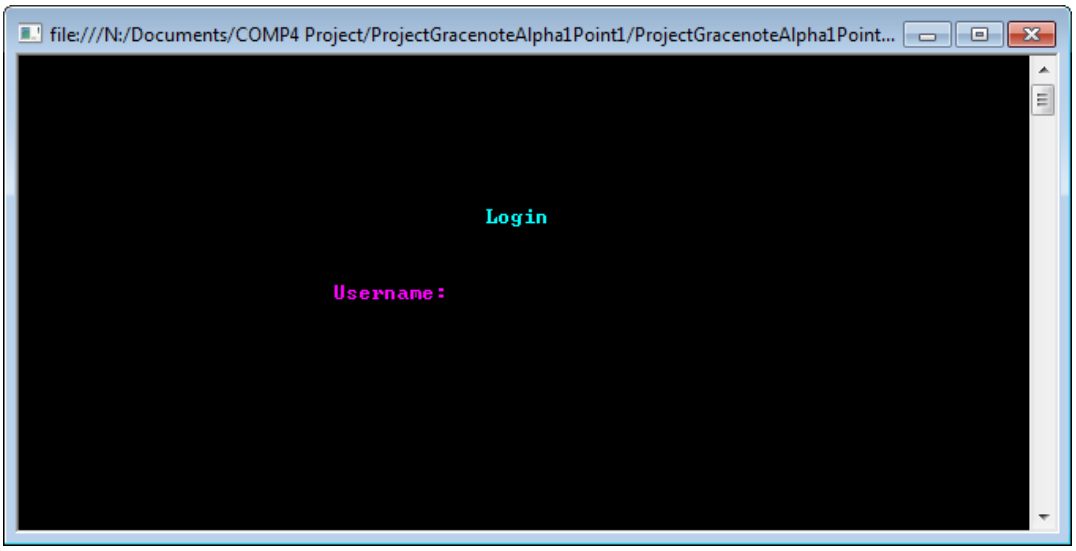


Figure 5.9 *- login screen for existing users. Enter username and password*

Once the user enters data in **Figure 5.9**, the password is hashed before a select statement against the Accounts table is run to check if the username and password are correct. If they are, show the main menu; if not, get the user to enter it again and show an error message (error messages are explored in the **Testing** section on page 105).

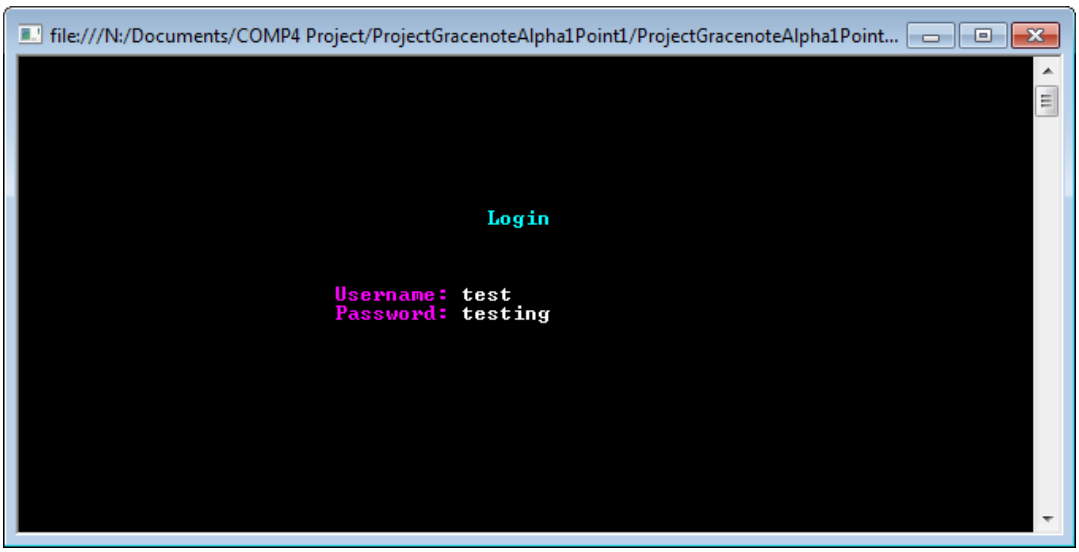


Figure 5.10 *– login screen with data. Press Enter*

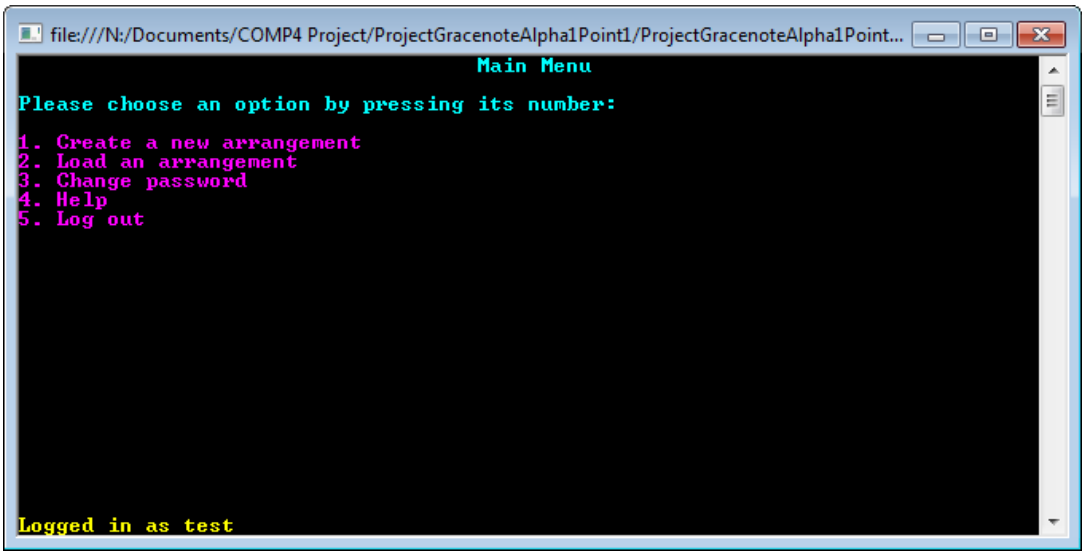


Figure 5.11 *– the main menu*

2.3 Creating a New Arrangement

Once the user is logged in, they can create a new arrangement. As a reminder, an arrangement is space which allows the user to create clips of notes. Each arrangement has a name, and key and time signature properties. Each one is stored as a record in the Arrangements table in the database.

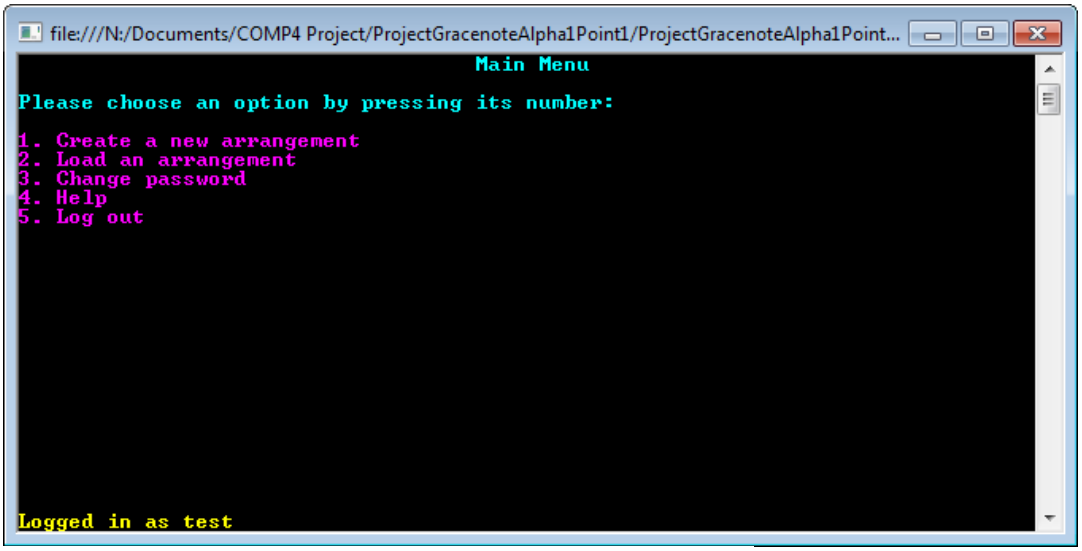


Figure 5.12 *– the main menu. Press 1 key*

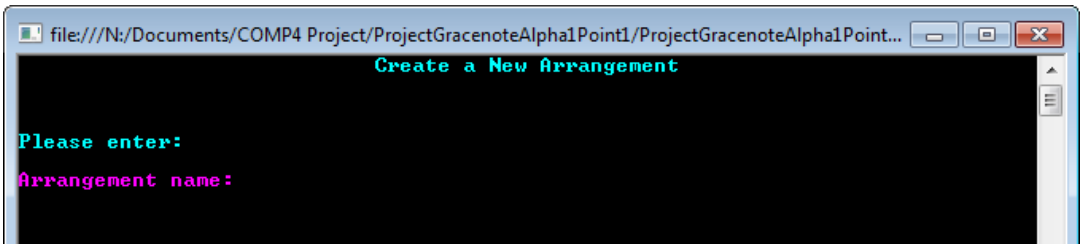


Figure 5.13 *– new arrangement wizard. Enter arrangement details*

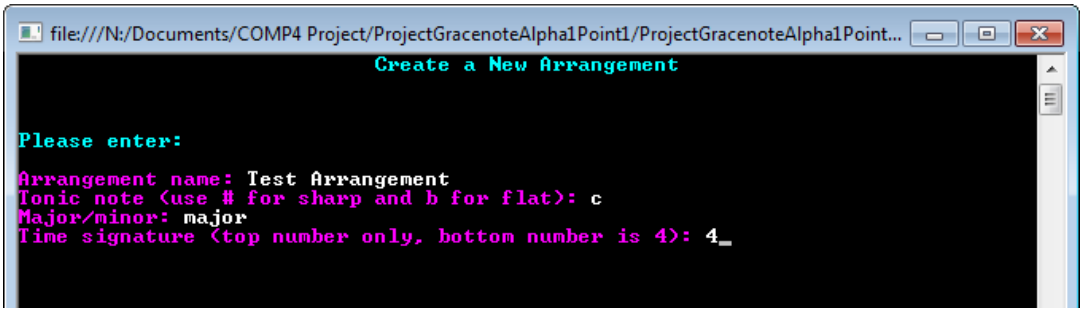


Figure 5.14 *– new arrangement wizard with data. Press Enter*

Once the user presses Enter on **Figure 5.14**, a new arrangement object is created with the following properties set:

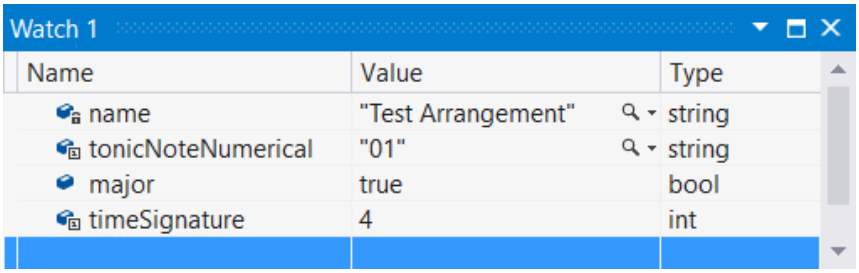


Figure 5.15 *– arrangement object properties*

In **Figure 5.15**, we see some of the private variables for the arrangement object we've just created. tonicNoteNumerical represents the 2-digit code for the tonic note. "01" is the code for C. If it was "02", it would be D, as explained in the **Design Section 3.3.1 (page 53)**. The major variable set to true shows that the key is C major (as oppose to C minor). The timeSignature value of 4 means that we are using a 4/4 time signature. If it was 3, we would be using 3/4.

A new record is also created in the Arrangements table in the database:

PK_ArrangementID	ArrangementName	ArrangementData	ClipContentsData	FK_AccountID
20	Test Arrangement	0011C24	*NULL*	3

Figure 5.16 *– new arrangement data*

We can see that MySQL has created a new arrangement ID. The account ID is 3 because this arrangement belongs to our test account, which is account number 3.

The 0011C24 string represents the properties as shown in **Figure 5.16**. The first 0 means tonic note next, followed by 01, which our C code. The 1 that follows means major/minor next, followed by a C, which means major. If that was a D, it would mean minor. The 2 means time signature next, followed by a 4, which means we're using a 4/4 time signature. So far, this is the only data we have, but more will be added once we insert clips. ClipContentsData is null because we have no unsaved clips yet.

Once this record and object is created, it is displayed to the user:

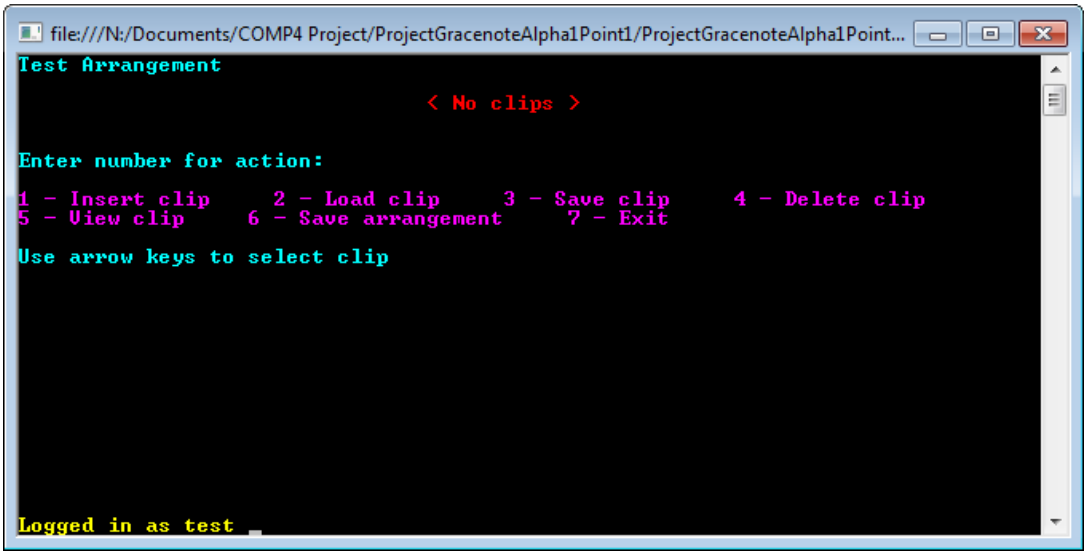


Figure 5.17 *– the arranger for the new arrangement*

2.4 Inserting a Clip

In order to create notes, we need to add a clip to the arrangement. This aggregates a clip to be added to the arrangement's clip list, which in turn has a list of notes generated using my note generation algorithm.

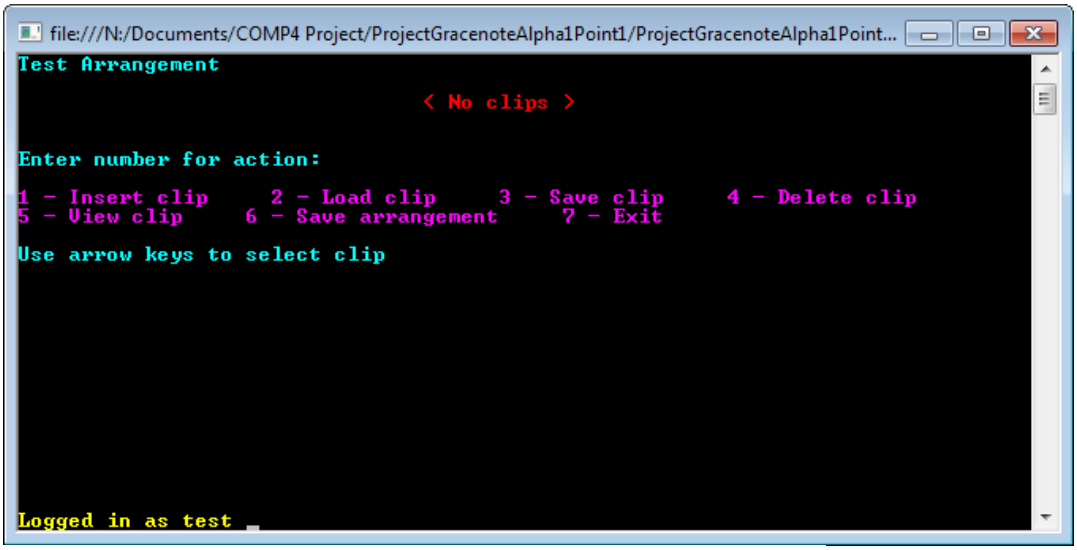


Figure 5.18 *– the arranger for the new arrangement. Press 1 key*

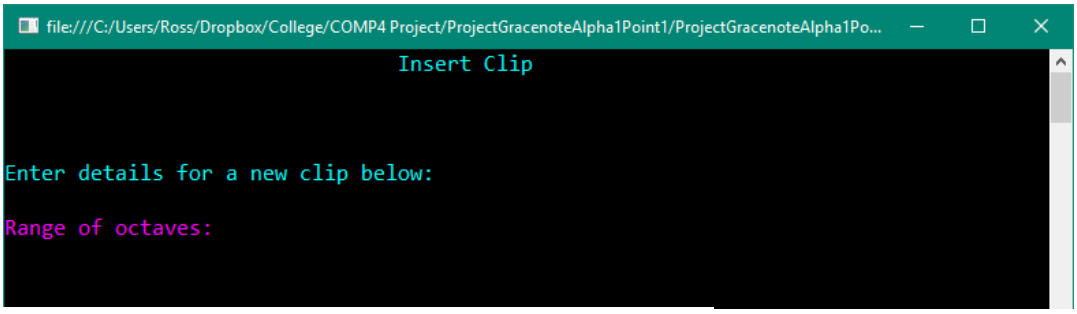


Figure 5.19 *– the insert clip wizard. Enter clip details*

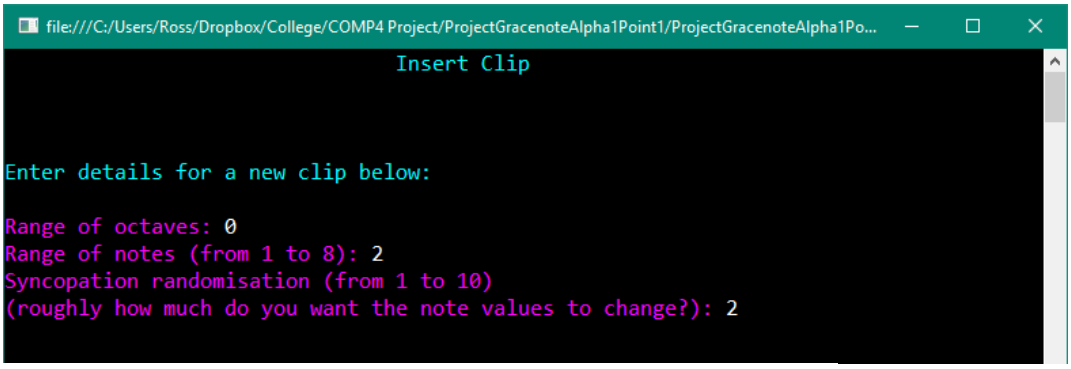


Figure 5.20 *– insert clip wizard with data. Press Enter*

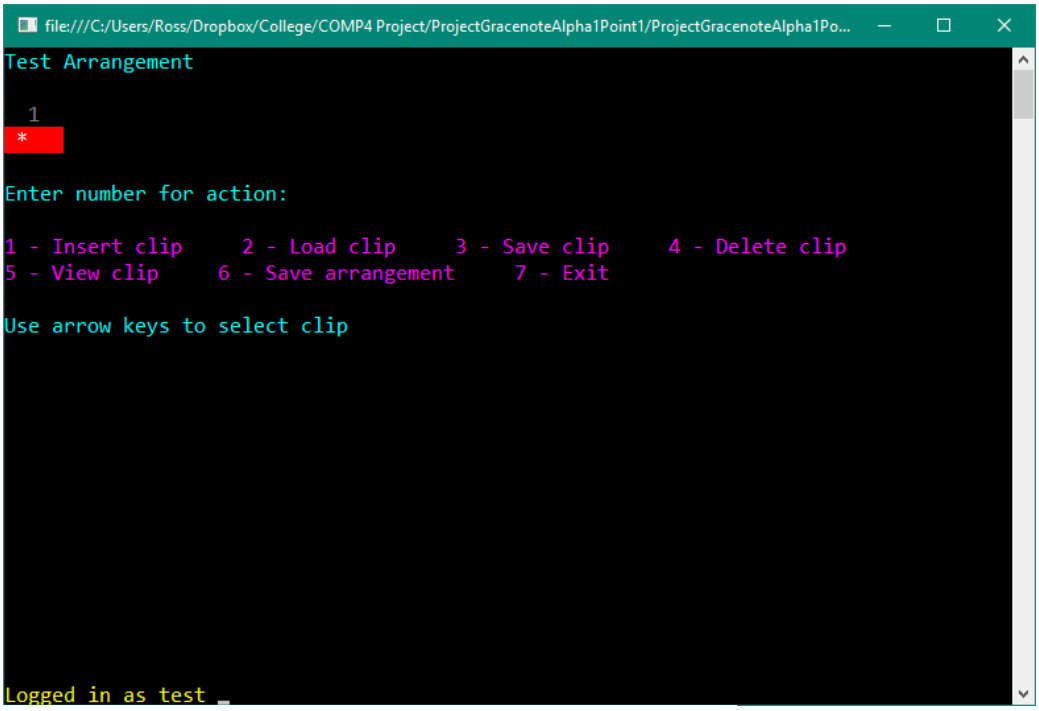


Figure 5.21 *– arrangement with 1 clip added. Press 5 key*

In **Figure 5.20**, the user enters data for the new clip. As the range of octaves is the range above and below octave 4 that the notes are allowed to be in, I have chosen 0 to make sure that the notes only stick to octave 4, for simplicity. I have chosen 2 for range of notes (the range of pitches across the octave) and syncopation randomisation to give a decent spread of different notes varying in pitch and note value.

Within the arrangement object is an aggregated list of type clip. Each clip, like the one shown in **Figure 5.21**, is an object, where the inserting of data in **Figure 5.20** defines the properties needed to create the new notes.

Within each clip is a list aggregating notes. The notes are randomly generated and are only allowed if they fit the key and time signature (I will go over the algorithm in the **System Maintenance** section on page 150). The randomly-generated notes are only created if they meet the constraints that the user entered. The program keeps generating notes until it has a whole bar's worth.

Once the clip is added in **Figure 5.21**, here is the contents of the list of clips in the arrangement object:

***Figure 5.22** – the contents of a clip after notes have been created*

All unsaved clips, such as that in **Figure 5.21**, are shown in red with an asterisk.

2.5 Viewing the New Clip

If we want to view the clip objects, another algorithm (which I will go over in detail in the **System Maintenance**) draws the notes as blocks of colour for every octave that's included:

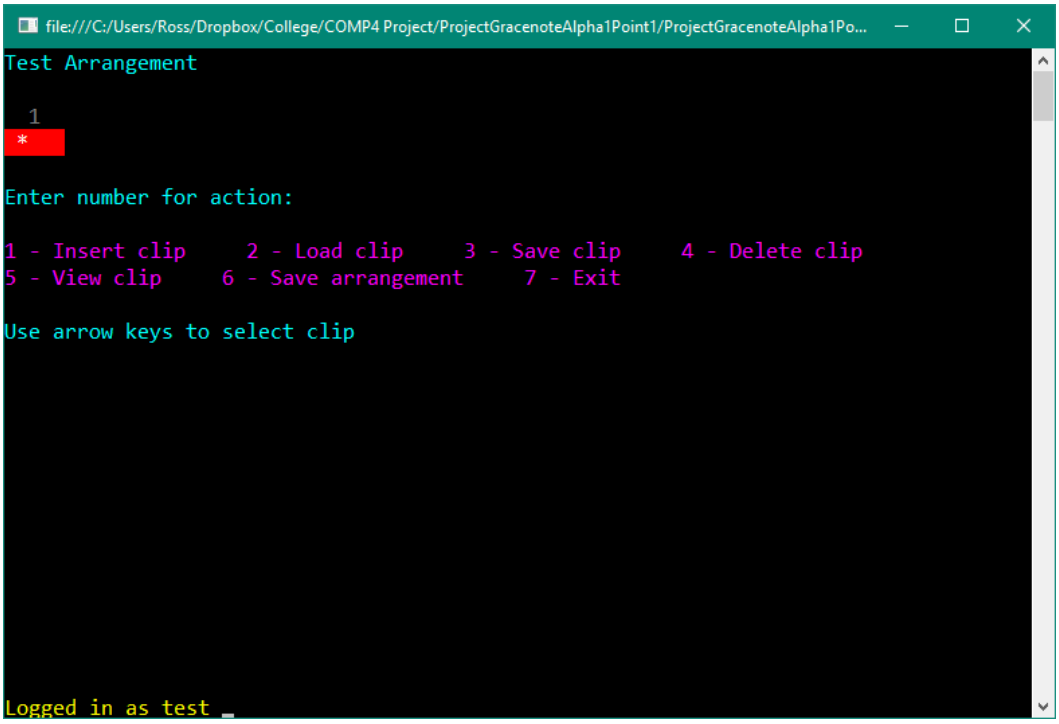


Figure 5.23 *– arrangement with 1 clip added. Press 5 key*

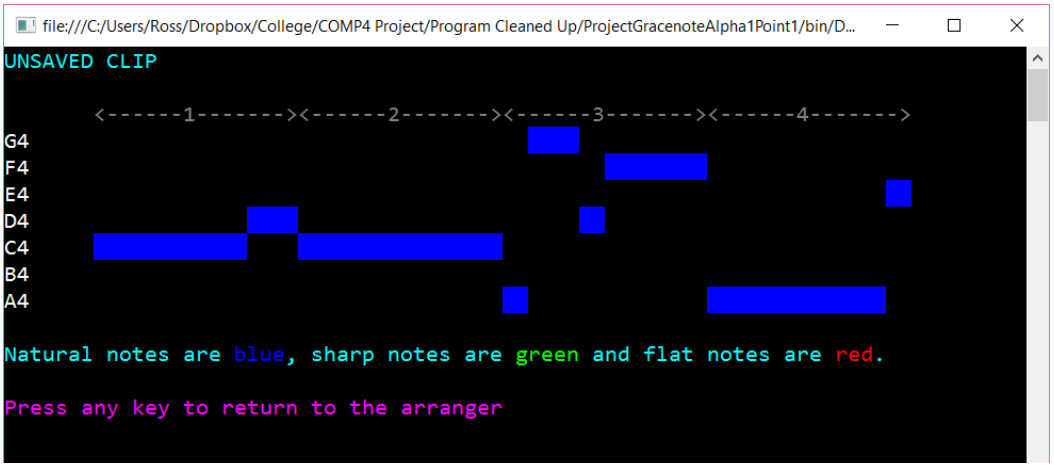


Figure 5.24 *– viewing the clip*

Each block of colour represents a note object. The width of each note shows the note value. The row represents the pitch. The colour represents whether it is natural (not sharpened or flattened), sharp or flat. All of the notes are blue, which means they are all natural. We know this is valid, because C major, which is the key that we set the arrangement to, does not sharpen or flatten any notes.

You can see that there are 7 rows: G4, F4, all the way down to A4. The location of the notes in each of the rows is represented by 7 string variables.

As the user presses the 5 key in the arranger (**Figure 5.23**) to view the clip, the program reads through all of the notes in that clip and marks out on these 7 strings where the notes should be positioned.

Before the notes are displayed, the variables look like this:

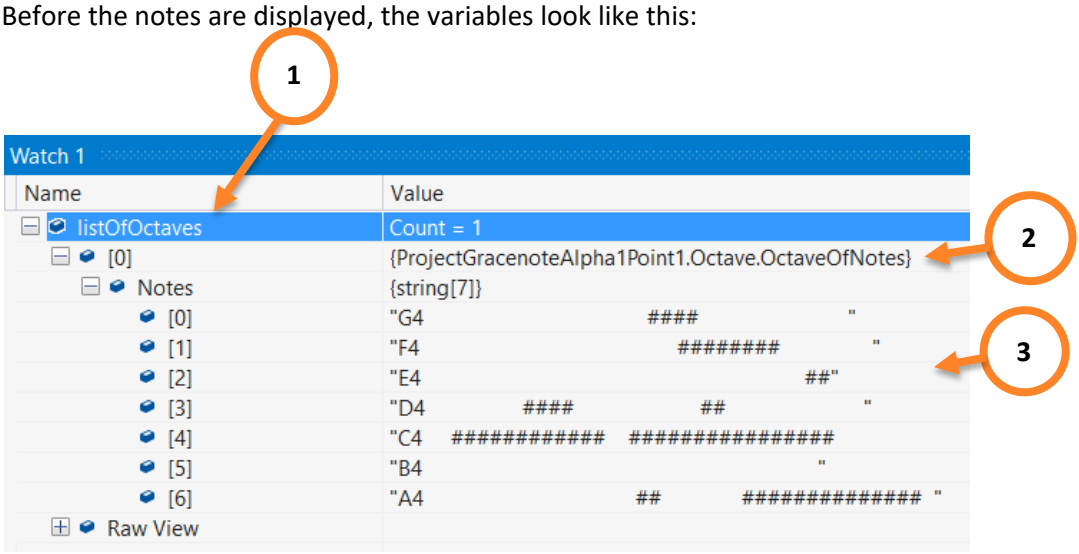


***Figure 5.25** – the octave variables*

1 The listOfOctaves is a list that contains user-defined objects, which I've called OctaveOfNotes. These objects each contain an array of the rows of notes to be displayed to the user. Each object is specific to one octave of notes.

2 This is the OctaveOfNotes object that is being used. As our clip only has notes in one octave (octave 4), there is only one object. If there were 2 octaves, there would be 2 objects, and so on.

3 Here, you can see the rows of notes that we saw in **Figure 5.24** in character form. You can see that the positions of the notes are marked out by the hashes. Where hashes are next to each other, they form one note. The length of each chain of hashes represents the length of the note. To represent the note as a natural, hashes are used. To represent a sharp note, slashes (/) are used. To represent a flat note, asterisks (*) are used.

Once these strings have been created, the program scans across each string. If there is a space, it prints an empty blank space in the console. If it detects a #, it prints a blue blank space; if it detects a /, it prints a green blank space; if it detects a *, it prints a red blank space. The colours are determined by changing the console's background colour each time a space is printed. The blocks of colour are printed, along with the other information to display the notes.

Here is another example which contains both natural and sharp notes:

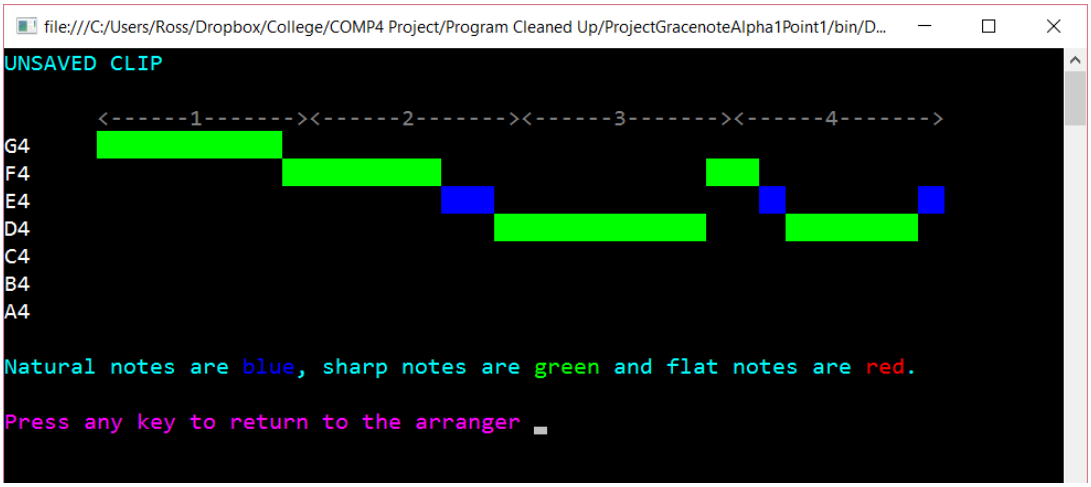


Figure 5.26 *– example clip with natural and sharp notes*

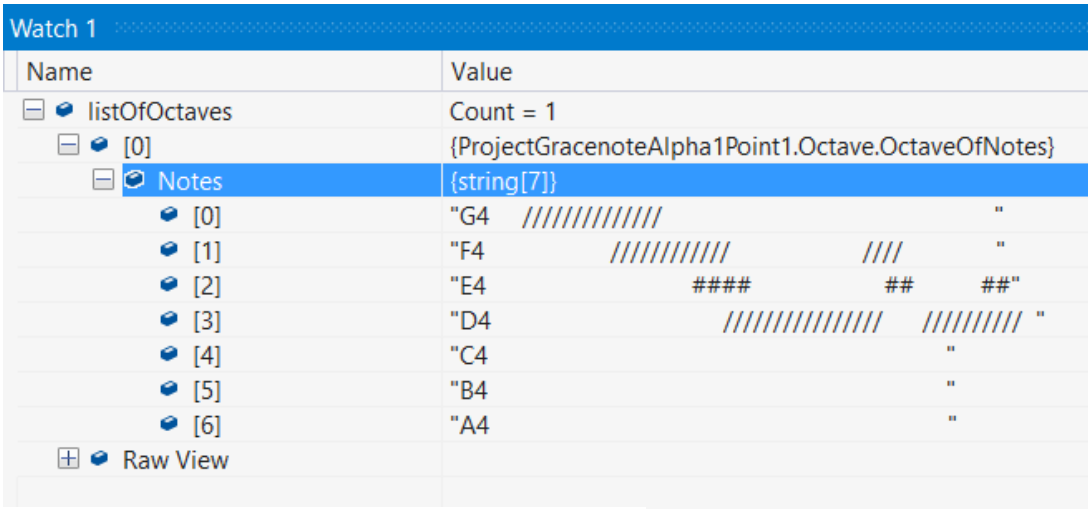


Figure 5.27 *– the example clip in character form*

2.6 Using Multiple Clips

If you have multiple clips in your arrangement, one will show as a bright red and the other will show as a darker red. The bright red one is the one that is selected. If you press the right arrow key to move to the other clip, the colours change:

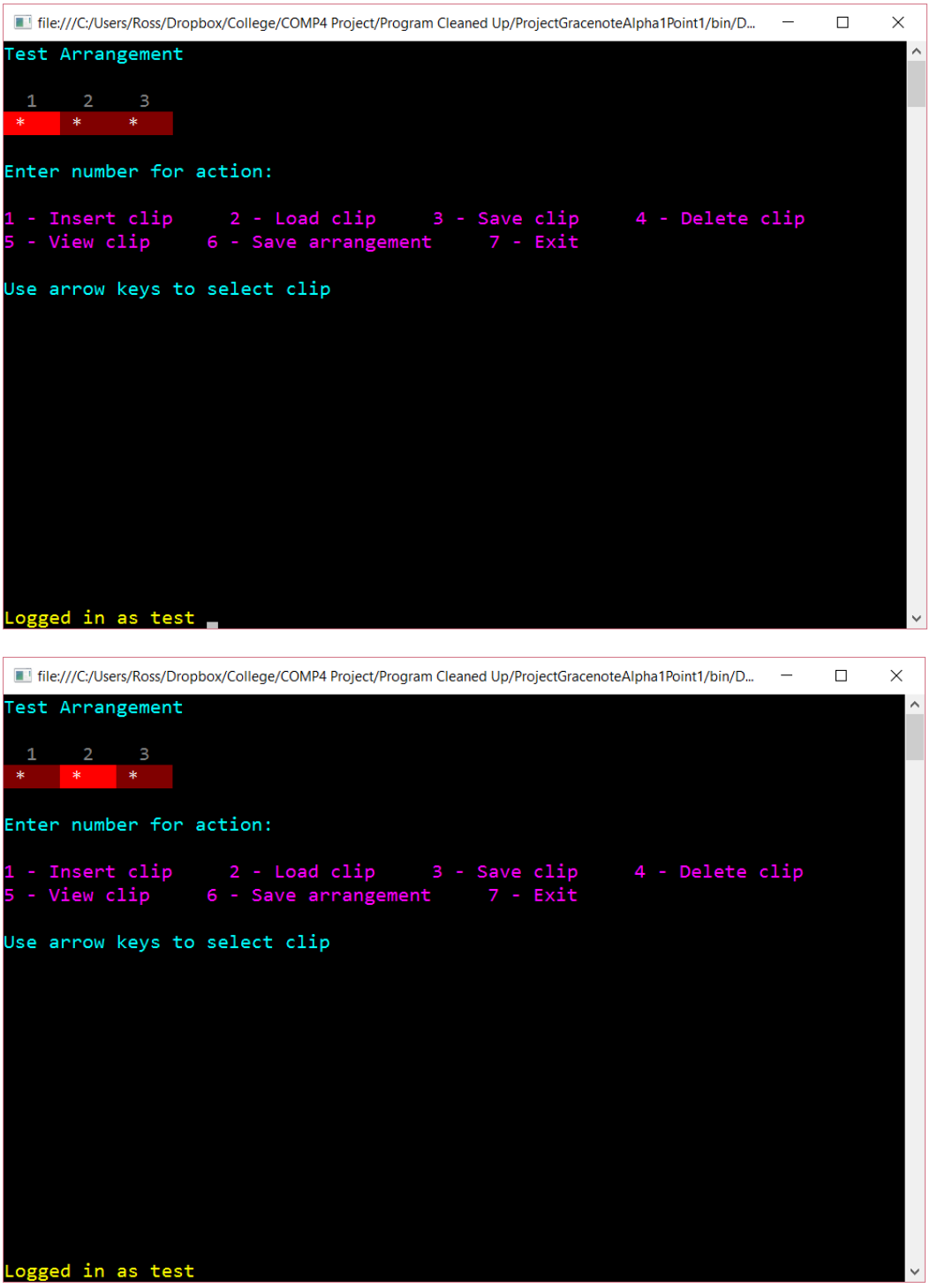


Figure 5.28 *– moving between 3 clips*

2.7 Saving an Arrangement with Unsaved Clips

Once you have a series of clips, you can save the arrangement to the database. This updates the existing arrangement record.

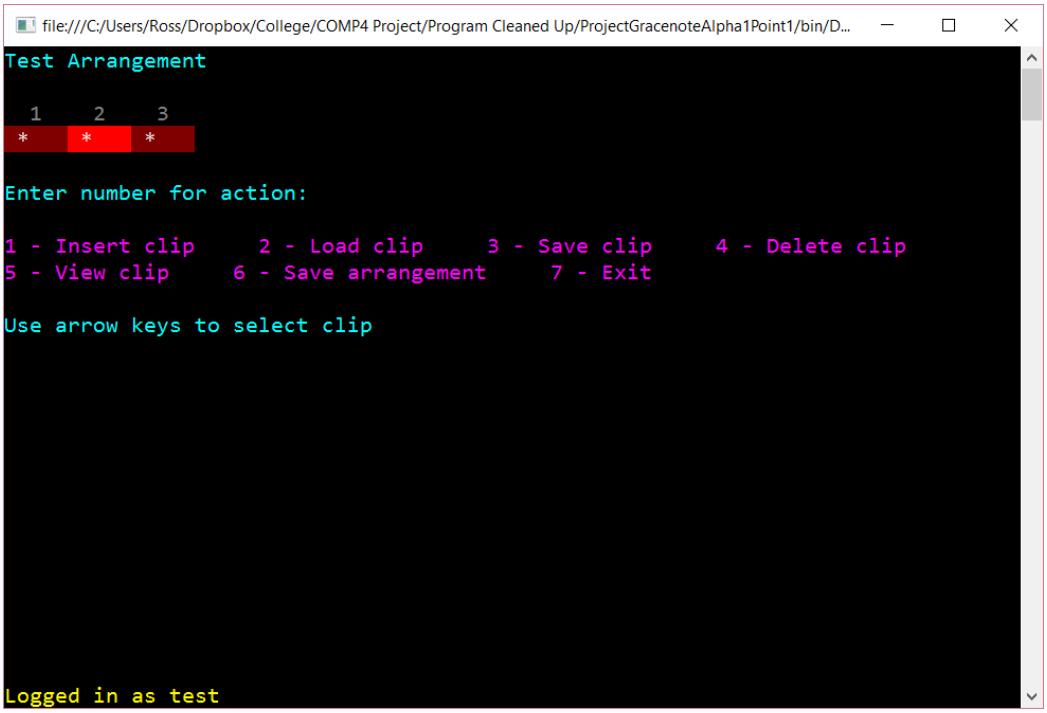


Figure 5.29 *– press 6 key*

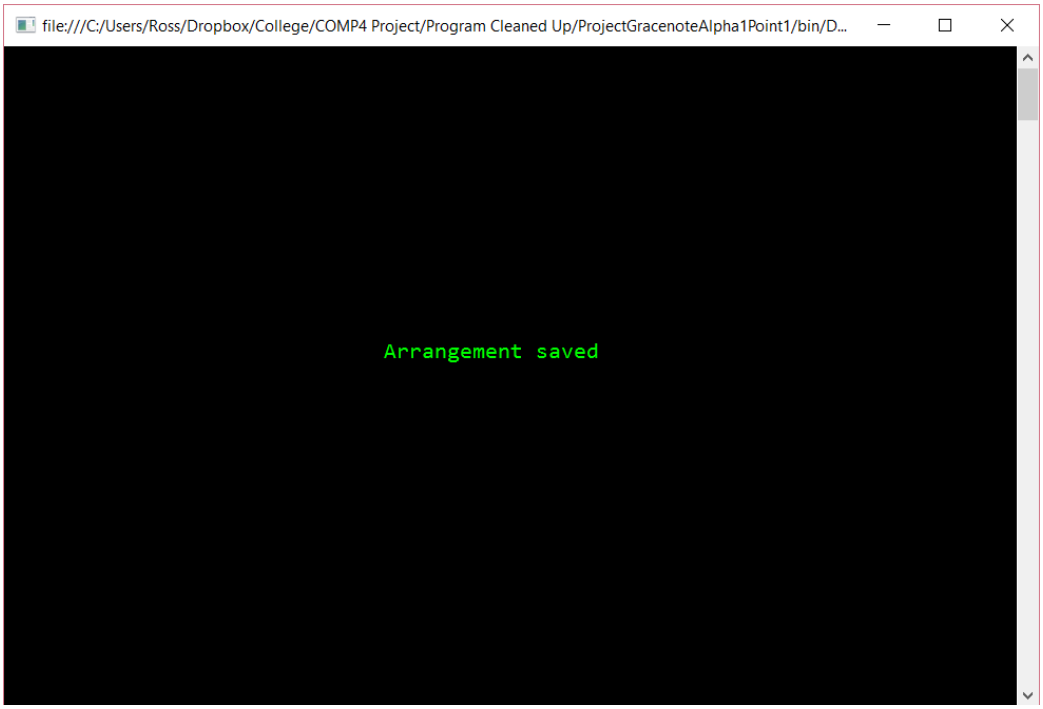


Figure 5.30 *– saved message. This is displayed for 1 second before you are returned to the arranger as in* ***Figure 5.29***

Before **Figure 5.23** is shown, the Arrangements table in the database is updated. Before we pressed the save button, the arrangement looked like this:

PK_ArrangementID	ArrangementName	ArrangementData	ClipContentsData	FK_AccountID
8	Test Arrangement	0011C24	*NULL*	4

Figure 5.31 *– arrangement record in database before saving the clips*

Remember that 4 represents the account number for our test account.

After the clips are saved, the record looks like this:

PK_ArrangementID	ArrangementName	ArrangementData
8	Test Arrangement	001011C24BE1948331075FBE313193004FBE1674691964F

ClipContentsData	FK_AccountID
E1948331075FXA3B03C4YXA4B02C4YXA0B05C4YXA4B02C4YXA...	4

Figure 5.32 *– arrangement record after saving the clips*

This might look like gibberish, but essentially, the program has serialised the clip and note data.

First, let's deconstruct the text in the ArrangementData field:

001011C24BE1948331075FBE313193004FBE1674691964F

We can ignore this part, as this is the main arrangement properties (key and time signature) that I went through earlier.

B means the start of an unsaved clip. The fact that there are three Bs shows that there are 3 clips, as there are.

This is a key which makes a reference to the clip's note data in the ClipContentsData field. E means the start of the key and F means the end of the key. Between these two letters is a randomly-generated integer; each time a key is generated, there is a check done to make sure that that integer hasn't been generated already for another key, which means each key is unique.

Now, let's take a look at the ClipContentsData field:

E1948331075FXA3B03C4YXA4B02C4YXA0B05C4YXA4B02C4YXA3B03C4YXA0B05C4YXA2B06C4YXA1B03C4YE313193004FXA4B06C4YXA2B05C4YXA4B02C4YXA1B01C4YXA2B05C4YXA4B01C4YXA1B02C4YXA2B03C4YXA4B06C4YXA3B01C4YXA4B03C4YXA4B03C4YE1674691964FXA1B05C4YXA2B04C4YXA2B05C4YXA4B03C4YXA1B05C4YXA1B06C4YXA2B04C4YXA4B05C4Y

We can see that the three keys that we identified are contained within this text. After each key is the data representing the notes within the clip that the key belongs to. Let's look at the data for the first clip (the text in blue):

XA3B03C4YXA4B02C4YXA0B05C4YXA4B02C4YXA3B03C4YXA0B05C4YXA2B06C4YXA1B03C4Y

The Xs mark the start of a note and the Ys mark the end of a note. The text in black is note properties. Let's look at one of these notes:

XA3B03C4Y

A means that the following number is the code for the note value. 3 is the code for a crotchet (1/4 note).

B means the following two numbers mean the note's pitch. 03 is the code for E natural.

C means the following number represents the octave. This note is in octave 4.

2.8 Saving a Clip

This is where you give the clip a name and its note data gets saved in the Clips table, with a foreign key linking to your account ID. This is used if the user wants to save particular clips for use later, or in another arrangement.

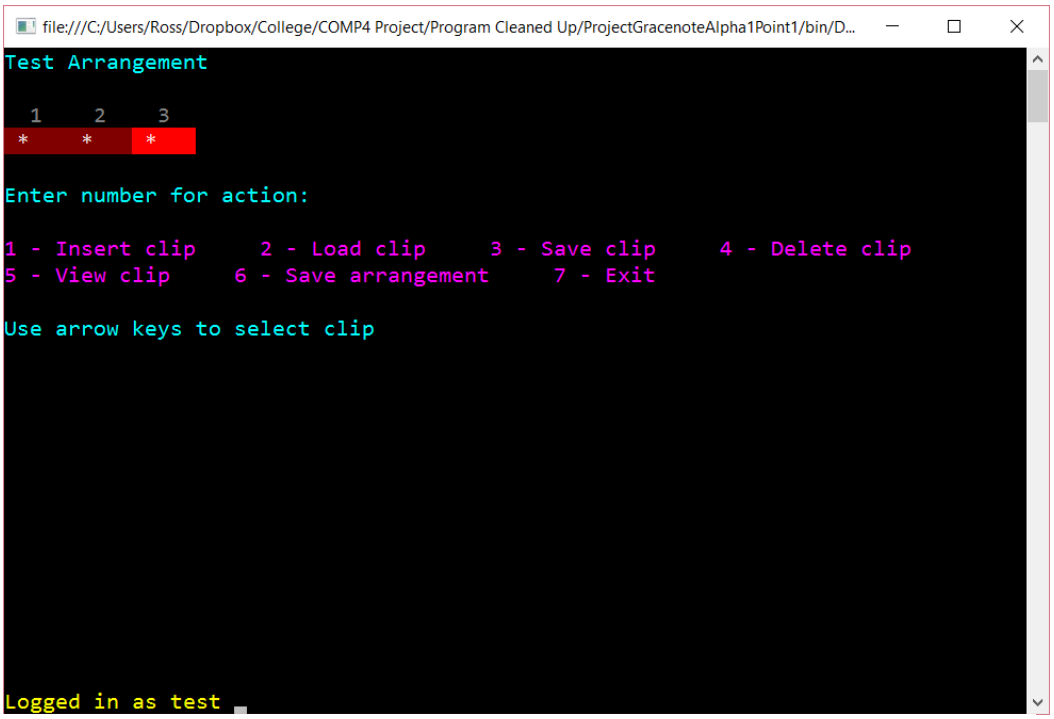


Figure 5.33 *– arranger, where the clip that we want to save is selected. Press 3 key*

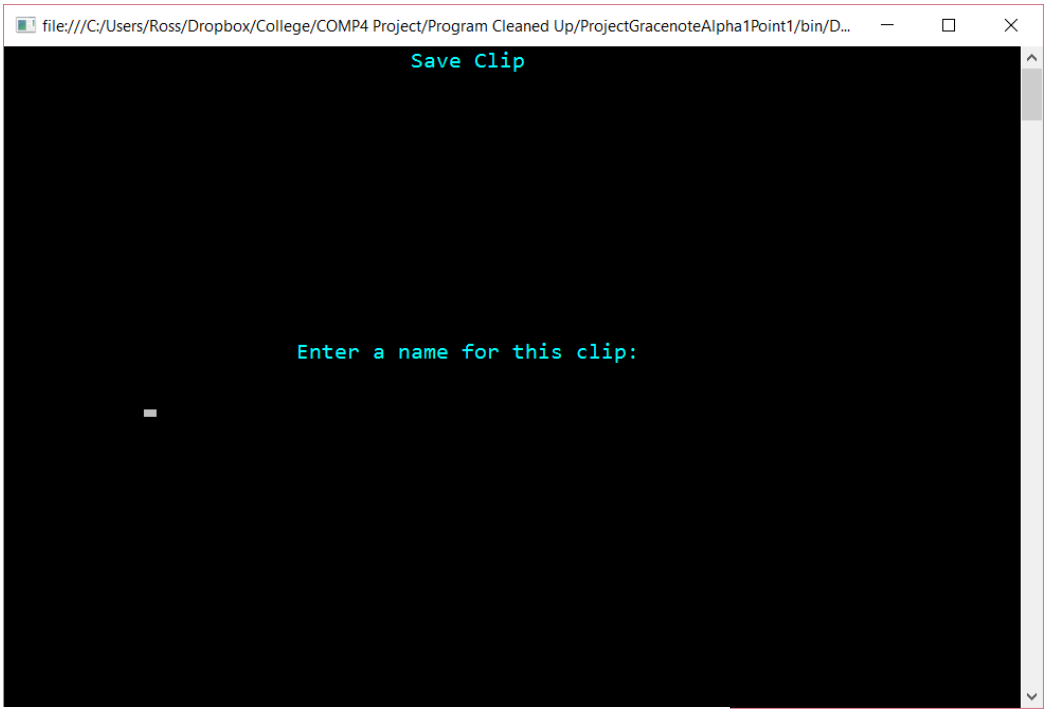


Figure 5.34 *– save clip wizard. Enter a name for the clip*

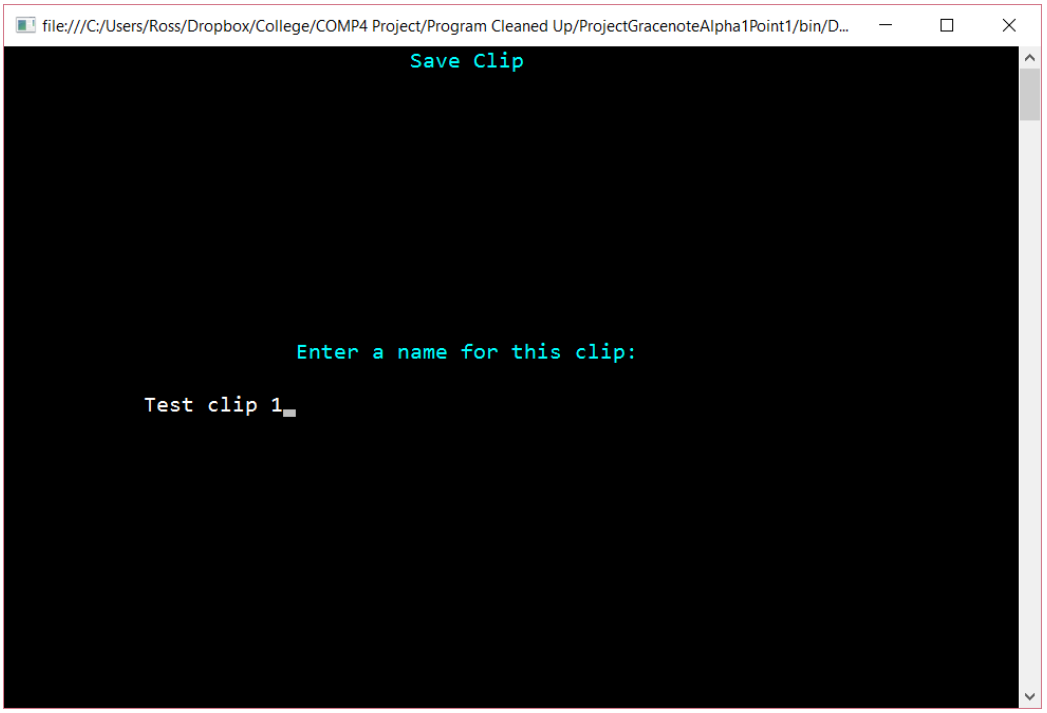


Figure 5.35 *– name is entered. Press Enter*

Once the user presses Enter, a new record in the Clips table has been created for this clip:

PK_ClipID	ClipName	NoteData	FK_AccountID
2	Test clip 1	XA1B05C4YXA2B04C4YXA2B05C4YXA4B03C4YXA1B05C4YXA1B0...	4

Figure 5.36 *– record in the Clips table*

We can see that the notes have been serialised into the same format that we saw earlier. We can also see that it is linked to our account ID.

To link the clip and its use within the arrangement, we insert a new record into the ClipUses table:

FK_ClipID	FK_ArrangementID
2	8

Figure 5.37 *– record in the ClipUses table*

Additionally, the properties of the 3rd clip object have been changed. Let's look at what the properties show now:

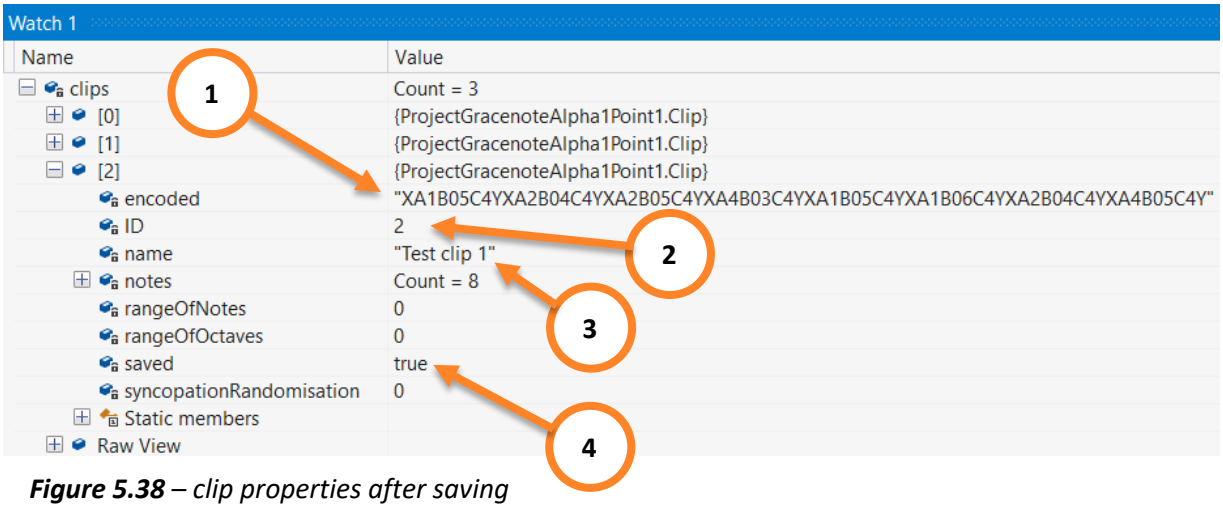


Figure 5.38 *– clip properties after saving*

1 The program serialises the note data into the format ready for the database.

2 After we insert the serialised data into a new record in the Clips table, we retrieve the ID that was given to the record. This is important for later.

3 Because we gave the clip a name, this name is stored here. This is needed for when the clip is shown in the arranger, where the name is printed.

4 As the clip has been saved, we mark it as being so with this variable. This is needed so that when the clip is displayed in the arranger, it is shown as green and not red.

The saved clip is then shown in the arranger as green instead of red:

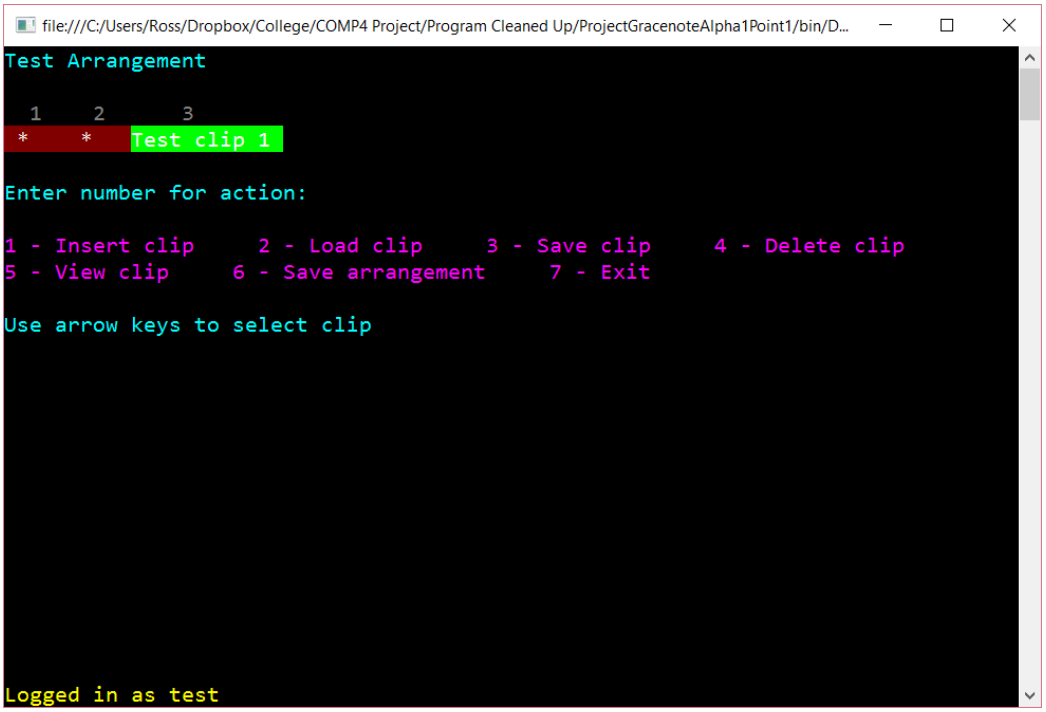


Figure 5.39 *– arranger with saved clip*

2.9 Saving an Arrangement with Saved Clips

Here, I have created a brand new arrangement consisting of two saved clips and an unsaved clip to demonstrate the difference in how saving is done here:

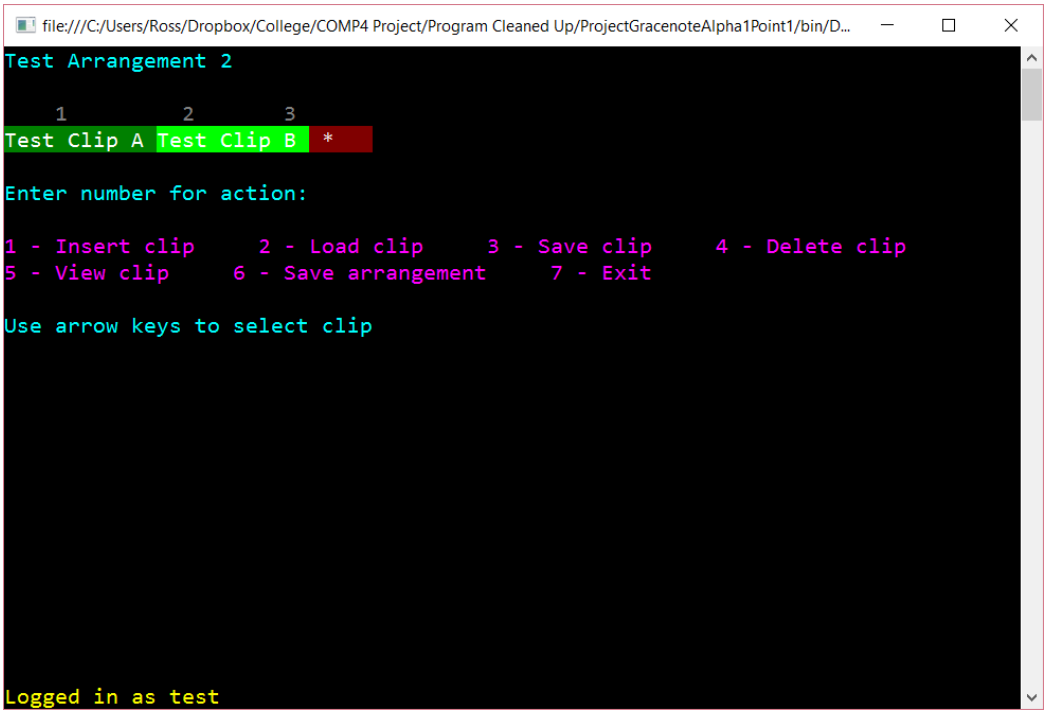


Figure 5.40 *– arranger. Press 6 key*

Once the user has pressed the 6 key, the Arrangements table gets updated just like before, except this time, the data is different:

PK_ArrangementID	ArrangementName	ArrangementData
8	Test Arrangement	001011C24BE1948331075FBE313193004FBE1674691964F
9	Test Arrangement 2	0011C24A39A49BE304842317F

ClipContentsData	FK_AccountID
E1948331075FXA3B03C4YXA4B02C4YXA0B05C4YXA4B02C4YXA...	4
E304842317FXA1B02C4YXA4B04C4YXA0B01C4YXA1B05C4YXA4...	4

Figure 5.41 *– arrangements table after saving the arrangement with saved clips*

The first row is for the first arrangement that we made earlier. Let's focus on the ArrangementData field for the second one:

0011C24A39A49BE304842317F

We can see that the field contains the properties of the arrangement, and the key for the 3rd clip that was unsaved, but for the first two clips, we have something different.

A means the start of a saved clip. The number after is the compound primary key of the saved clip in the ClipUses table.

Let's look at the ClipUses table again:

FK_ClipID	FK_ArrangementID
2	8
3	9
4	9

Figure 5.42 *– ClipUses table again*

The two clips which are in our second arrangement, Test Clip A and Test Clip B, are the clips with clip IDs 3 and 4 (the second and third rows in the table). Combined with the arrangement ID, they give the compound keys 39 and 49, which relate to our ArrangementData field's contents.

For **Figure 5.38**, I mentioned how it was important that we stored the ID of the clip in the clip object. This is so that we could insert it into the serialisation. When the arrangement object is created at the very start, the ID for that is also retrieved and stored as a variable:

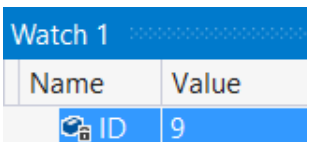
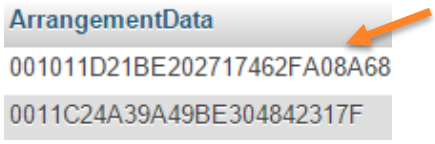


Figure 5.43 *– the arrangement ID*

This thus allows for the concatenation of 3 and 9, and 4 and 9.

Unfortunately, there exists a bug with this part of the program, in that sometimes, saving an arrangement with saved clips leads to invalid serialisation:



This is an impossible key. It was originally A58, which was valid, but has now changed. The 5 was the clip ID part of the compound key, but is now 0. You cannot have a clip ID as 0.

When this happens, the arrangement loads with no clips.

I have no idea why this happens and I was unable to debug it in the time that I had available for programming. Thus, I have let the end user know about this in the user guide, assuring them that this is an early alpha version of the program and that I aim to fix it by the next version.

2.10 Loading a Clip

When a user saves a clip, it links to their account. Therefore, they can re-insert it into any other arrangement.

Here, I've created a blank arrangement, into which I inserted a clip that I saved earlier:

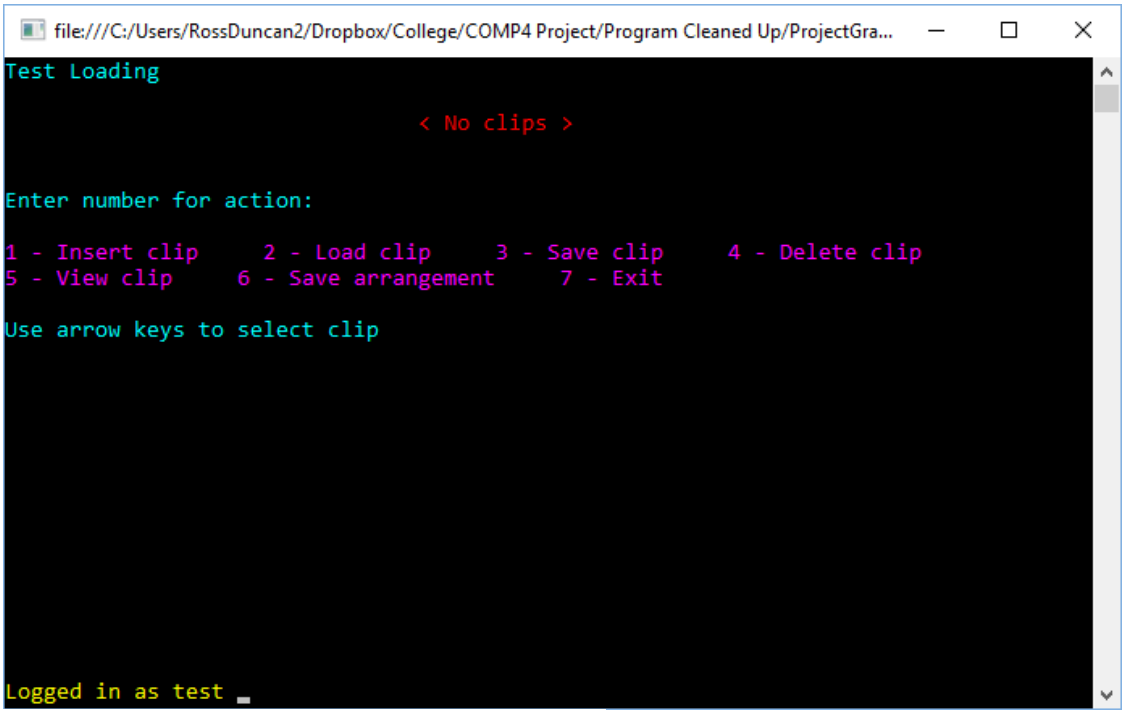


Figure 5.44 *– blank new arrangement. Press 2*

The program then reads the names of every clip saved under the user's account using SQL:

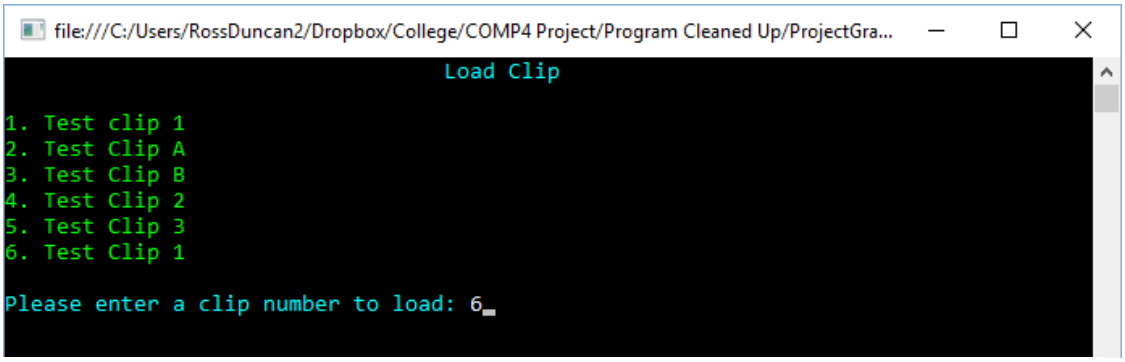


Figure 5.45 *– list of clips available to load with choice of 6 (Test Clip 1) made. Press Enter*

The clip data is loaded using SQL from the database and a clip object is created using that data and added to the list:

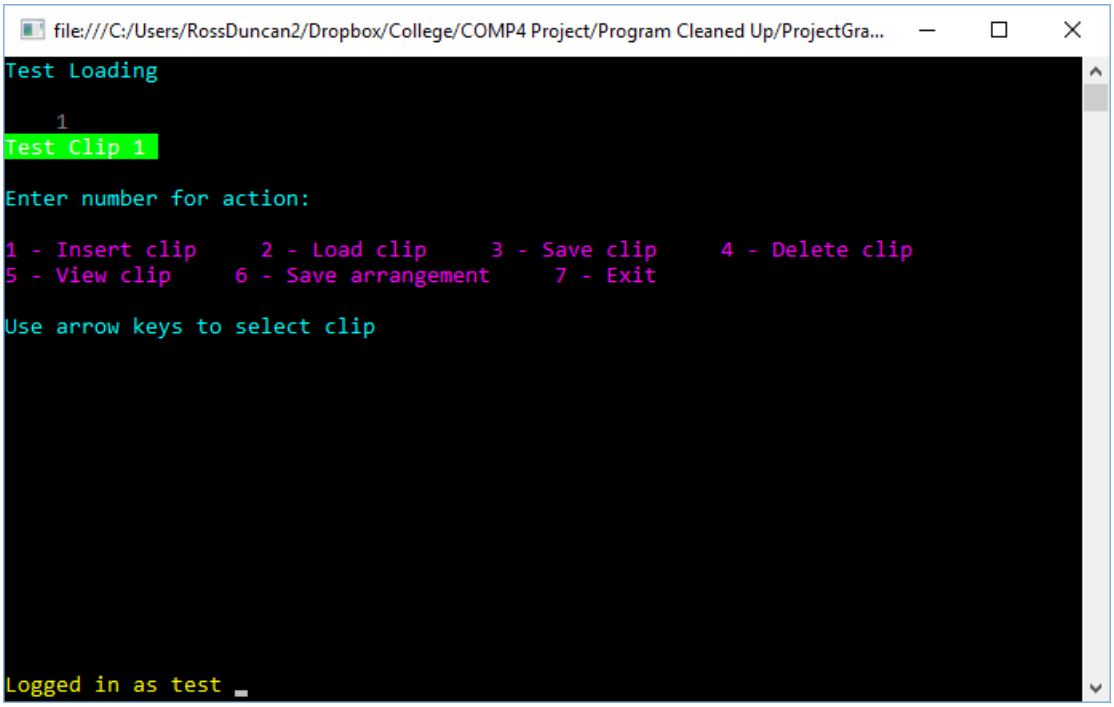


Figure 5.46 *– arrangement with clip added*

To prove that the clip is loaded properly, I then pressed 5 to display the notes:

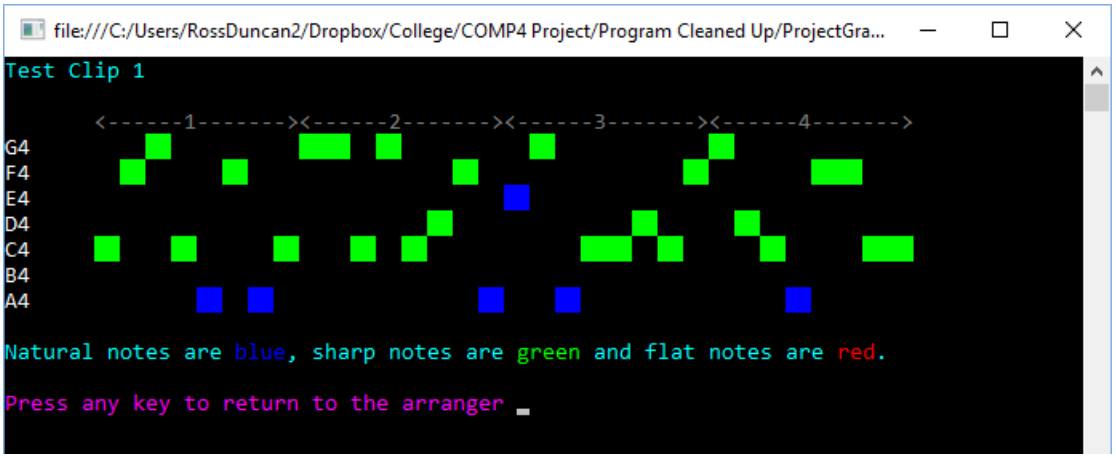


Figure 5.47 *– displaying notes of loaded clip*

The fact that it contains notes means that the note data has been successfully unserialised.

2.11 Deleting a Clip

It is possible for a clip to be deleted from an arrangement.

Here we start with an arrangement with 3 clips:

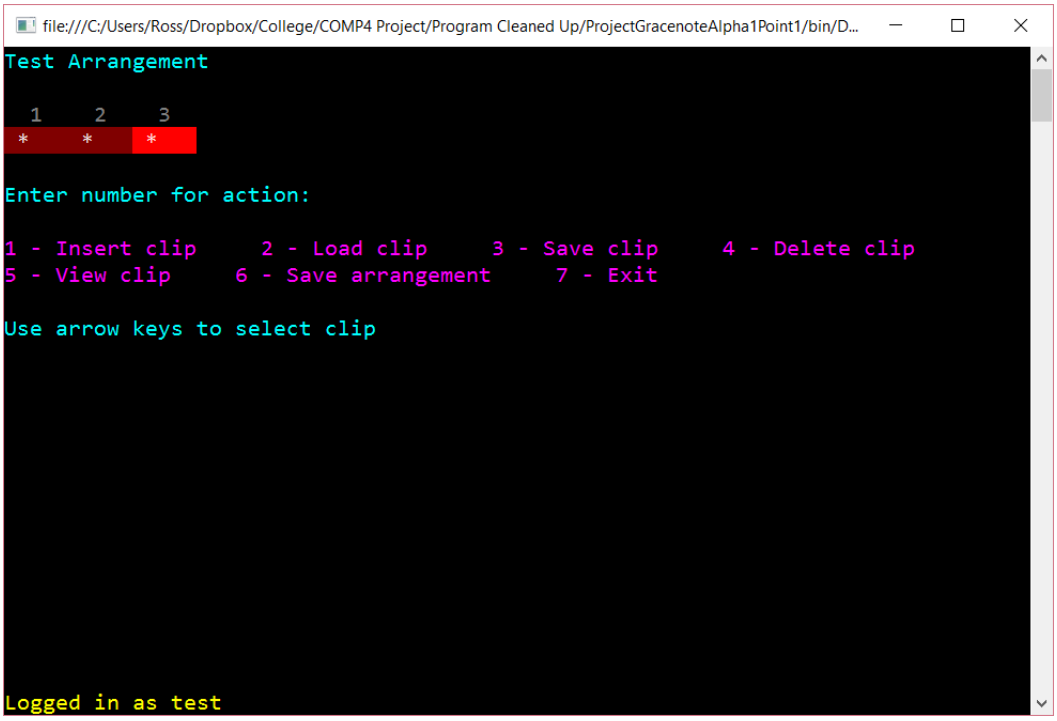


Figure 5.48 *– arrangement with 3 clips. Press 4 key*

In the arrangement, there is a variable that keeps track of the current index in the clips list that is selected. Before the clip got deleted, the variables in the arrangement object looked like this:

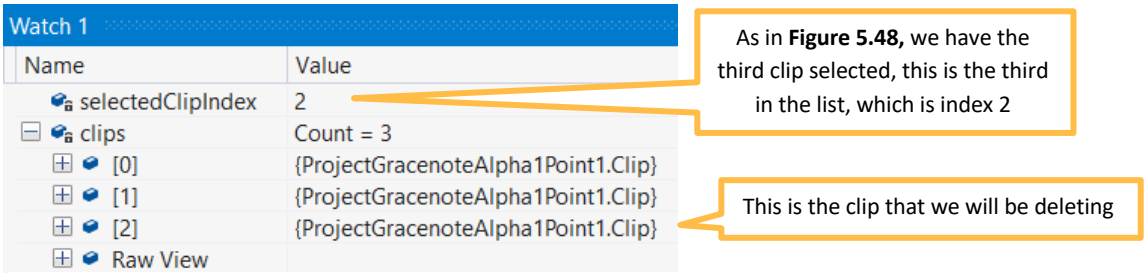


Figure 5.49 *– the clips in the arrangement before deletion*

After we press the 4 key to delete the clip in **Figure 5.48**, this is what the list looks like:

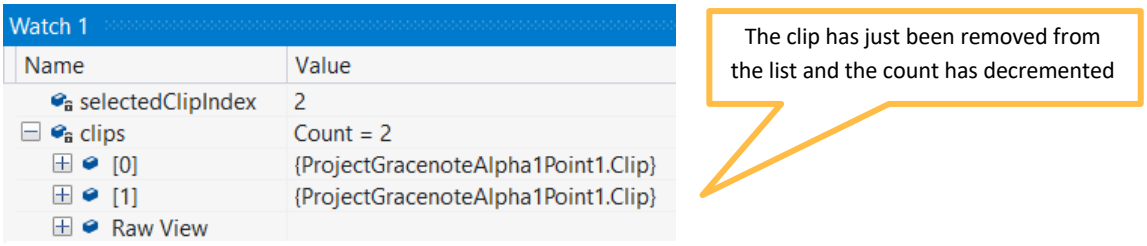


Figure 5.50 *– the clips in the arrangement after deletion*

And this is the output:

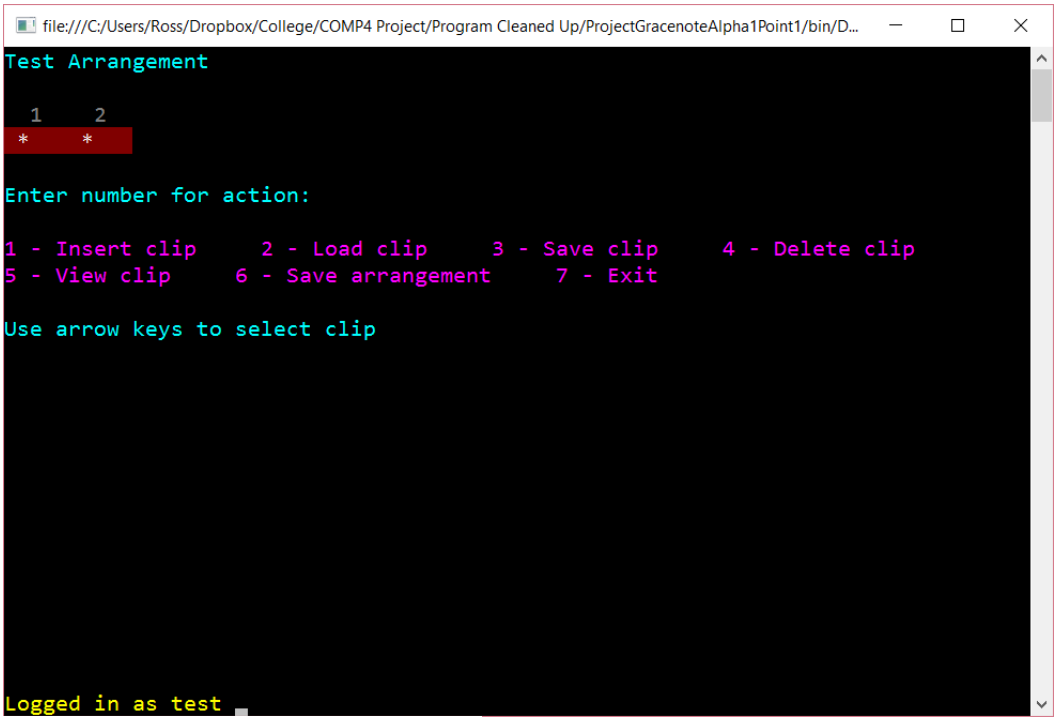


Figure 5.51 *– the clip is now deleted*

2.12 Loading an Arrangement

I have saved an arrangement called Test Arrangement 3, which has 4 red clips in it. We will now load it.

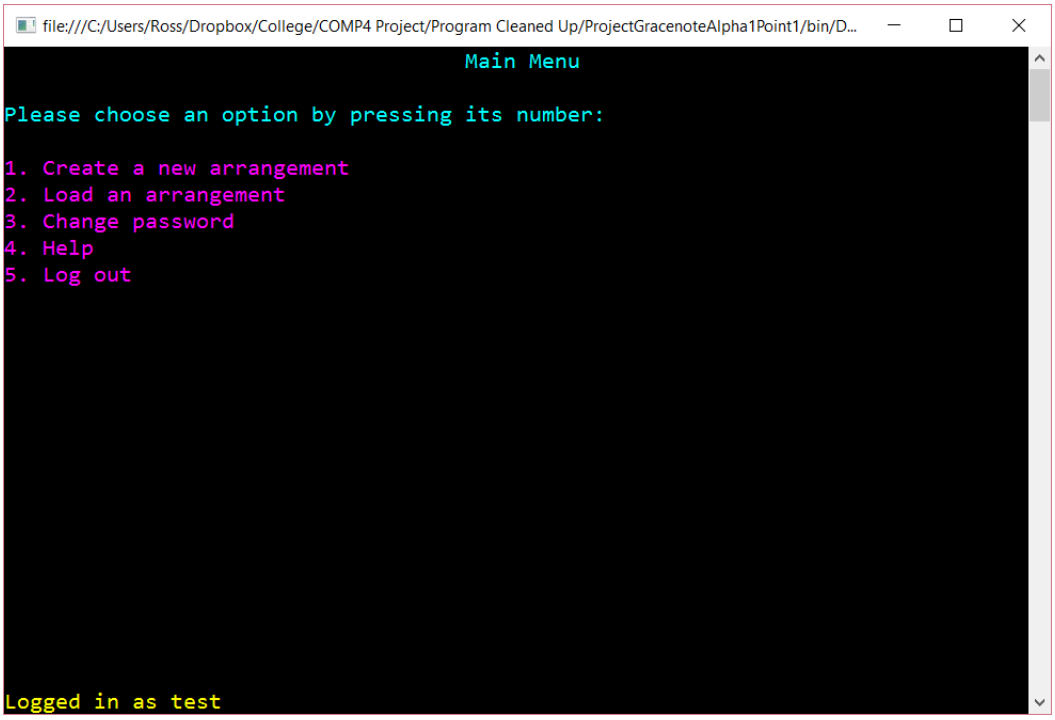


Figure 5.52 *– the main menu. Press 2 key*

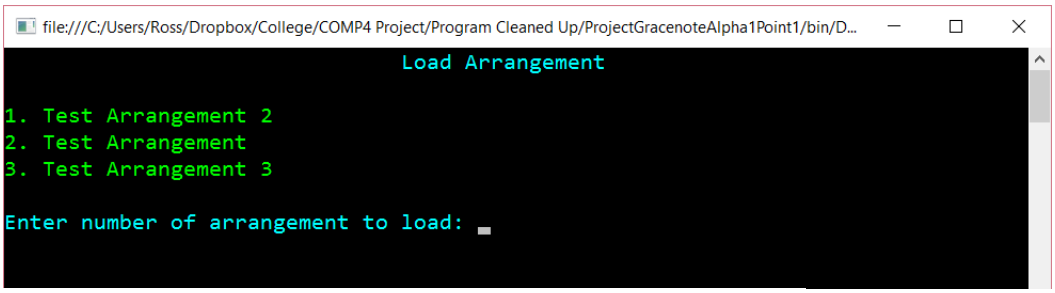


Figure 5.53 *– load arrangement wizard. Enter number in the list*

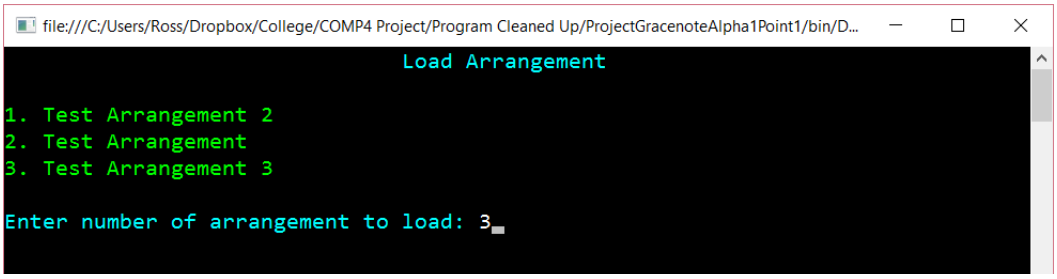


Figure 5.54 *– load arrangement wizard with choice entered. Press Enter*

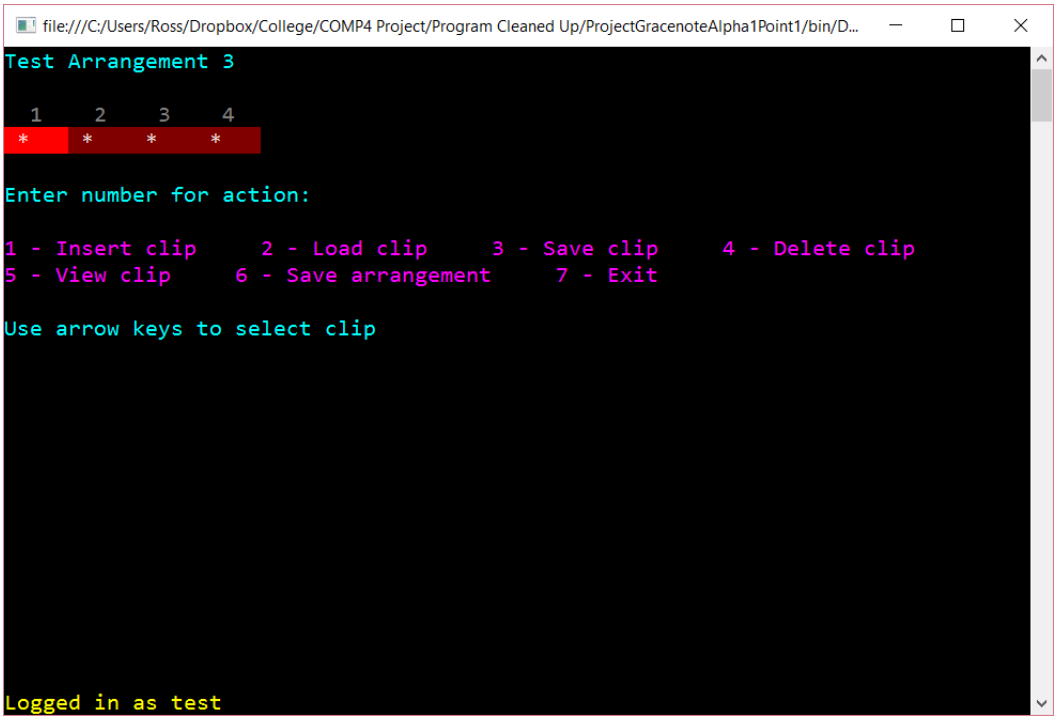


Figure 5.55 *– Test Arrangement 3 has loaded*

The program loads the arrangement data and clip contents data fields from the arrangements table in the database, and stores them in variables:

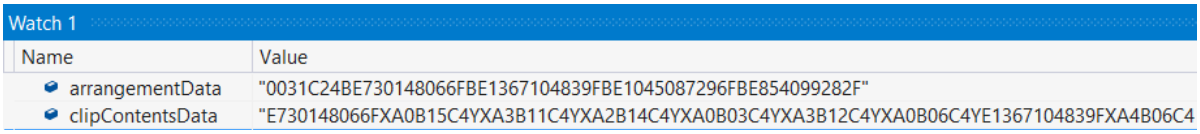


Figure 5.56 *– retrieved data (clipContentsData does go further)*

The program scans these strings and extracts the properties into variables and makes up the clips:

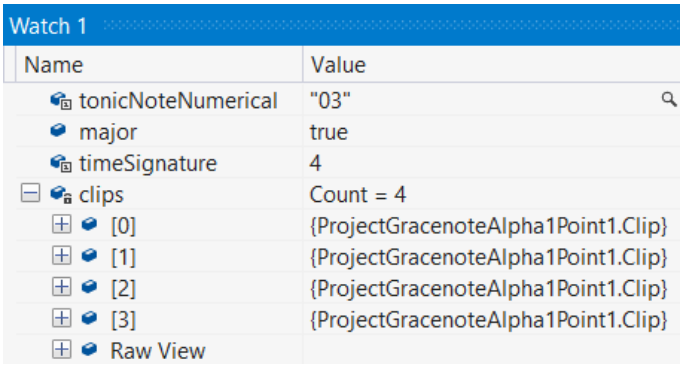
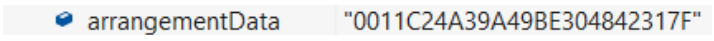
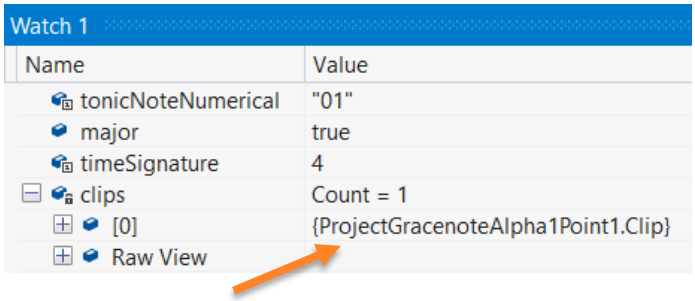


Figure 5.57 *– data as variables*

Sadly, there is a bug with this part of the program as well. If the arrangement contains any saved clips, such as here:



…we can see that there are two saved clips, A39 and A49, and that there is also an unsaved clip afterwards. However, when the data is extracted, we get this result:



Only one clip has been loaded, when there should be 3. The program loaded the first saved clip and then didn't load any of the others. Again, I was unable to debug this problem in the time that I had available and I have stated this in the user guide.

2.13 Changing Your Password

As part of acceptable security measures, I have given the user the ability to change their password from the main menu:

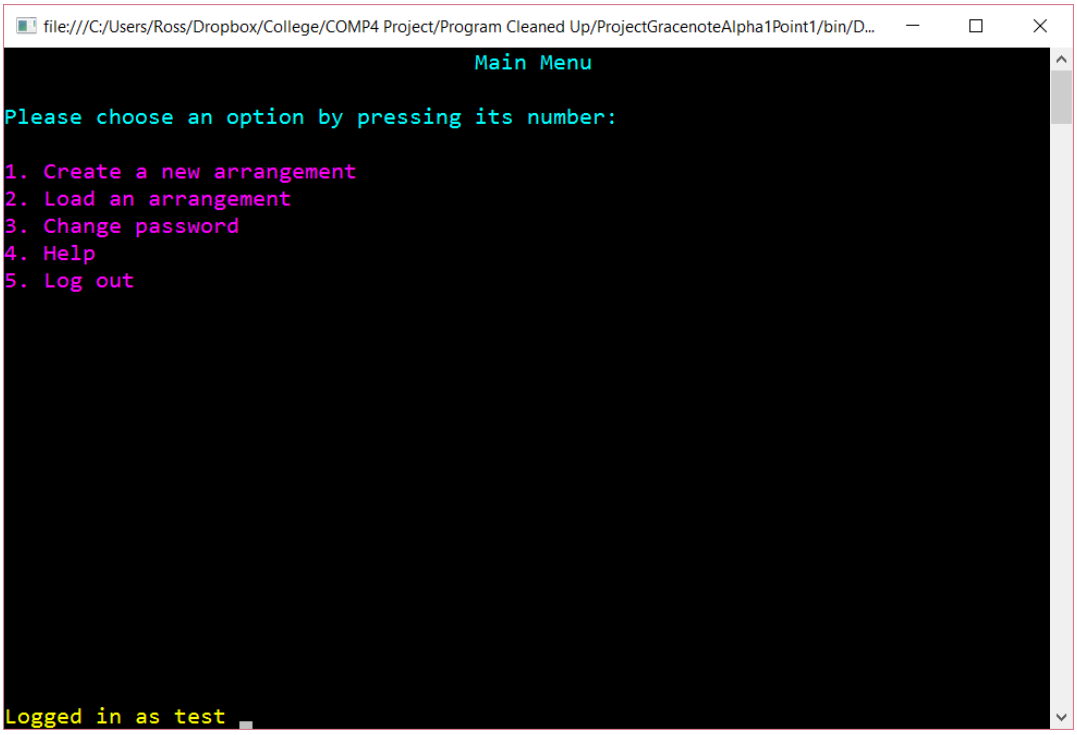


Figure 5.58 *– main menu. Press 3 key*

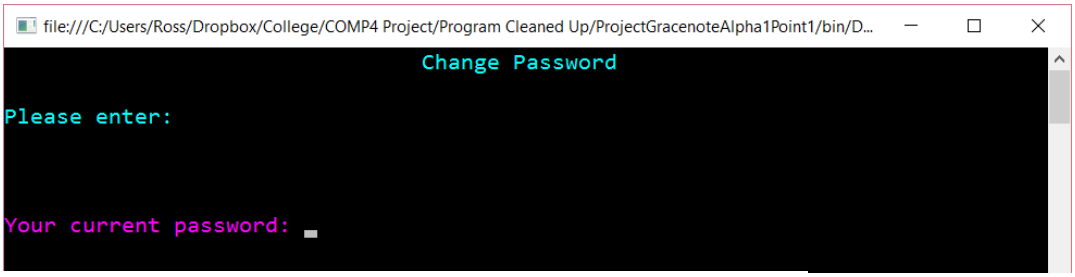


Figure 5.59 *– change password wizard. Enter details it asks for*

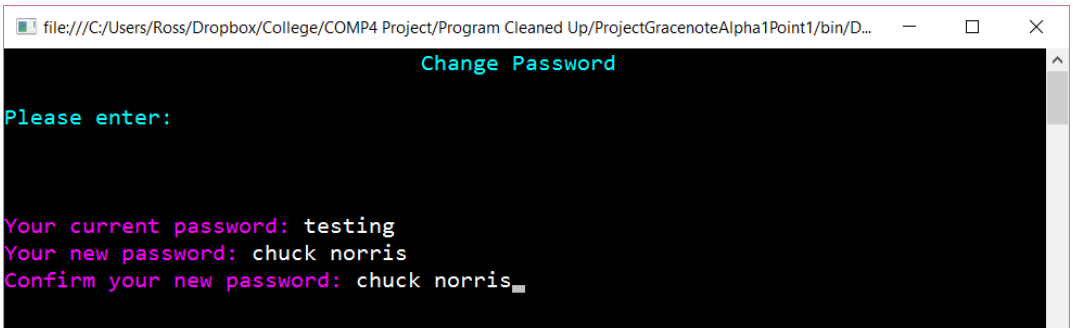


Figure 5.60 *– change password wizard with details entered. Press Enter*

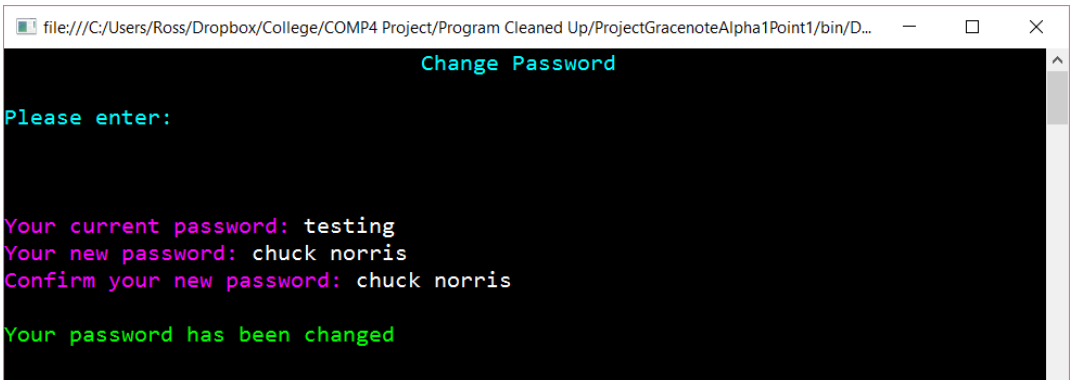


Figure 5.61 *– the green message is shown for 1 second before you're returned to the main menu (**Figure 5.58**)*

Here is the change that happens in the database:

PK_AccountID	Username	Password
4	test	z4DNiu1ILV0VJ9fccvzv+E5jJlkoSER9LcCw6H38mpA=

Figure 5.62 *– Accounts table before change*

PK_AccountID	Username	Password
4	test	CfVzxNNBzLgrs8ni85ED/sZsUomXXpwNQjE6loTHNdA=

Figure 5.63 *– Accounts table after change*

Testing

1 Introduction and Test Strategy

In the **Design Section 6.2 (page 74)**, I made an overall test strategy, outlining different conditions to test. This section of the project will be devoted to carrying out those tests in detail.

For each test series I will create multiple sub-tests in order to test the program using normal, erroneous, boundary and extreme data. I originally stated in the design that I would use mostly white box testing, but I have decided that this is not necessary. Therefore, I will be using black box testing for all of the series.

Here is the overall test strategy again, but with detail as to what data or results to test for. I have modified them slightly to make them more suited to our test plan.

Series	Condition to Test	Results to Test
1	The user is able to successfully navigate the program using all forms.	The correct forms are displayed after the successive key presses, and the user can log in.
2	A new account can be created.	The database is correctly updated.
3	An arrangement can be created and loaded correctly.	Check if the note and clip objects have been instantiated and set appropriate values.
4	A clip can be created correctly. Clips are displayed properly in the arranger.	We check that the notes list in the clip object is populated with note objects with appropriate properties. Multiple clips should be displayed in the right order. The notes are displayed correctly on the screen according to how they were set in the previous test.
5	Navigation using the keyboard within the arranger works correctly.	We should be able to correctly set the different clips.
6	A clip can be loaded, saved and deleted successfully.	We check that the serialised code the clip within the program.
7	An arrangement can be saved successfully.	We check that the serialised code in the database matches the arrangement's properties and clips within the program.
8	The user can change their password.	We check that the password has changed in the database.

2 Test Plan

Here, I outline the tests and the data to use. Each table is separated according to the series it belongs to.

The plan will outline the results that I expect to happen from performing certain tasks or using certain data.

Each test is accompanied by a test run ID, which cross-references with the test runs in **Section 3 (page 110)**.

If a test failed, its actual outcome is displayed and discussed in the test run.

2.1 Series 1

Test ID	Description	Data Category	Expected Outcome	Action	Test Run ID
1.1	Go from welcome to login screen	Normal	Login screen shown	PASS	TR1
1.2	Log in to an existing account (I will use my test account)	Normal	Main menu shown	PASS	TR2
1.3	Use incorrect username	Erroneous	Username doesn't exist error	PASS	TR3
1.4	Use incorrect password	Erroneous	Password incorrect error	PASS	TR4
1.5	Go from main menu to create arrangement wizard	Normal	Create arrangement wizard shown	PASS	TR5
1.6	Go from main menu to load arrangement wizard	Normal	Load arrangement wizard shown	PASS	TR6
1.7	Go from main menu to change password wizard	Normal	Change password wizard shown	PASS	TR7
1.8	Go from main menu to help screen	Normal	Help screen shown	PASS	TR8
1.9	Log out from main menu	Normal	Welcome screen shown	PASS	TR9
1.10	Make invalid choice on main menu	Exceptional	Invalid choice error	PASS	TR10

2.2 Series 2

Test ID	Description	Data Category	Expected Outcome	Action	Test Run ID
2.1	Create new account with valid details	Normal	New record in database with new account ID Username: test2 Password: (hashed value)	PASS	TR11
2.2	Leave the username field blank	Exceptional	Username blank error	PASS	TR12
2.3	Leave password blank	Exceptional	Password blank error	PASS	TR13
2.4	Leave confirm password blank	Exceptional	Confirm password blank error	PASS	TR14
2.5	Confirm password is not the same as password	Erroneous	Passwords mismatch error	PASS	TR15

2.3 Series 3

Test ID	Description	Data Category	Expected Outcome	Action	Test Run ID
3.1	Create an	Normal	Arranger screen shown	PASS	TR16

	arrangement with valid data and see if the database is updated		New record in database with name Test Arr 1 and ArrangementData set to 0031C24, and account ID set to whatever ID our test account is		
3.2	Leave name blank	Exceptional	Name blank error	PASS	TR17
3.3	Leave tonic note blank	Exceptional	Tonic note blank error	PASS	TR18
3.4	Leave major/minor blank	Exceptional	Major/minor blank error	PASS	TR19
3.5	Leave time signature blank	Exceptional	Time signature blank error	PASS	TR20
3.6	Use incorrect note	Erroneous	Invalid note error	PASS	TR21
3.7	Use invalid major/minor	Erroneous	Invalid major/minor error	PASS	TR22
3.8	Use text for time signature	Exceptional	Invalid time signature error	PASS	TR23
3.9	Use negative number for time signature	Extreme	Invalid time signature error	FAIL	TR24

2.4 Series 4

Test ID	Description	Data Category	Expected Outcome	Action	Test Run ID
4.1	Create clip with notes in the same octave and all notes are natural	Normal	Notes should be displayed. Each note should be in octave 4 and be all natural (appearing blue on the screen)	PASS	TR25
4.2	Create clip with notes in the same octave with natural and sharp notes	Normal	Notes should be displayed. Each note should be in octave 4 and there should be natural and sharp (appearing as green) notes	PASS	TR26
4.3	Create clip with notes in the same octave with natural and flat notes	Normal	Notes should be displayed. Each note should be in octave 4 and there should be natural and flat (appearing as red) notes	PASS	TR27
4.4	Create clip with notes in different octaves	Normal	Notes should be displayed of octaves in the range 3 to 5	PASS	TR28
4.5	Enter text into range of octaves	Exceptional	Invalid range of octaves error	PASS	TR29
4.6	Leave range of octaves blank	Exceptional	Range of octaves blank error	PASS	TR30
4.7	Enter negative number for range of octaves	Extreme	Invalid range of octaves error	FAIL	TR31
4.8	Enter a value greater	Extreme	Invalid range of octaves	PASS	TR32

	than the Int32 upper limit for range of octaves		error		
4.9	Leave range of notes blank	Exceptional	Range of notes blank error	PASS	TR33
4.10	Test upper limit of range of notes	Boundary	Invalid range of notes error	PASS	TR34
4.11	Test lower limit of range of notes	Boundary	Invalid range of notes error	PASS	TR35
4.12	Enter text into range of notes	Exceptional	Invalid range of notes error	PASS	TR36
4.13	Leave syncopation randomisation blank	Exceptional	Sync. rand. blank error	PASS	TR37
4.14	Test lower limit of sync. rand.	Boundary	Invalid sync. rand. error	PASS	TR38
4.15	Test upper limit of sync. rand.	Boundary	Invalid sync. rand. error	PASS	TR39
4.16	Enter text into sync. rand.	Exceptional	Invalid sync. rand. error	PASS	TR40
4.17	One clip is displayed correctly	Normal	The notes displayed in the clip viewer should match the note objects stored in the list	PASS	TR41
4.18	Multiple clips are displayed in the right order	Normal	The clip with notes in 1 octave should be displayed first. The clip with notes in 3 octaves should be displayed second	PASS	TR42

2.5 Series 5

Test ID	Description	Data Category	Expected Outcome	Action	Test Run ID
5.1	Check we can move from the next clip to the previous clip	Normal	The first clip should change to dark red. The second clip should change to bright red	PASS	TR43
5.2	Check we can move from the next clip to the previous clip	Normal	The second clip should change to dark red. The first clip should change to bright red	PASS	TR44
5.3	Check we can't select before the first clip	Boundary	Nothing should happen	PASS	TR45
5.4	Check we can't select after the last clip	Boundary	Nothing should happen	PASS	TR46
5.5	Check we can insert a new clip	Normal	Insert clip wizard should show	PASS	TR47

5.6	Check we can load a clip	Normal	Load clip wizard should show	PASS	TR48
5.7	Check we can save a clip	Normal	Save clip wizard should show	PASS	TR49
5.8	Check we can delete a clip	Normal	Clip should disappear	PASS	TR50
5.9	Check we can view a clip	Normal	Clip viewer should show	PASS	TR51
5.10	Check we can save the arrangement	Normal	'Arrangement saved' message should show	PASS	TR52
5.11	Check we can exit the arrangement	Normal	Main menu should show	PASS	TR53
5.12	Test invalid key press	Exceptional	Invalid option error	FAIL	TR54

2.6 Series 6

Test ID	Description	Data Category	Expected Outcome	Action	Test Run ID
6.1	Check a clip can be saved	Normal	A new record should be created in the Clips table with name Test Clip 1 and note data filled. A record in ClipUses to link the clip ID with the arrangement ID should also be created	PASS	TR55
6.2	Save with no name	Exceptional	Name blank error	PASS	TR56

2.7 Series 7

Test ID	Description	Data Category	Expected Outcome	Action	Test Run ID
7.1	Save arrangement with no new clips	Normal	No changes should be made to the arrangement's record in the Arrangement's table	PASS	TR57
7.2	Save arrangement with 3 unsaved clips	Normal	The clip and note data is serialised in the ClipContentsData field in a new record in the Arrangements table	PASS	TR58
7.3	Save arrangement with 2 unsaved clips and 2 saved clips	Normal	The clip and note data should be filled in the Clips, ClipUses and Arrangements tables, with the ArrangementData field containing 2 references to unsaved clips and 2 references to saved clips	PASS	TR59

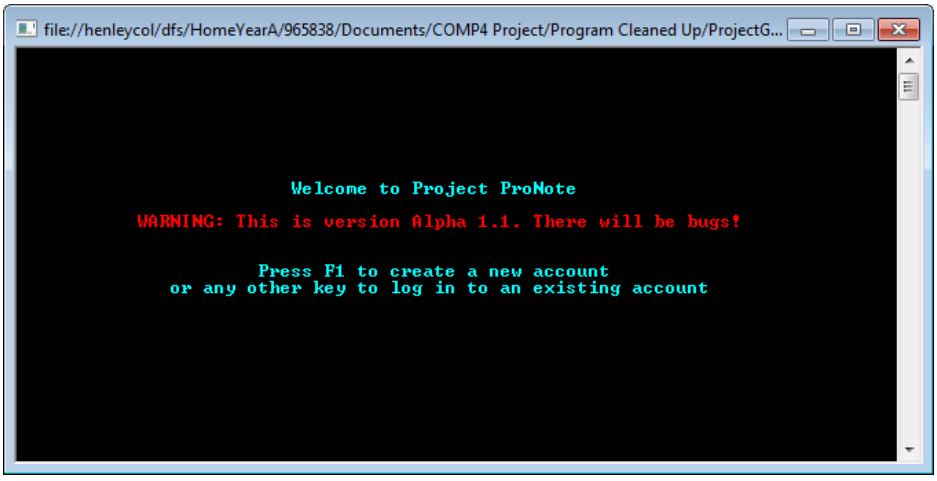
2.8 Series 8

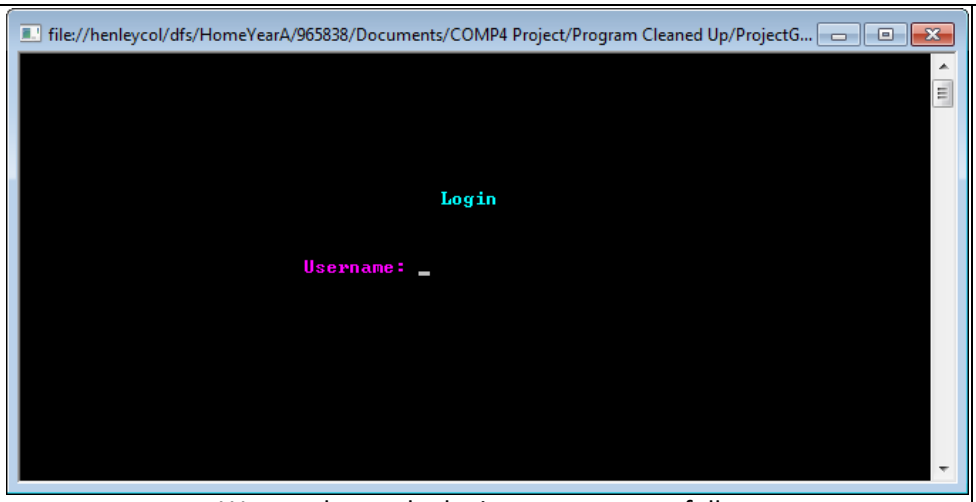
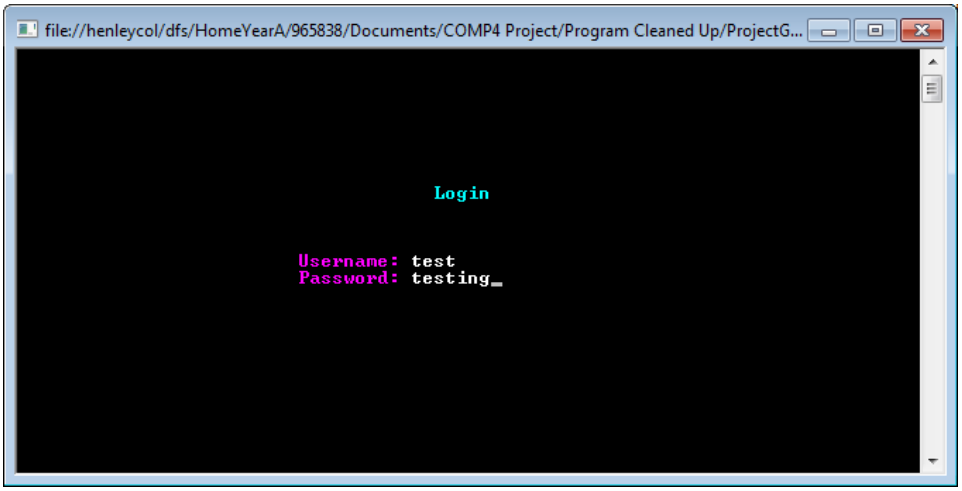
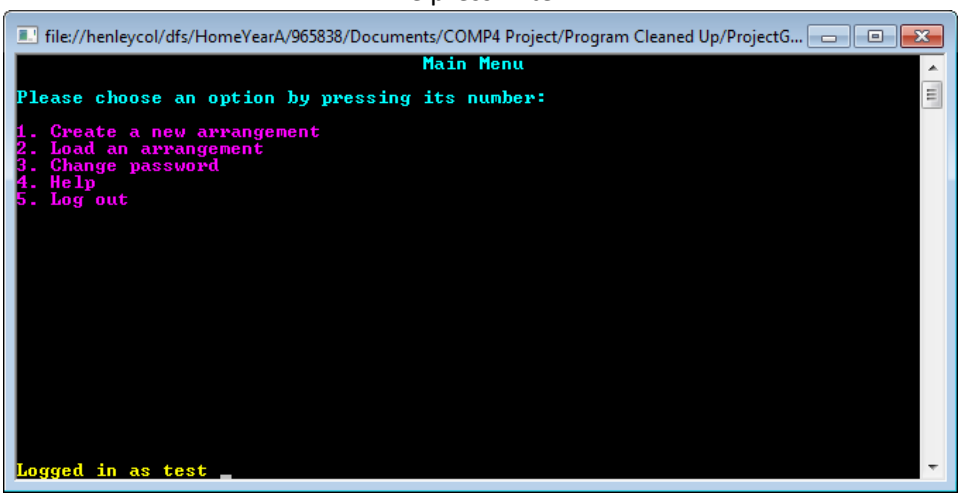
Test ID	Description	Data Category	Expected Outcome	Action	Test Run ID
8.1	Check if user can change password	Normal	Hash for password in Accounts table should change	PASS	TR60
8.2	Leave current password blank	Exceptional	Current password blank error	PASS	TR61
8.3	Leave new password blank	Exceptional	New password blank error	PASS	TR62
8.4	Leave confirm new password blank	Exceptional	Confirm new password blank error	PASS	TR63
8.5	Use the wrong current password	Erroneous	Incorrect current password error	PASS	TR64
8.6	Have the confirm password different to the first one	Erroneous	Password mismatch error	PASS	TR65

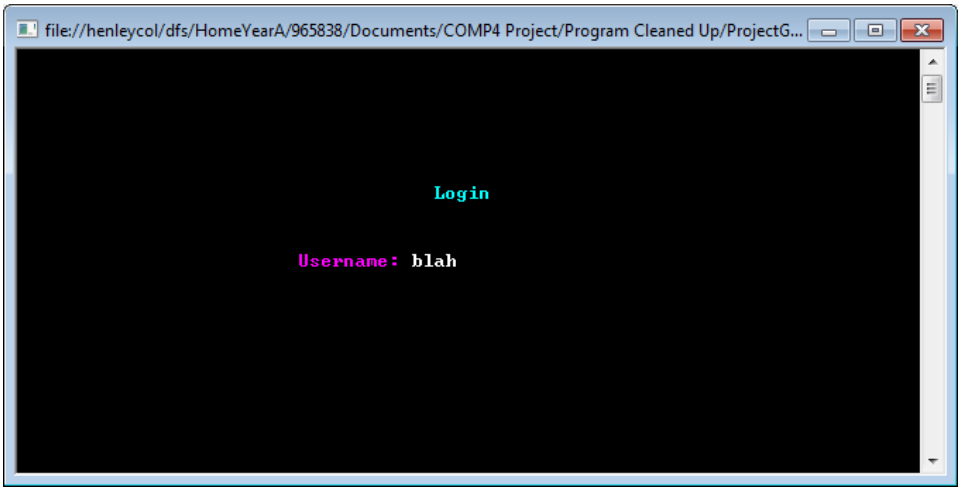
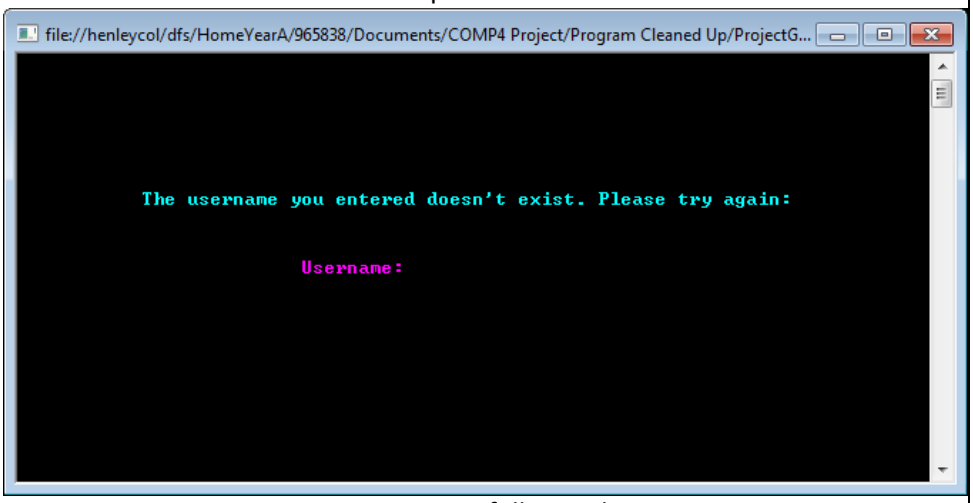
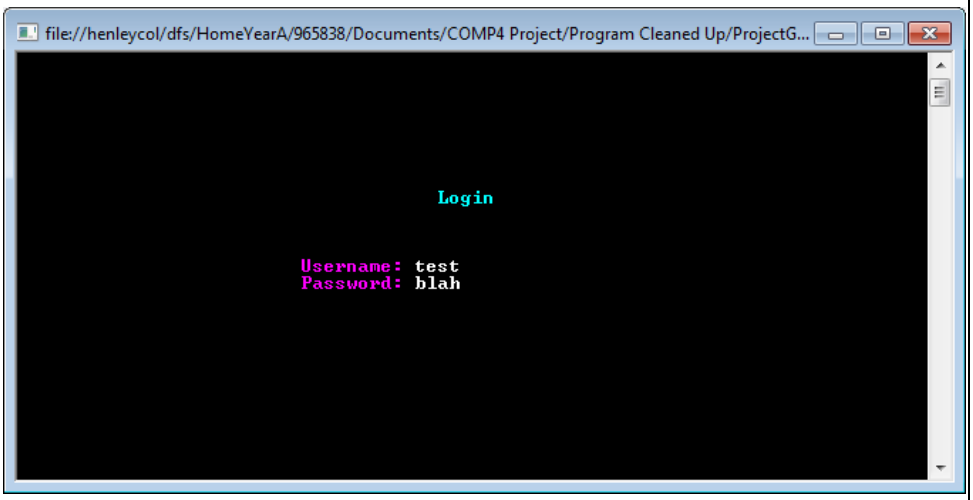
3 Test Runs

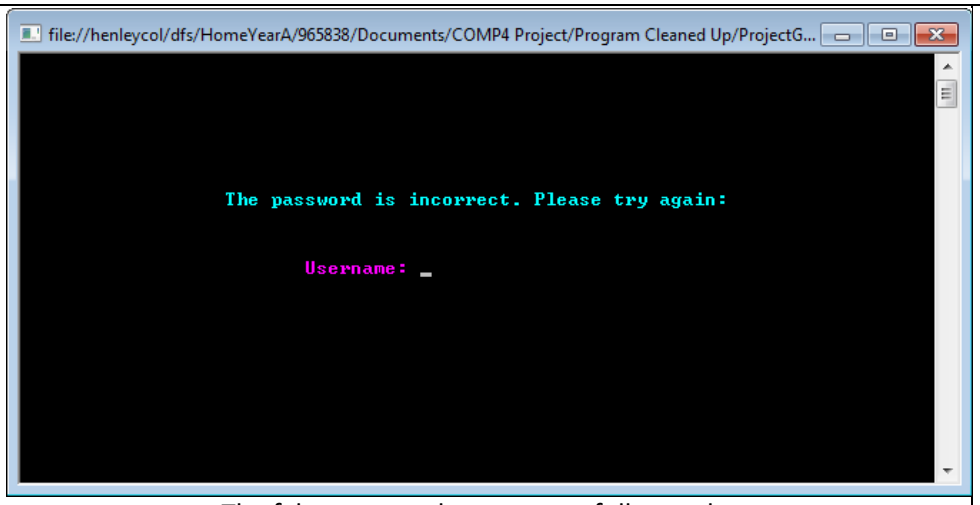
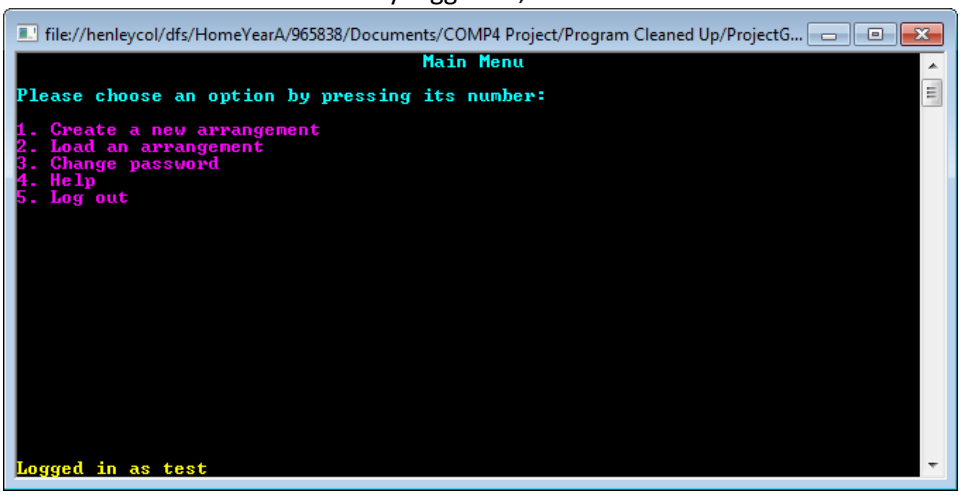
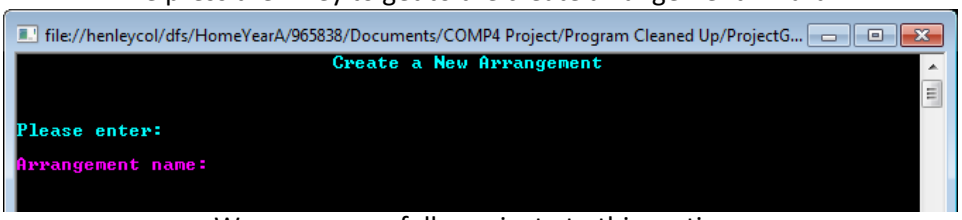
For most of the test runs, I have shown two screenshots. The first is when we are in the first stage of the test or we are entering some data, and the second is the result of that data. More complicated tests will have more screenshots and brief descriptions.

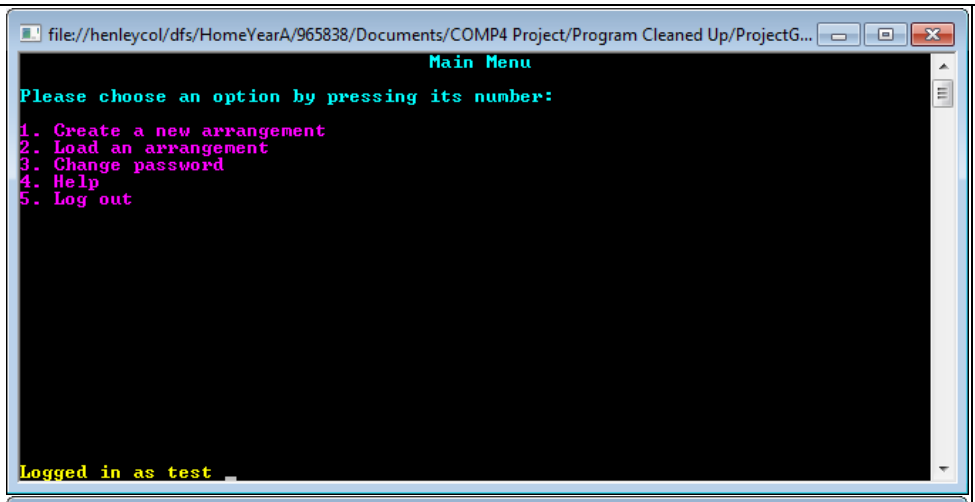
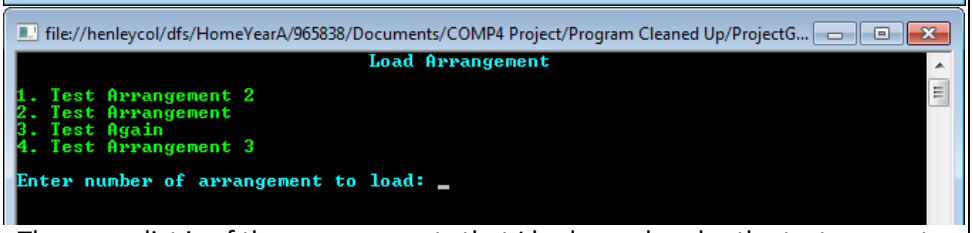
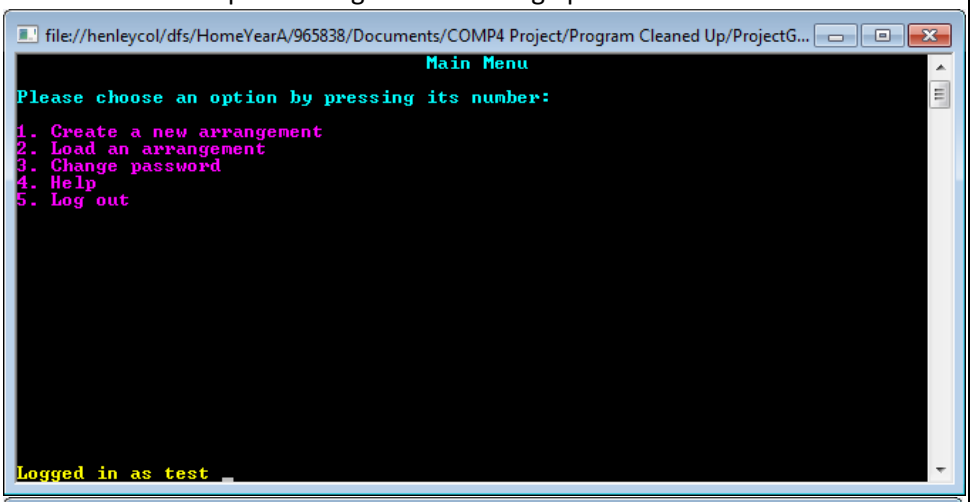
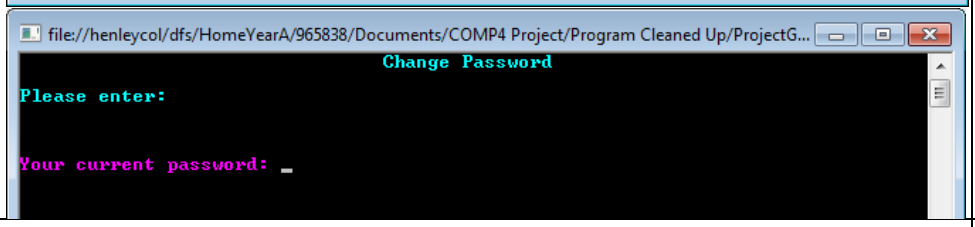
Please refer back to the test plan to see what key presses I used, if it's not described in the test run.

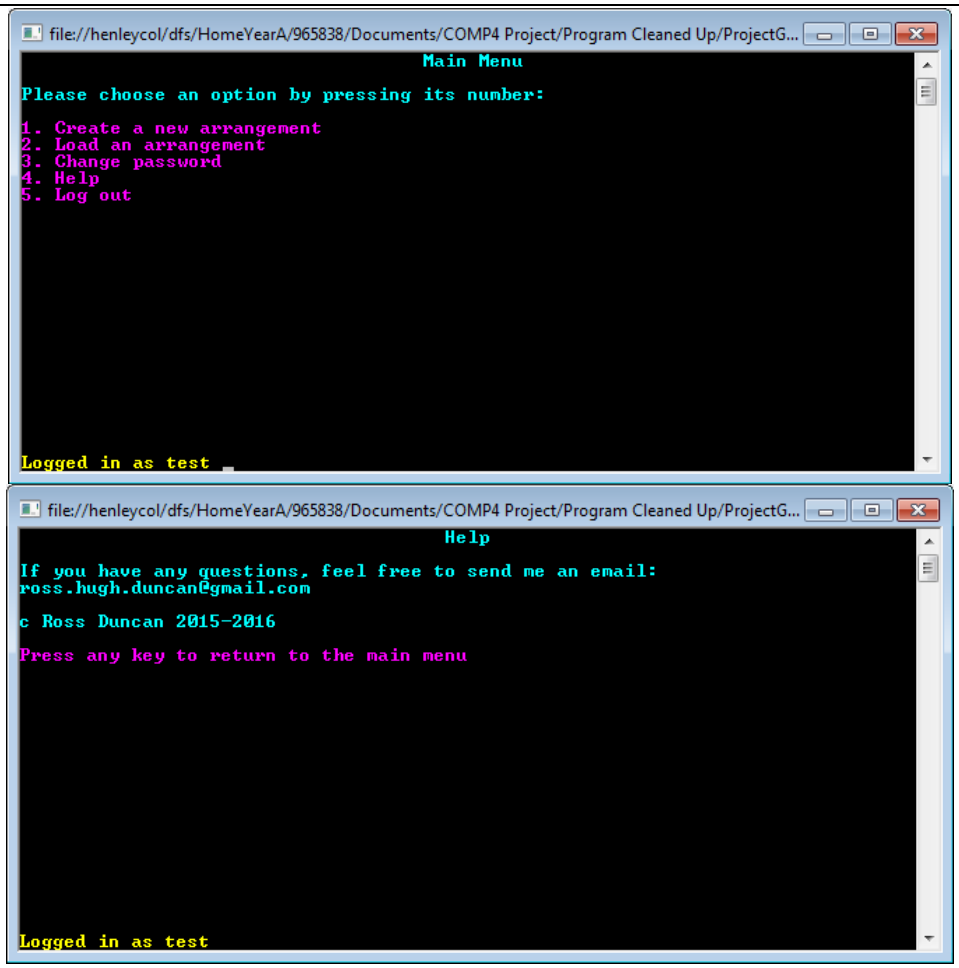
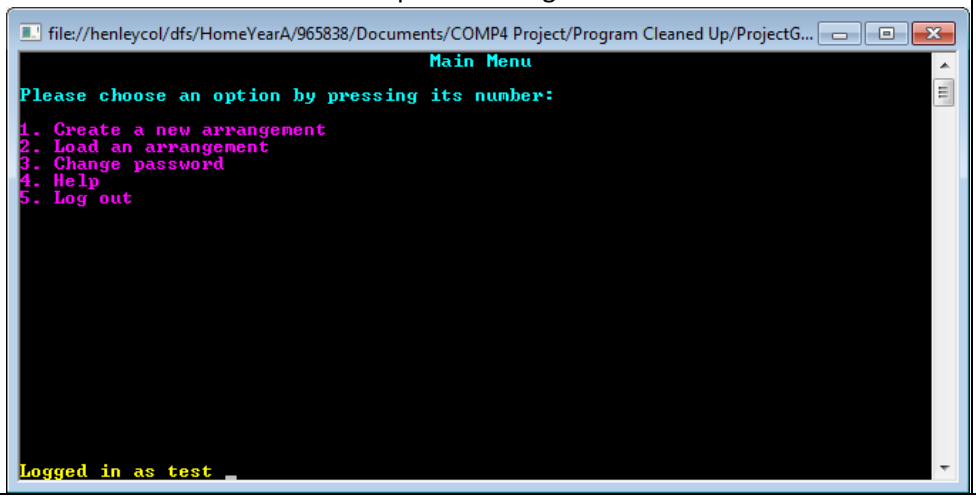
Test Run ID	Test Run	Test ID
TR1	The welcome screen is the first to be shown:  We press Enter:	1.1

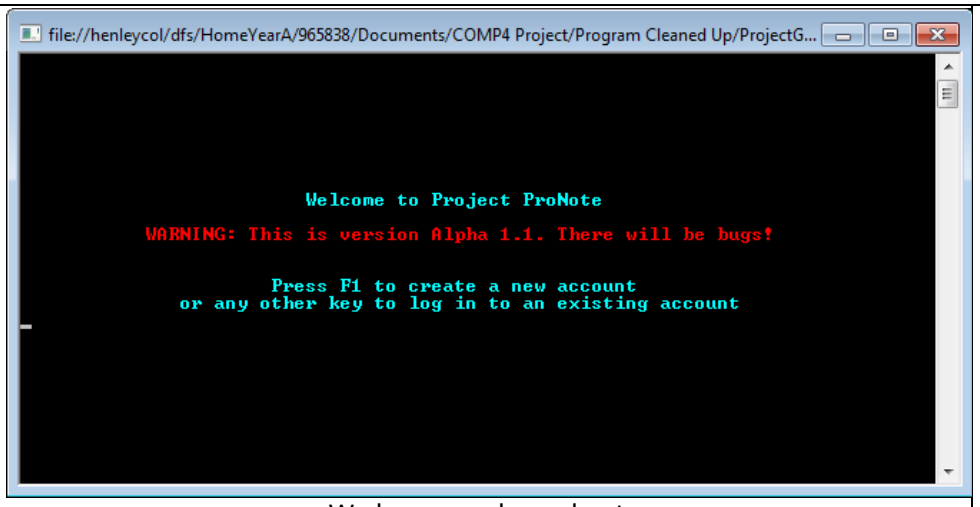
	 We are shown the login screen successfully.	
TR2	We press any key other than F1 on the welcome screen to get to the login screen, and we enter our details:  We press Enter:  We are now logged in.	1.2

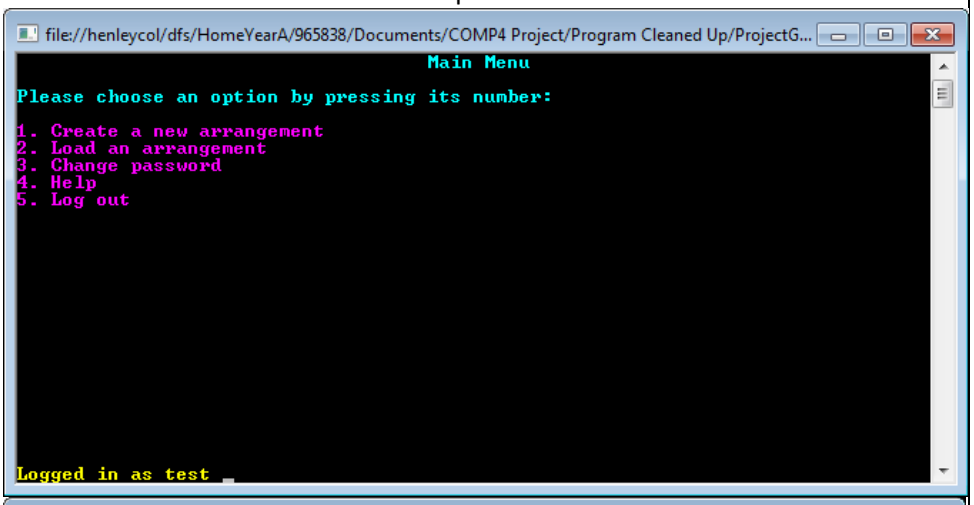
TR3	We enter our invalid details:  We press Enter:  It was successfully caught.	1.3
TR4	We enter our invalid data:  And press Enter:	1.4

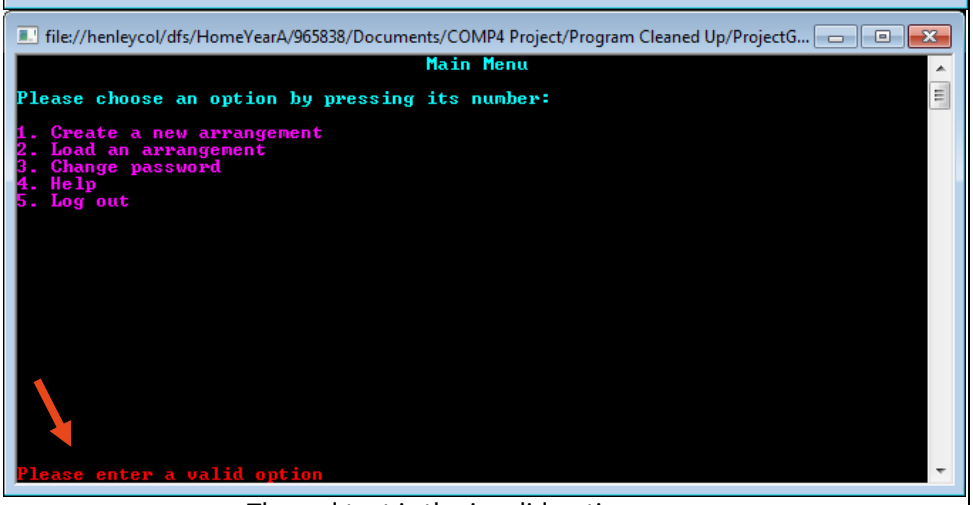
	 The fake password was successfully caught.	
TR5	Once we've successfully logged-in, we are at the main menu:  We press the 1 key to get to the create arrangement wizard:  We can successfully navigate to this section.	1.5
TR6	From the main menu, we press 2 to the load arrangement wizard:	1.6

	  The green list is of the arrangements that I had saved under the test account, which means we can successfully select.	
TR7	We press 3 to get to the change password wizard:  	1.7
TR8	We press 4 to show the help screen:	1.8

<table>
<tr>
<td></td>
<td> 
</td>
<td></td>
</tr>
<tr>
<td>TR9</td>
<td>We press 5 to log out:
 
</td>
<td>1.9</td>
</tr>
</table>

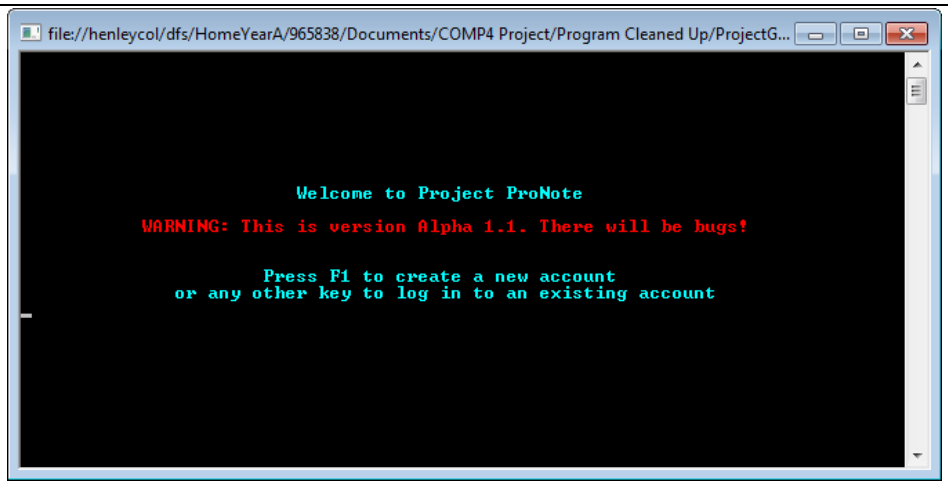
<table>
<tr><td></td><td> 

We have now logged out.</td><td></td></tr>
<tr><td>TR10</td><td>We press 6:
 

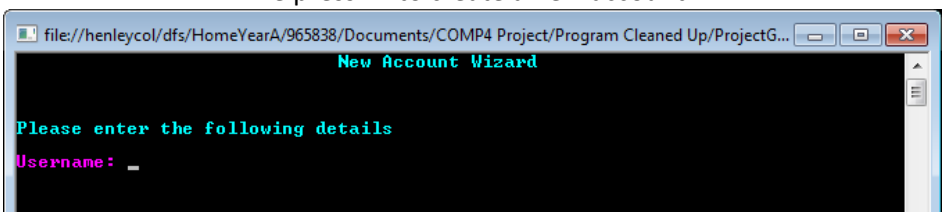


The red text is the invalid option error.</td><td>1.10</td></tr>
</table>

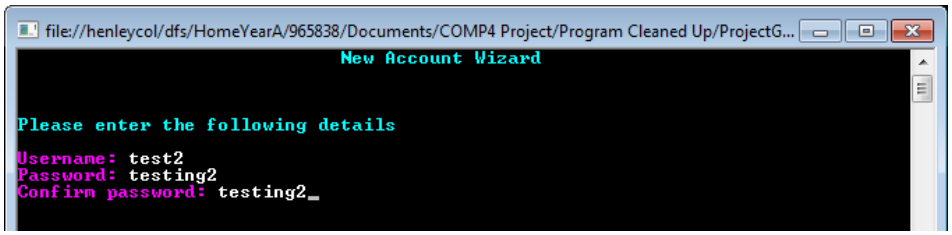
TR11



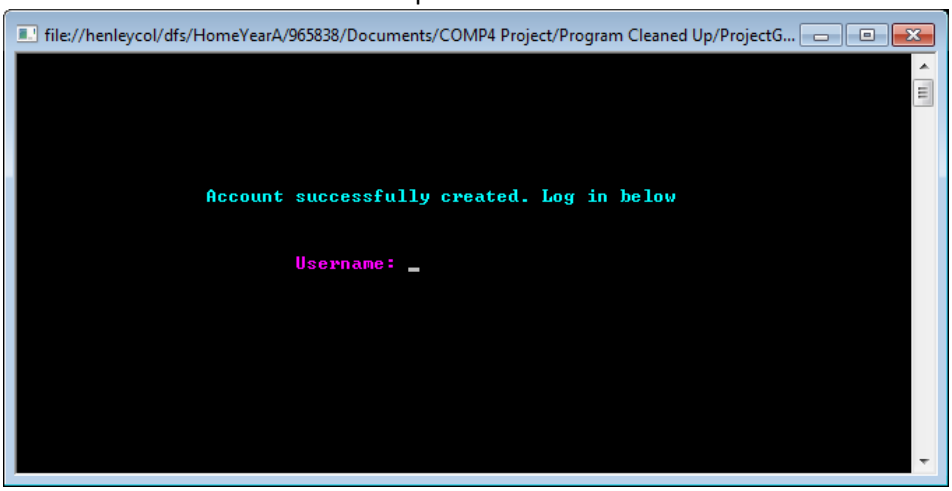
We press F1 to create a new account:



I entered the details:



And pressed Enter:

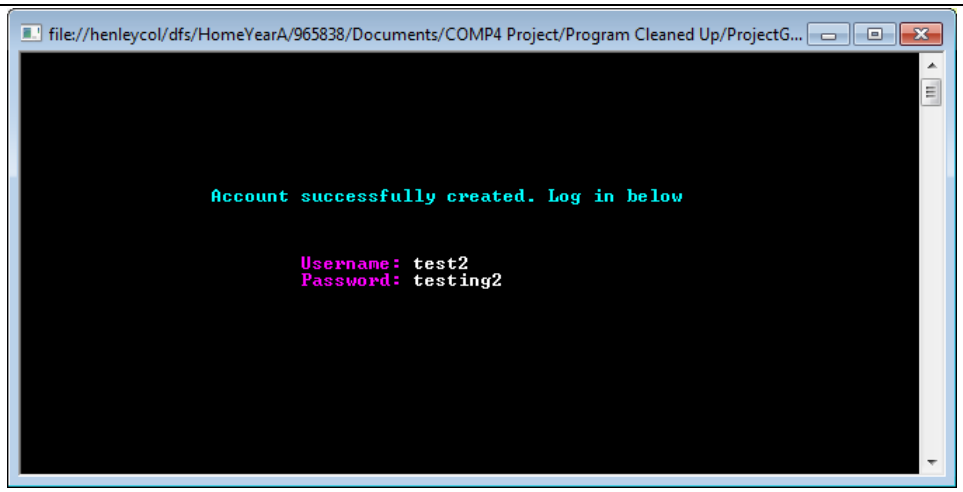
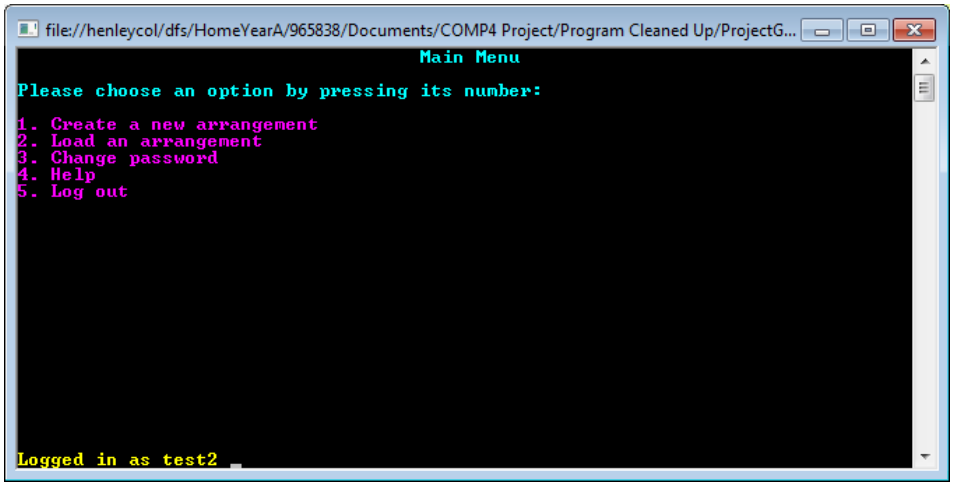
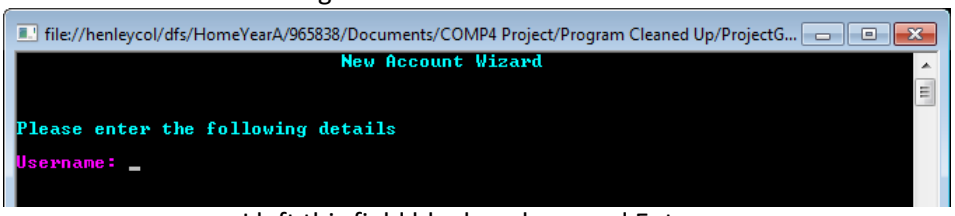
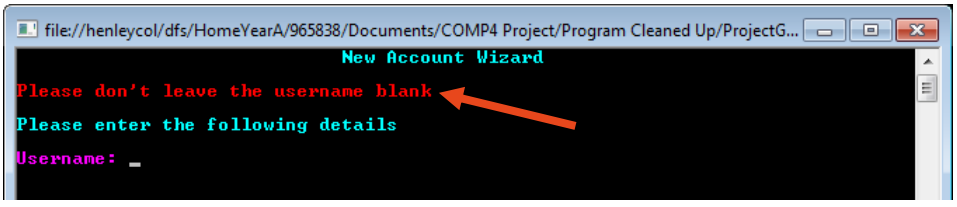
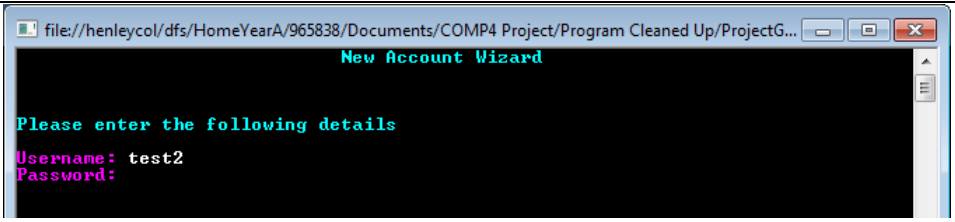


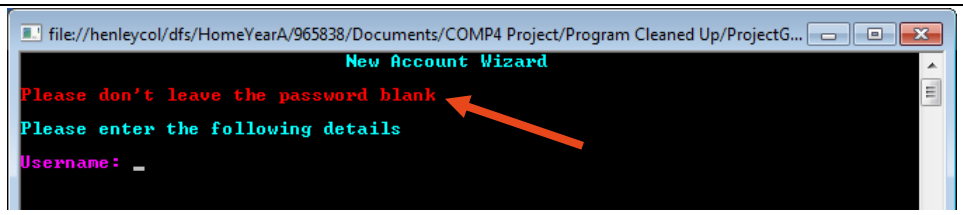
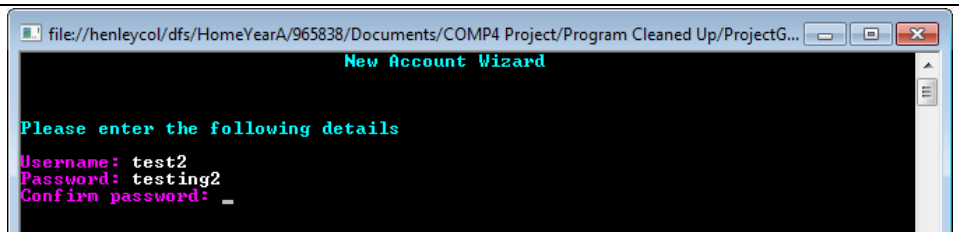
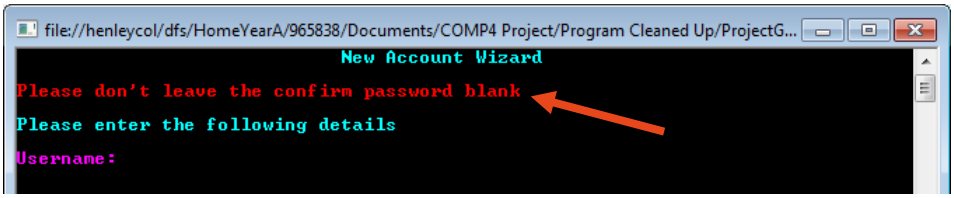
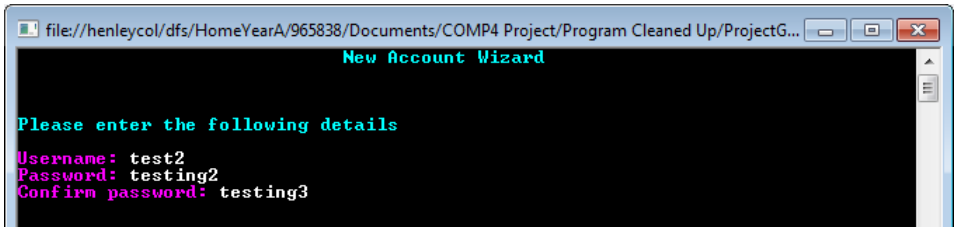
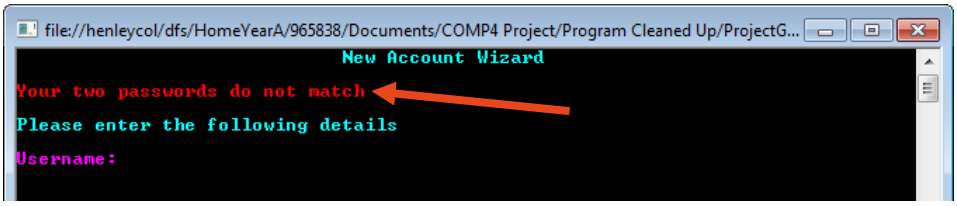
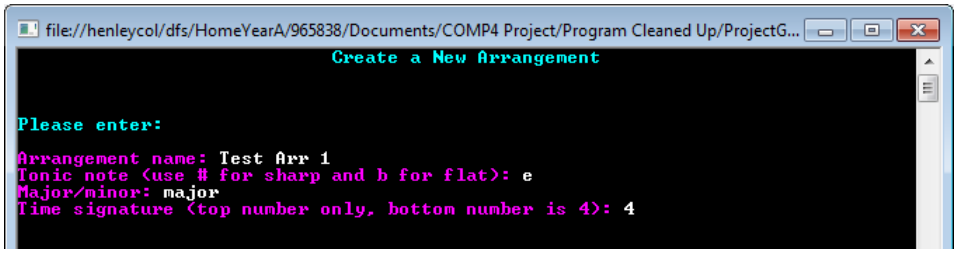
To check that the account was created, we check the Accounts table in the database:

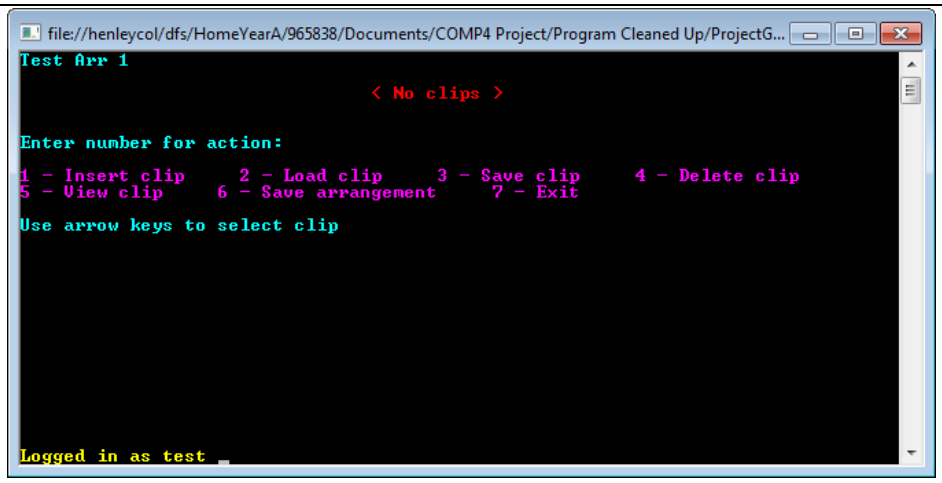
PK_AccountID	Username	Password
4	test	z4DNiu1ILV0VJ9fccvzv+E5jJlkoSER9LcCw6H38mpA=
5	test2	QxERRymTv02bizR0dreTIf6oozfzwcsv7aoYW1QYVUA=

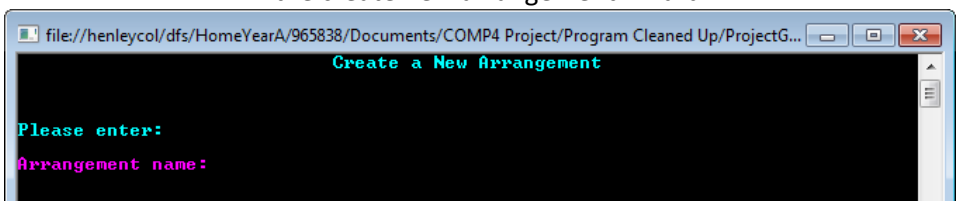
A new record is created. To check that the password is correctly hashed, we enter our login details:

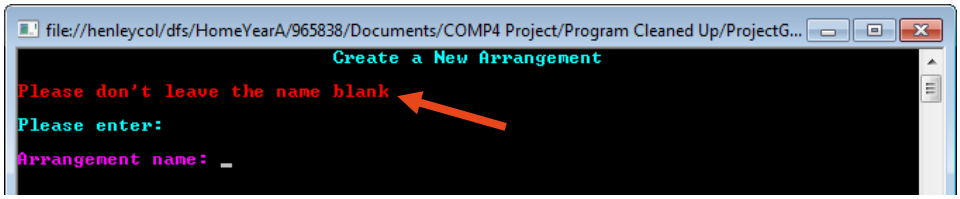
2.1

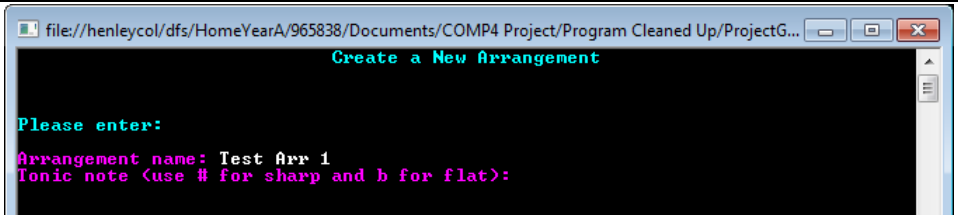
	 And press Enter:  We have successfully logged in.	
TR12	We go to the new account wizard:  I left this field blank and pressed Enter: 	2.2
TR13	 I left the password field blank and pressed Enter:	2.3

		
TR14	 I left the confirm password field blank and pressed Enter: 	2.4
TR15	I entered mismatching passwords:  And pressed Enter: 	2.5
TR16	In the create new arrangement wizard, I entered some details:  And pressed Enter:	3.1

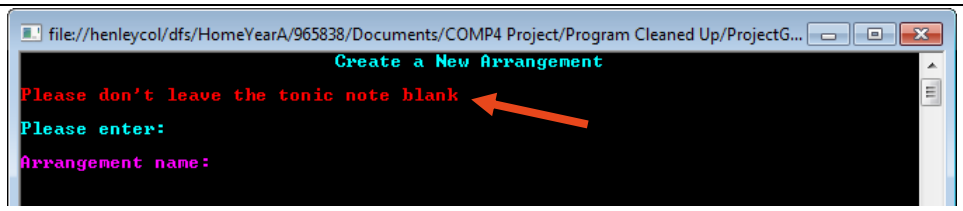
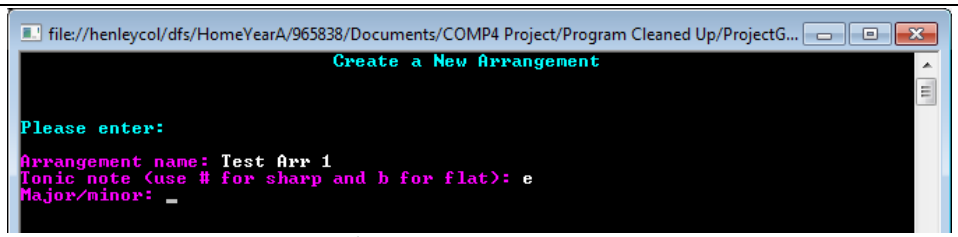
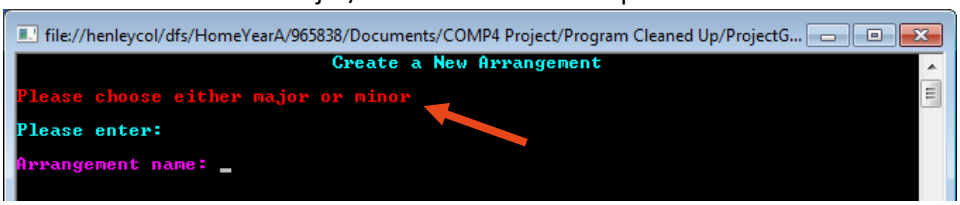
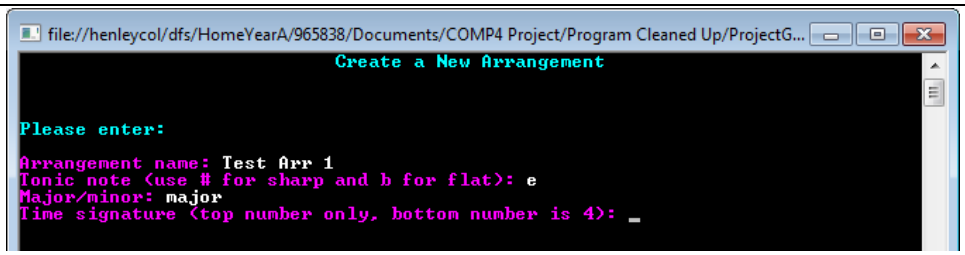
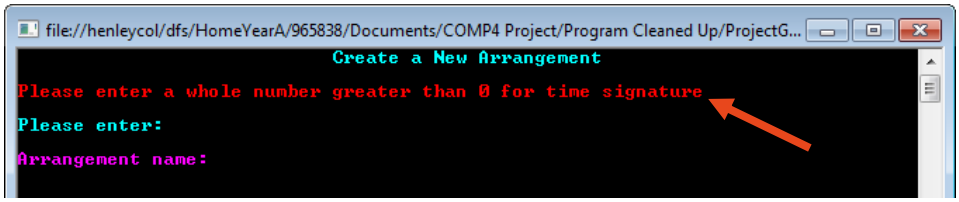
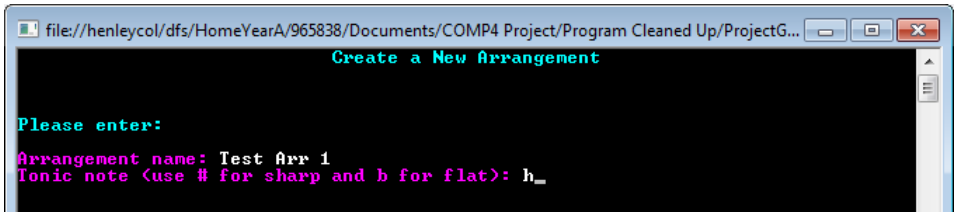
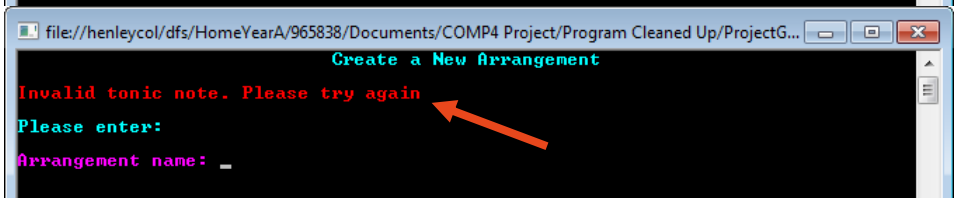
<table>
<tr>
<td></td>
<td>


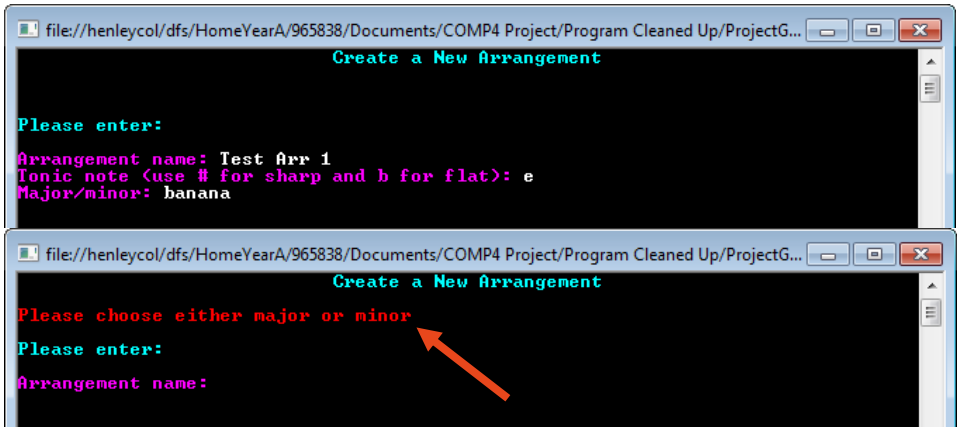
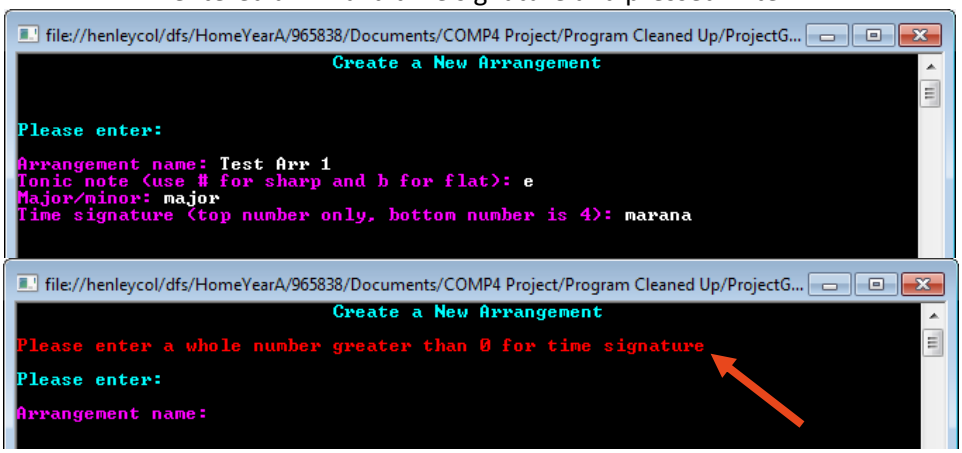
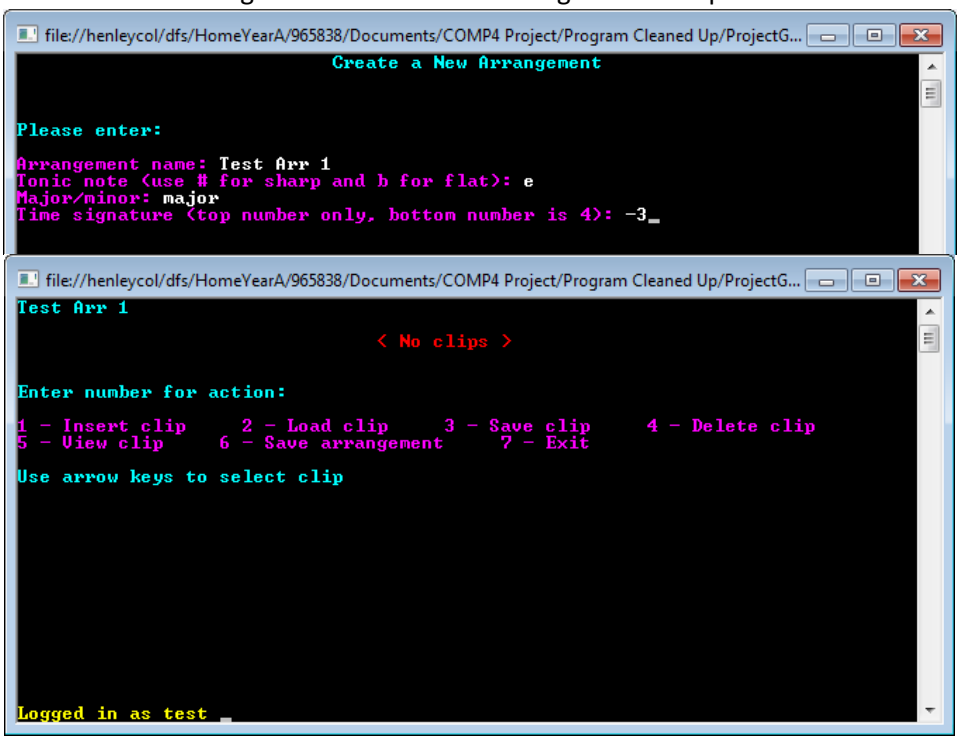
Our new arrangement has been displayed. To check that it has been saved in the database, we check the Arrangements table:
<table>
<tr><th>PK_ArrangementID</th><th>ArrangementName</th><th>ArrangementData</th><th>ClipContentsData</th><th>FK_AccountID</th></tr>
<tr><td>8</td><td>Test Arrangement</td><td>001011D21BE202717462FA08A68</td><td>E202717462FXA3B03C4YX/</td><td>4</td></tr>
<tr><td>9</td><td>Test Arrangement 2</td><td>0011C24A39A49BE304842317F</td><td>E304842317FXA1B02C4YX/</td><td>4</td></tr>
<tr><td>11</td><td>Test Arrangement 3</td><td>0031C24BE730148066FBE1367104</td><td>E730148066FXA0B15C4YX/</td><td>4</td></tr>
<tr><td>12</td><td>Test Again</td><td>0071C24</td><td>NULL</td><td>4</td></tr>
<tr><td>13</td><td>Test Arr 1</td><td>0031C24</td><td>NULL</td><td>4</td></tr>
</table>
The new record has been created with the account ID as 4.
We created the arrangement with the test account. Let's check the Accounts table to see what ID the test account is:
<table>
<tr><th>PK_AccountID</th><th>Username</th><th>Password</th></tr>
<tr><td>4</td><td>test</td><td>z4DNiu1ILV0VJ9fccvzv+E5jJlkoSER9LcCw6H38mpA=</td></tr>
<tr><td>5</td><td>test2</td><td>QxERRymTv02bizR0dreTlf6oozfzwcsv7aoYW1QYVUA=</td></tr>
</table>
It has an ID of 4, so this test has passed.
</td>
<td></td>
</tr>
<tr>
<td>TR17</td>
<td>
In the create new arrangement wizard:


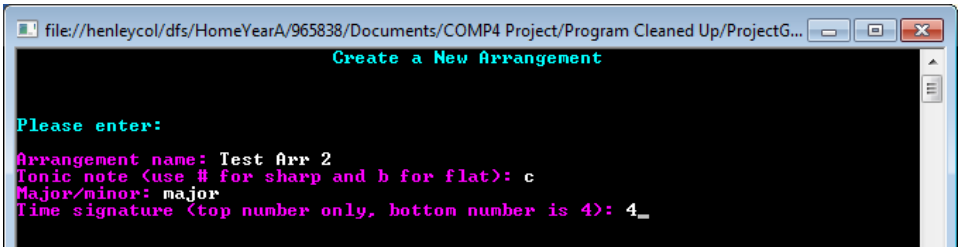
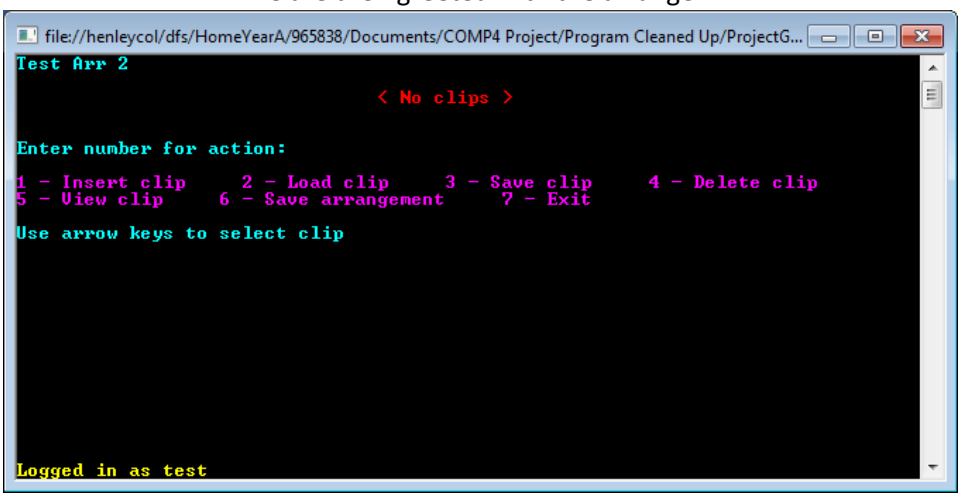
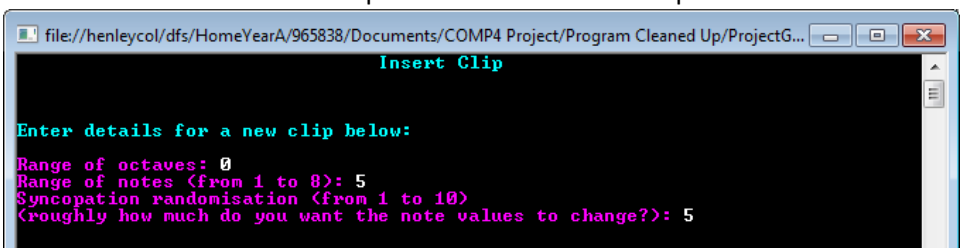
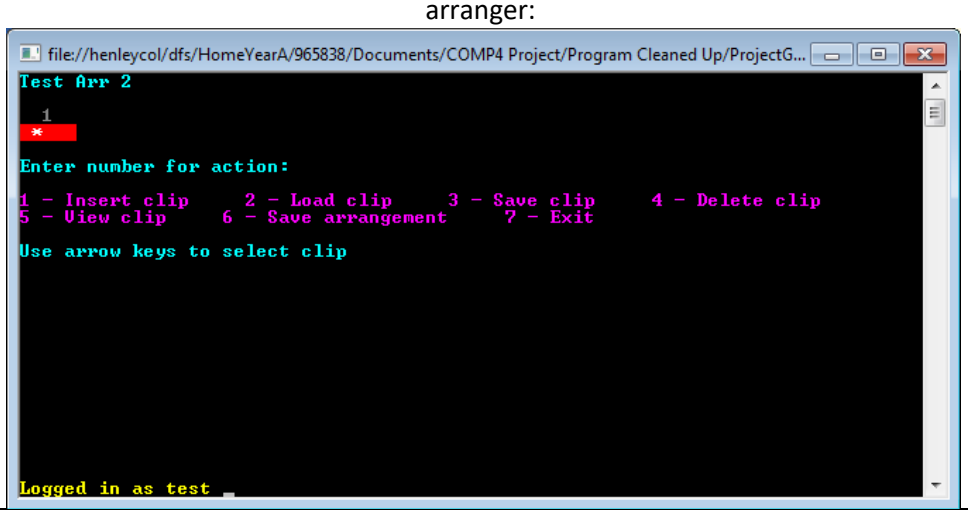
I left the name field blank and pressed Enter:


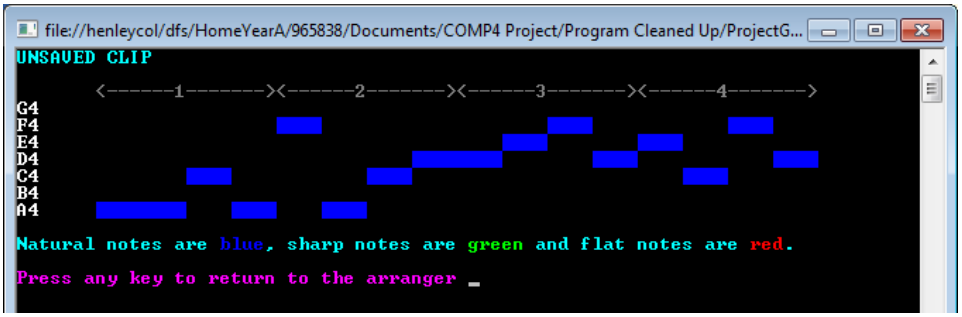
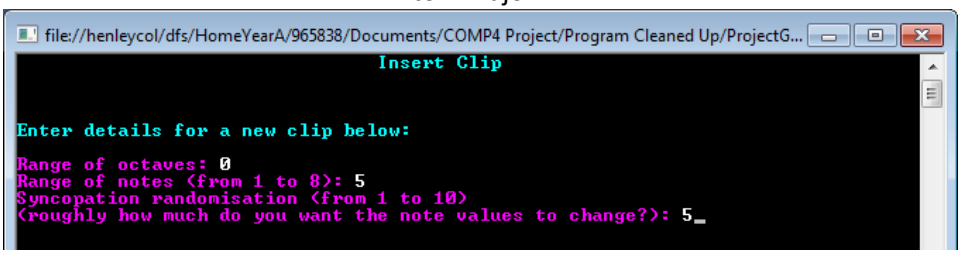
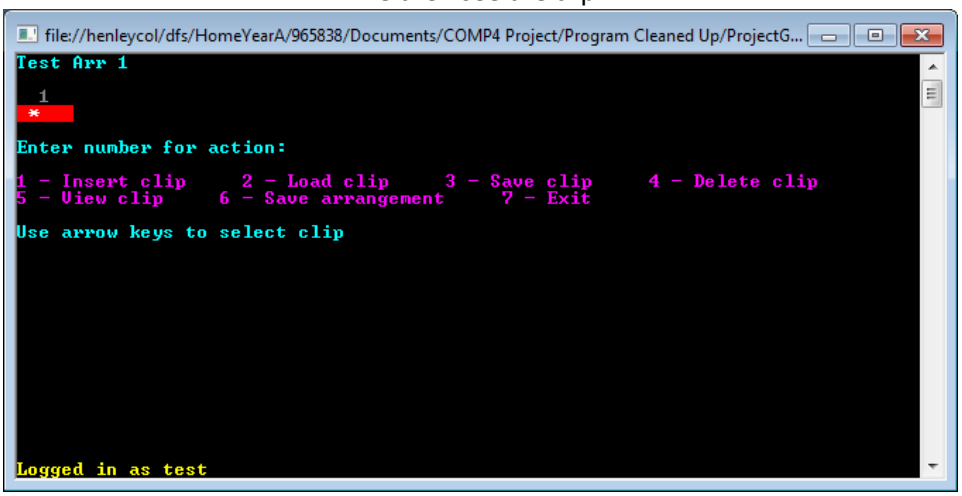
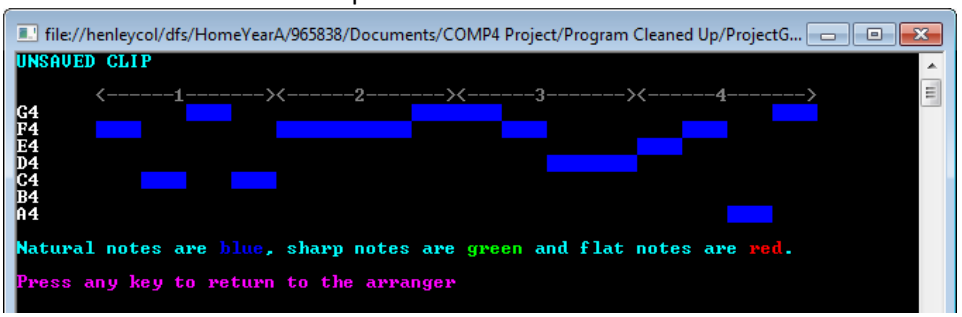
Our blank field was caught.
</td>
<td>3.2</td>
</tr>
<tr>
<td>TR18</td>
<td>


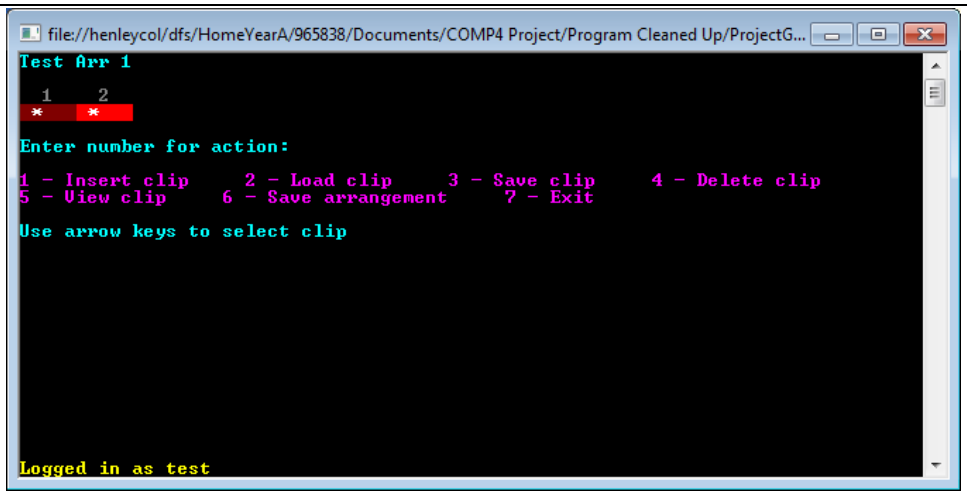
I left the tonic note blank and pressed Enter:
</td>
<td>3.3</td>
</tr>
</table>

		
TR19	 If left the major/minor field blank and pressed Enter: 	3.4
TR20	 I left the time signature blank and pressed Enter: 	3.5
TR21	I entered an invalid tonic note and pressed Enter:  	3.6

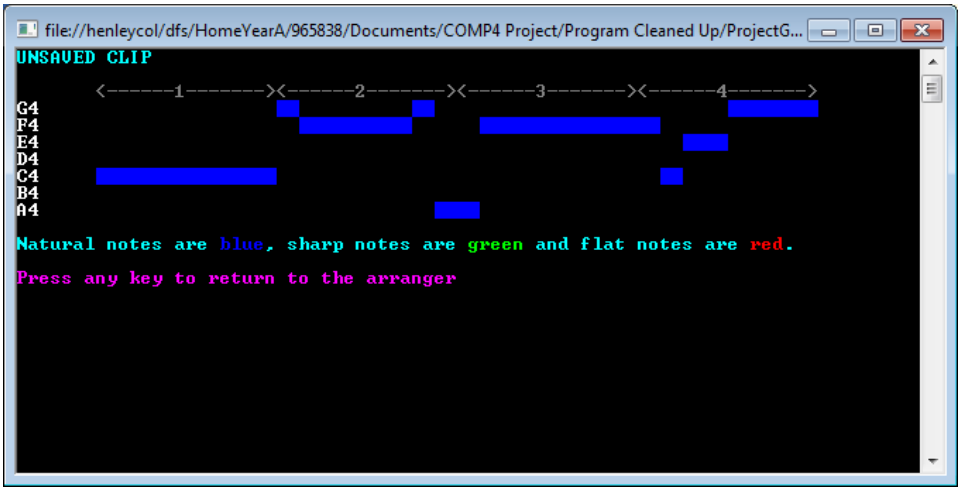
TR22	I entered an invalid major/minor and pressed Enter: 	3.7
TR23	I entered an invalid time signature and pressed Enter: 	3. 8
TR24	I entered a negative value for the time signature and pressed Enter:  The arranger is displayed, meaning that it has allowed a negative number for the range of octaves. This is not possible for an arrangement and it did not	3.9

	throw an error. Hence, this test failed. This could have been solved by using an if statement that checked if the input was greater than 0 before continuing.	
TR25	To start, we need to create an arrangement in C major, so we enter the following in the New Arrangement wizard:  We are then greeted with the arranger:  We then press 1 to create a new clip:  We insert the data for a new clip as above and press Enter to get back to the arranger:  	4.1

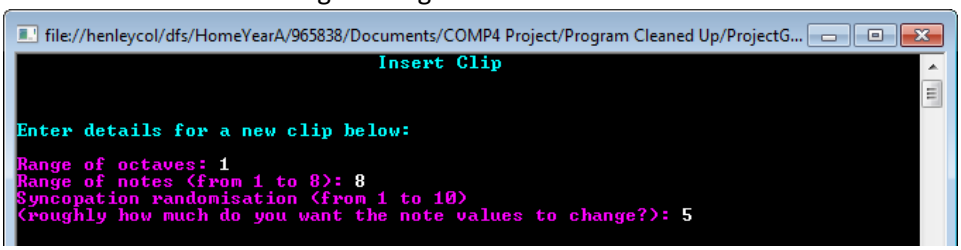
	You can see the clip, represented as the red block with the asterisk. Now, we press 5 to see what notes it contains:  All of the notes are natural and in octave 4, so this test has passed.	
TR26	For this test, we create a new clip in the Test Arr 1 arrangement, which was set to E major:  We then see the clip:  And press 5 to view the notes:  All of the notes are natural, which is wrong. We expect some sharp notes. This is no problem because a clip doesn't have to have sharp notes. So, we create another clip:	4.2



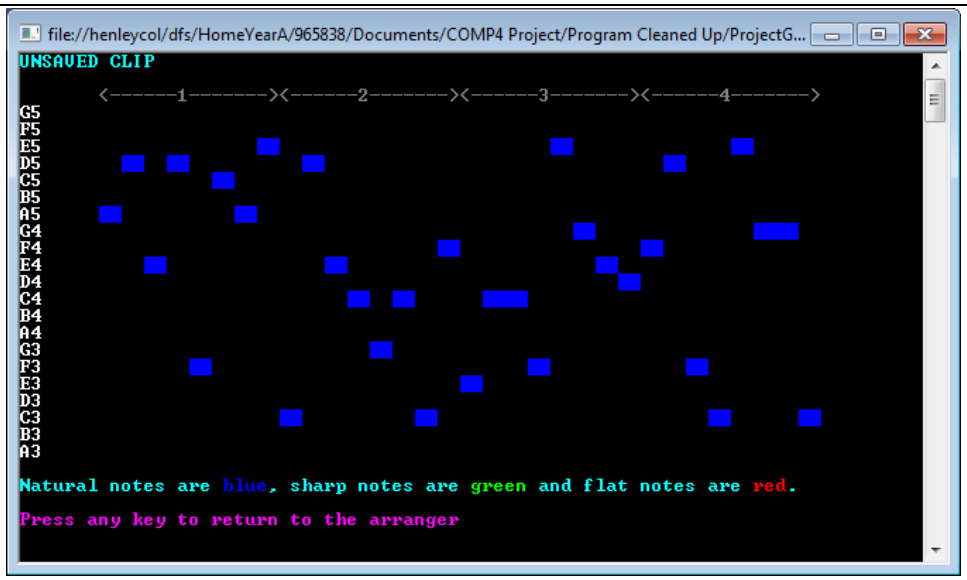
And try again:

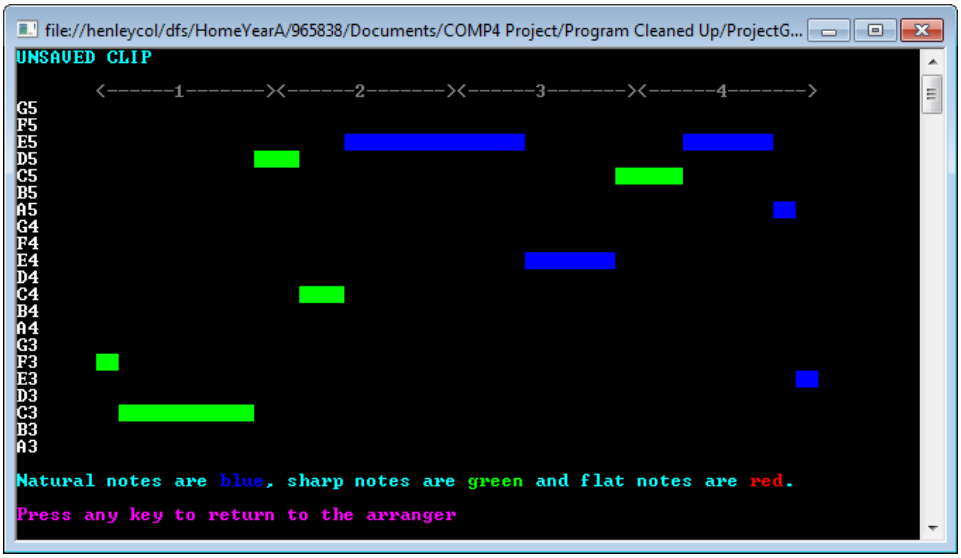


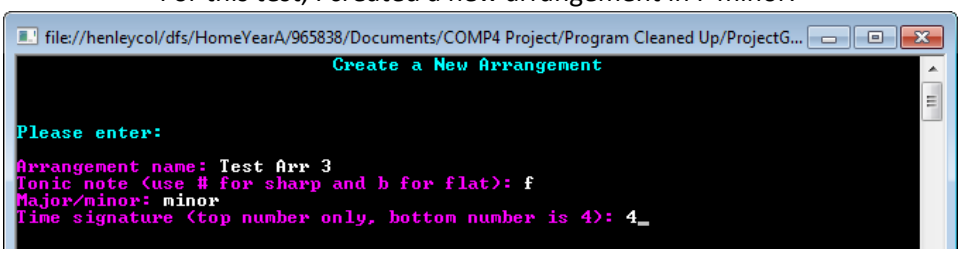
Again, we still expect green notes. Therefore, we can try creating another clip with higher ranges of octaves and notes:



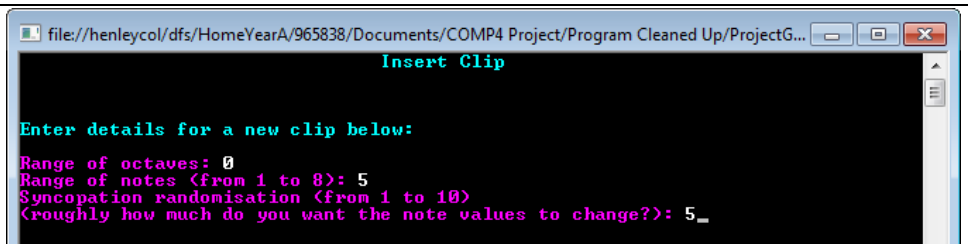
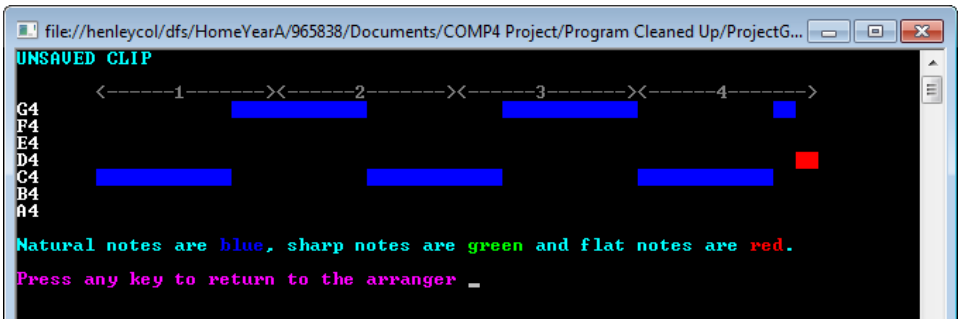
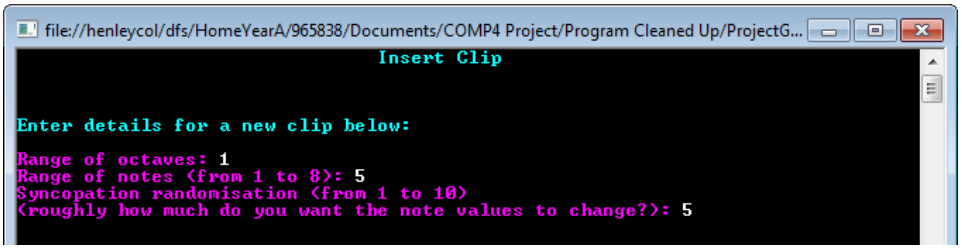
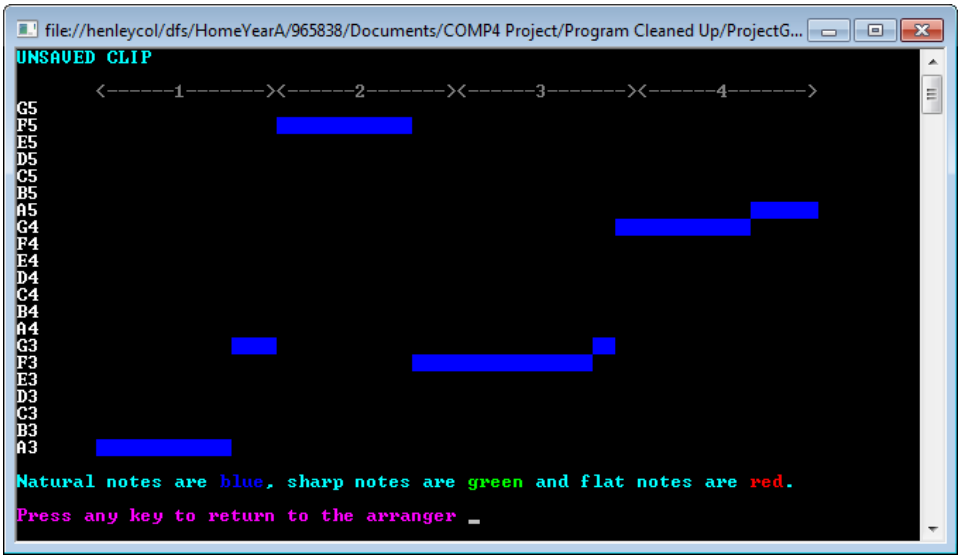
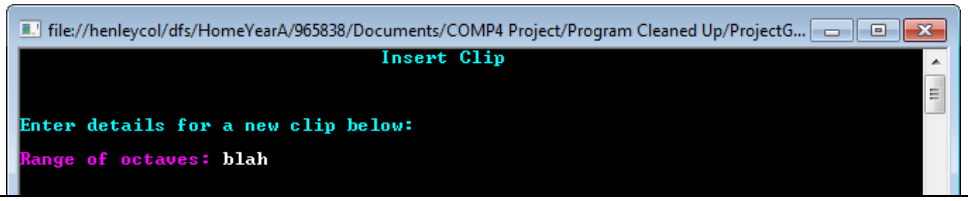
And viewed that:

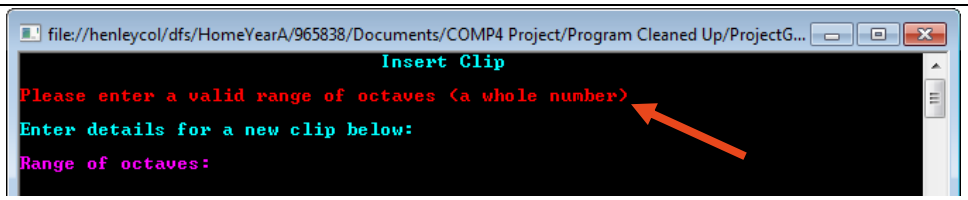
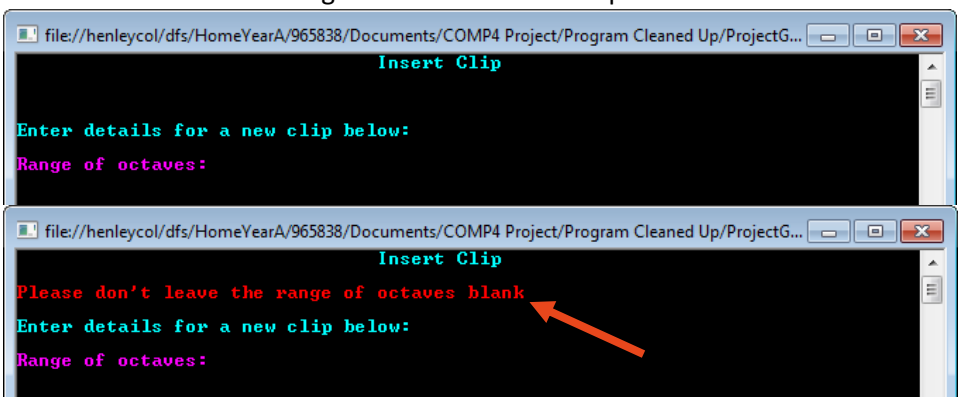
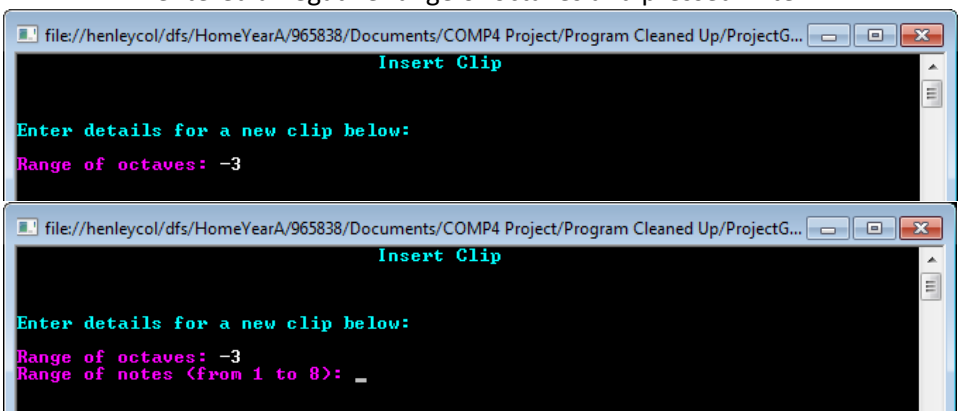
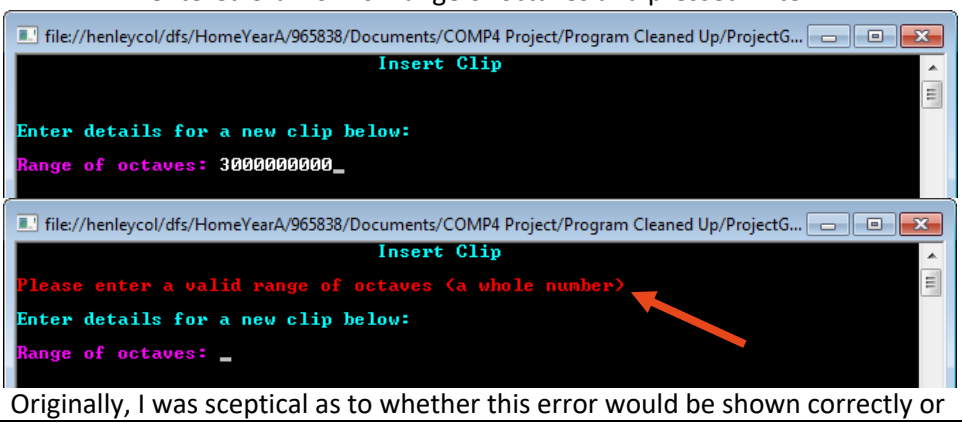
<table>
<tr>
<td></td>
<td>


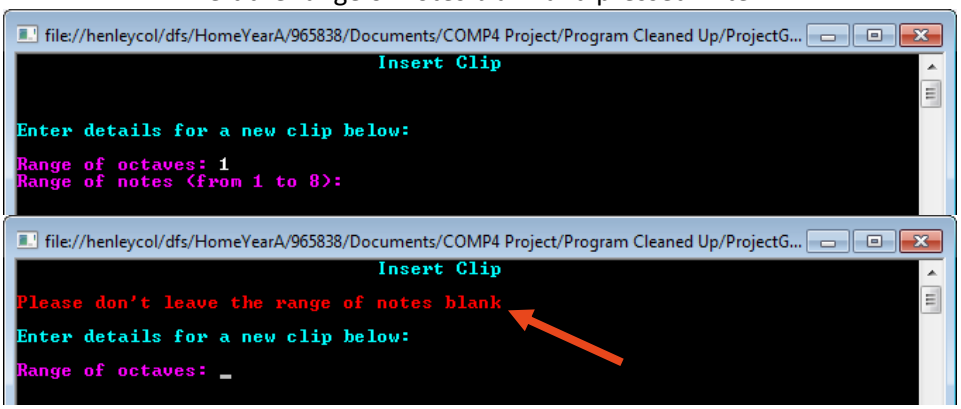
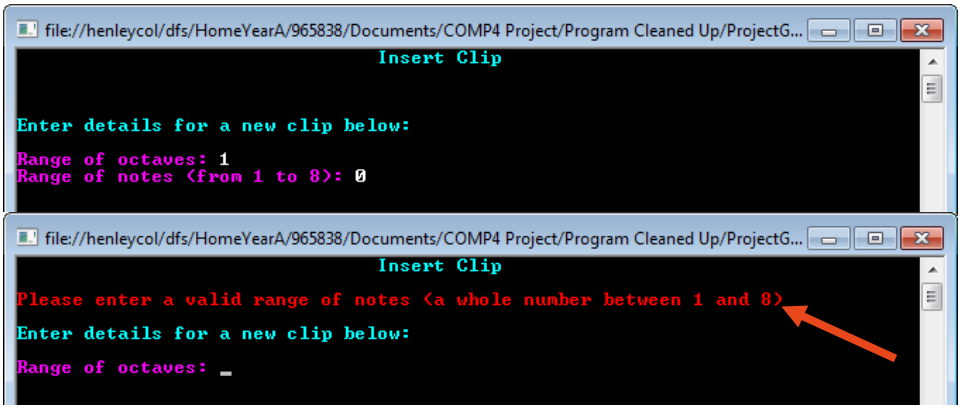
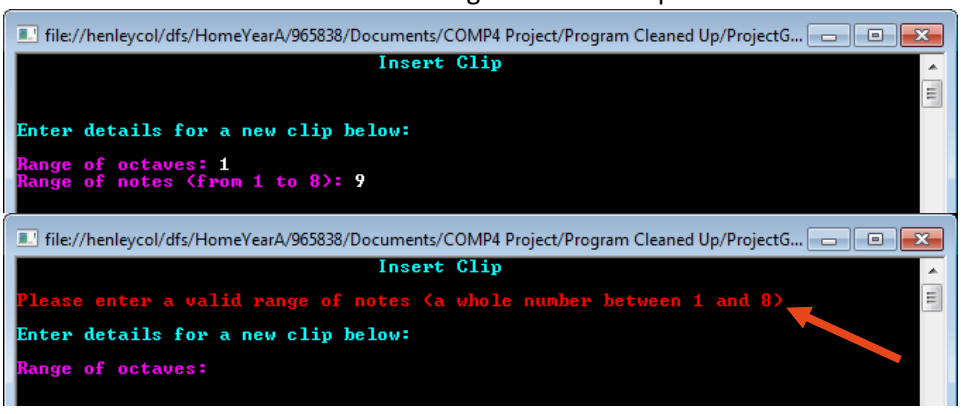
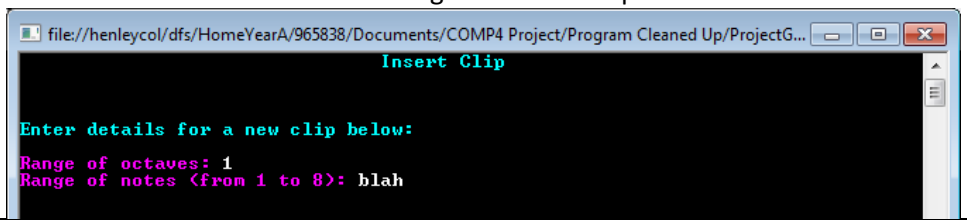
This is very strange, as the Test Arr 1 arrangement is set to E major and should produce sharp notes. As a last ditch, I exited the arrangement and tried loading it again, and creating the same clip as above again:


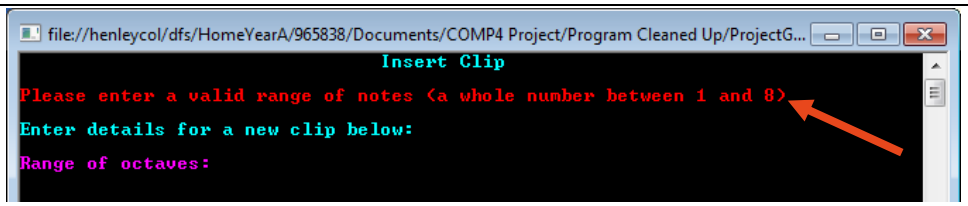
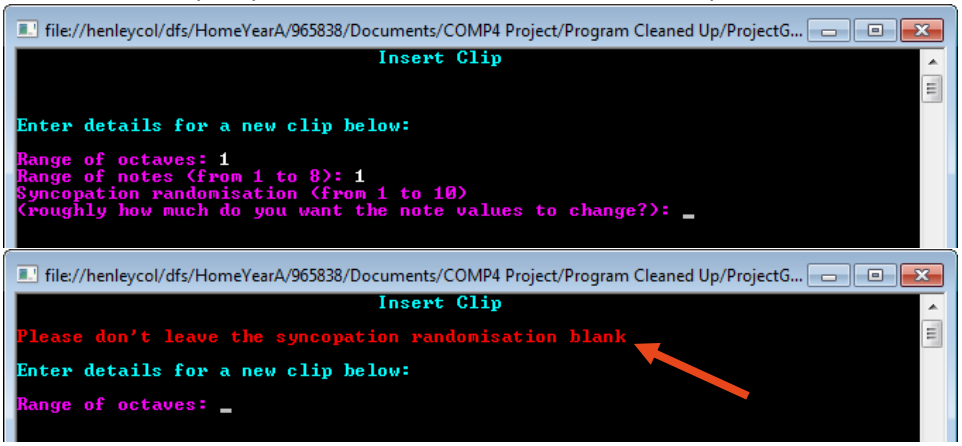
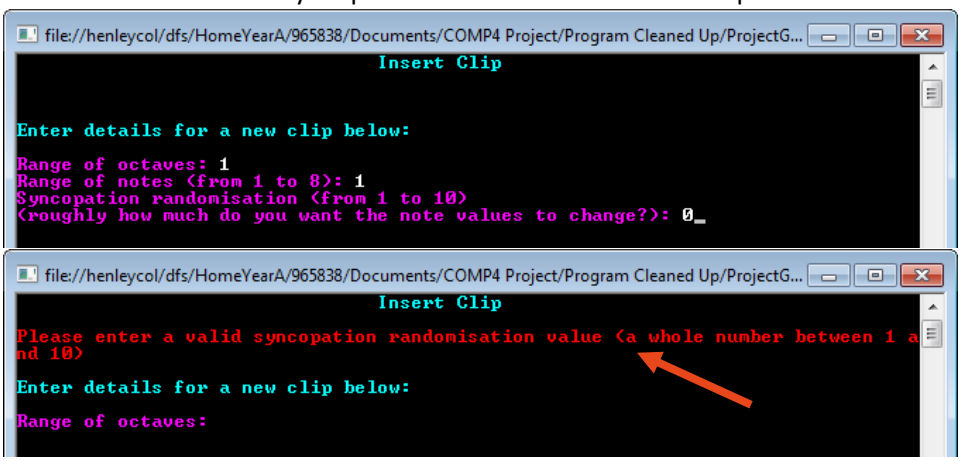
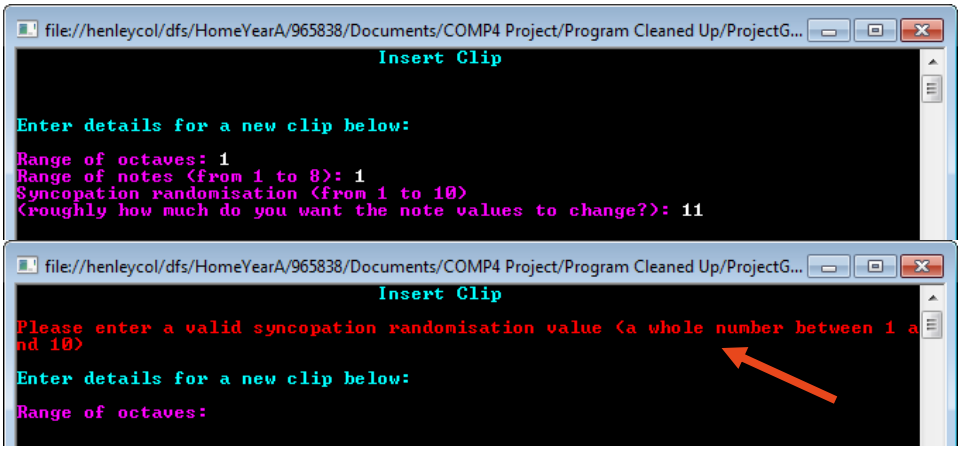
You can now see the green notes. This means that I know that the note generator algorithm can generate sharp notes, but some bug was preventing it from doing it in the first place. I have marked the test as passed.
</td>
<td></td>
</tr>
<tr>
<td>TR27</td>
<td>
For this test, I created a new arrangement in F minor:


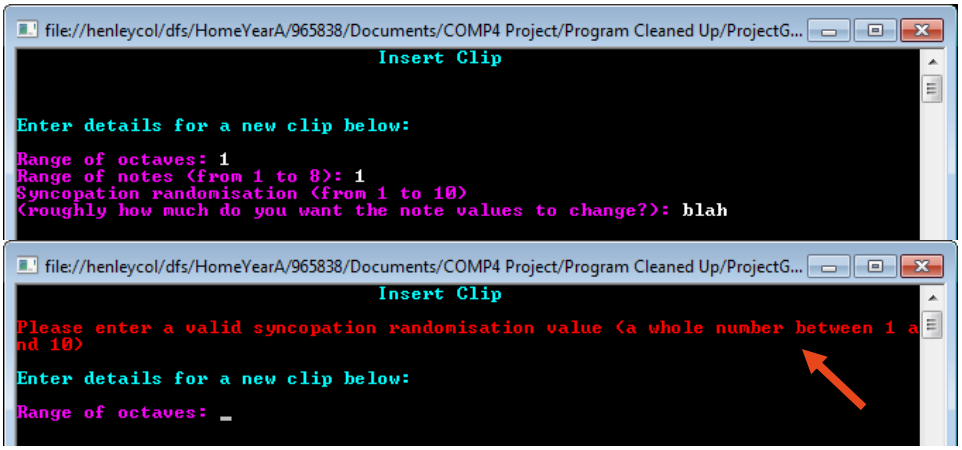
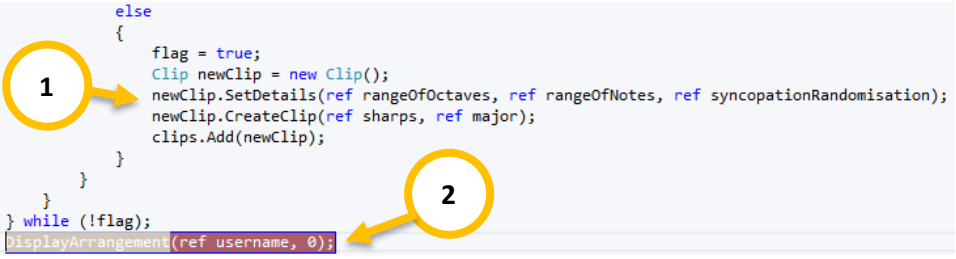
And created a clip using this data:
</td>
<td>4.3</td>
</tr>
</table>

	 And viewed it:  We can see a red note, which means that this test has passed.	
TR28	For this test, we need an arrangement in C major, so I've loaded my Test Arr 1 arrangement again and created a clip with the following details:  And these are the notes that followed:  The notes span between octaves 3 and 5, so this test has passed.	4.4
TR29	I entered an invalid range of octaves and pressed Enter: 	4.5

	 We have the correct error, so this test has passed.	
TR30	I left the range of octaves blank and pressed Enter: 	4.6
TR31	I entered a negative range of octaves and pressed Enter:  What happened here was that it did not catch the negative number and happily allowed me to enter the next field. This meant it is now using a negative number for the range of octaves, which is impossible. This test has failed. Again, this could be solved by having used an if statement to check if a negative number was entered.	4.7
TR32	I entered 3 billion for range of octaves and pressed Enter:  Originally, I was sceptical as to whether this error would be shown correctly or	4.8

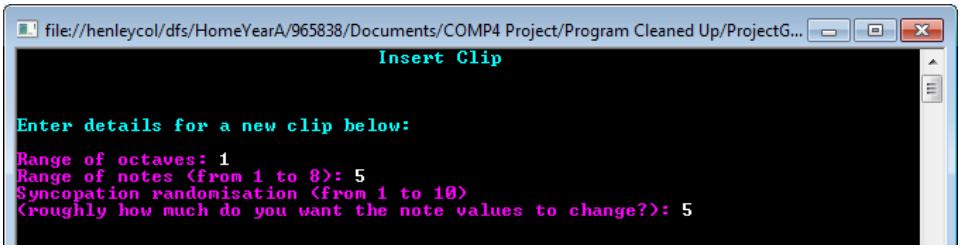
	not, but it turns out that I did implement a catch to check if the range of octaves was too large. If there was no catch, it would lead to an integer overflow exception. Anyway, this test has passed.	
TR33	I left the range of notes blank and pressed Enter: 	4.9
TR34	I entered an invalid range of notes and pressed Enter: 	4.10
TR35	I entered another invalid range of notes and pressed Enter: 	4.11
TR36	I entered text into range of notes and pressed Enter: 	4.12

		
TR37	I left the syncopation randomisation value blank and pressed Enter: 	4.13
TR38	I entered an invalid syncopation randomisation value and pressed Enter: 	4.14
TR39	I entered another invalid syncopation randomisation value and pressed Enter: 	4.15

<table>
<tr>
<td>TR40</td>
<td>I entered text into the syncopation randomisation field and pressed Enter:
 
</td>
<td></td>
</tr>
<tr>
<td>TR41</td>
<td>In order to see the contents of the clips list, we need to set a breakpoint in the code. Most of the code isn't important, but the bits that are are labelled:
 

1 This is the part where the clip is created (had notes assigned to) and added to the clips list.

2 This is the call for the method that displays the arrangement to the console. We want to break the program before the display is done so that we can apply a watch on the clip list, which is why there's a breakpoint here.

First, we enter the details for creating a new clip:
 

Then, I pressed Enter to enter the break mode and added a watch on the clips list. This is the contents of the created clip after the notes were generated:</td>
<td>4.17</td>
</tr>
</table>

Watch 1

Name	Value
clips	Count = 1
[0]	{ProjectGracenoteAlpha1Point1.Clip}
encoded	""
ID	0
name	null
notes	Count = 6
[0]	{ProjectGracenoteAlpha1Point1.Note}
noteValue	1
octave	3
pitchNumerical	"03"
[1]	{ProjectGracenoteAlpha1Point1.Note}
noteValue	1
octave	4
pitchNumerical	"12"
[2]	{ProjectGracenoteAlpha1Point1.Note}
noteValue	1
octave	3
pitchNumerical	"11"
[3]	{ProjectGracenoteAlpha1Point1.Note}
noteValue	1
octave	3
pitchNumerical	"14"
[4]	{ProjectGracenoteAlpha1Point1.Note}
noteValue	1
octave	5
pitchNumerical	"15"
[5]	{ProjectGracenoteAlpha1Point1.Note}
noteValue	3
octave	4
pitchNumerical	"03"
Raw View	
rangeOfNotes	5
rangeOfOctaves	1
saved	false
syncopationRandomisation	5
Static members	
Raw View	

We can deduce from this that 6 notes have been created.
Just to clarify, each note value is given as a code. As a reminder, this is what the codes mean:

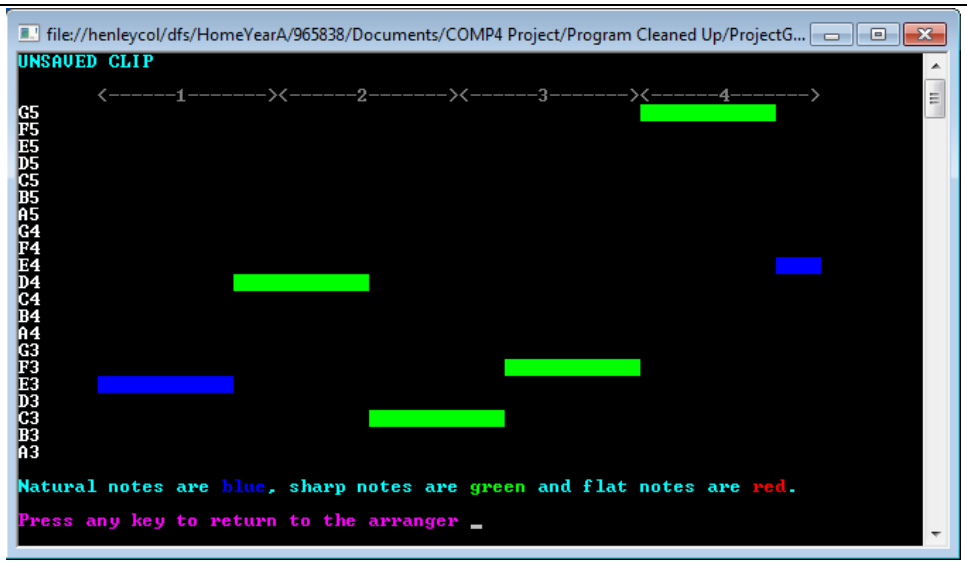
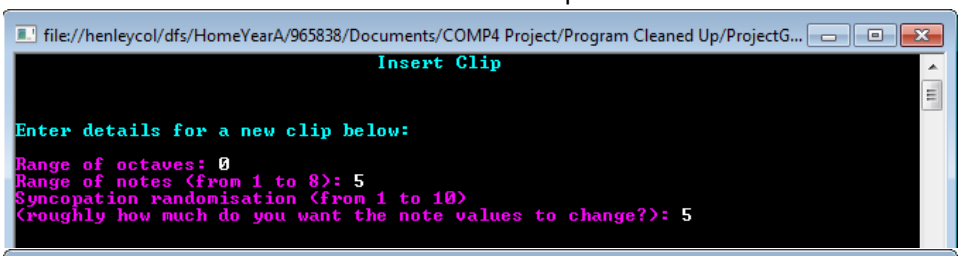
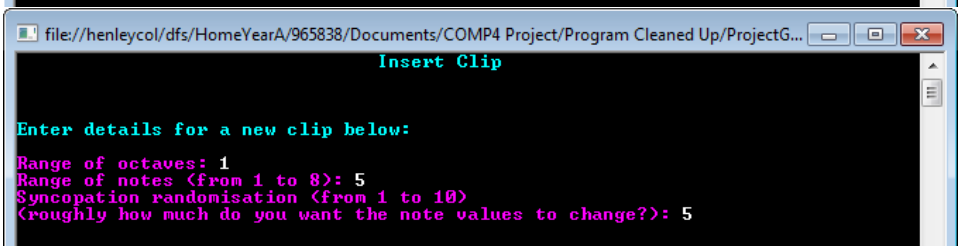
- 0 – semibreve (whole note)
- 1 – dotted minim (3/4 note)
- 2 – minim (1/2 note)
- 3 – crotchet (1/4 note)
- 4 – quaver (1/8 note)
- 5 – semiquaver (1/16 note)

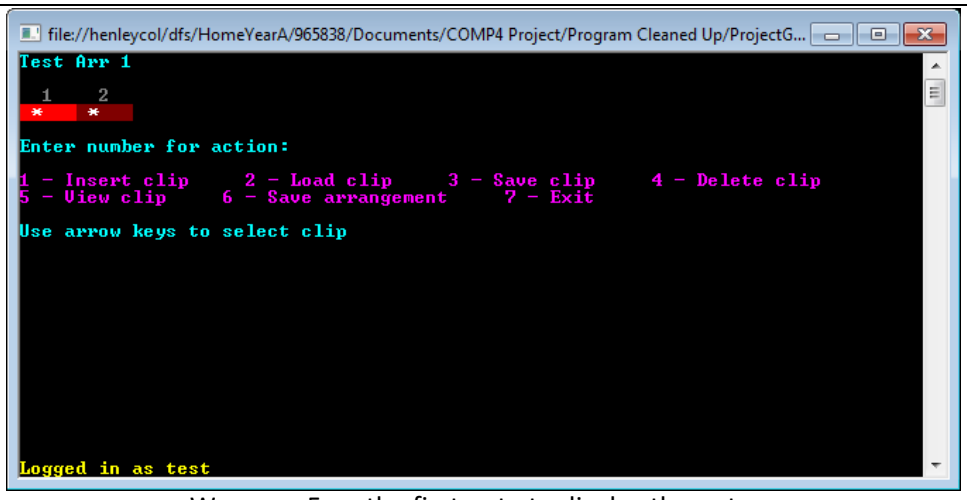
Also as a reminder, a pitch numerical code starting with a 0 is a natural note, 1 is a sharp note and 2 is a flat note. The next number goes from 1 (meaning C), 2 means D, 3 means E, all the way up the scale to 7 (meaning B).

Therefore, our notes have the following properties in plain English:

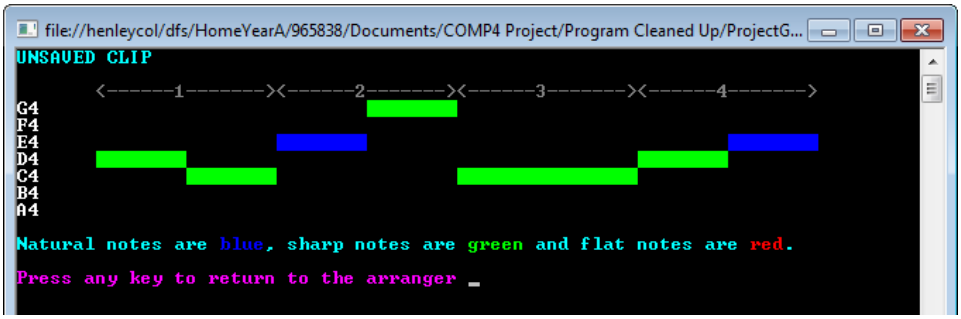
1. Dotted minim, octave 3, E natural
2. Dotted minim, octave 4, D♯
3. Dotted minim, octave 3, C♯
4. Dotted minim, octave 3, F♯
5. Dotted minim, octave 5, G♯
6. Crotchet, octave 4, E natural

Let's take a look at how the clip is displayed:

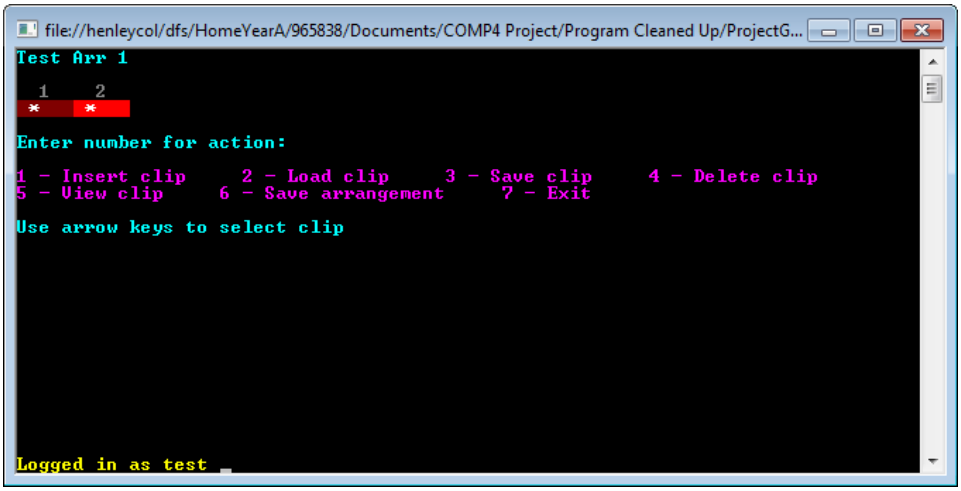
	 We know that clips 2 – 5 are sharp notes, and as we can see, they are shown as green in this window, which means sharp. The other two notes are natural, and you can see that they are blue, which means natural. The first note is E octave 3, and as you can see, it's within the E3 row, so that's been displayed correctly. It should also be a dotted minim (3/4 note), and it fills 3 quarters of the first bar, so that's been drawn correctly. The other dotted minim notes have also been drawn the correct length and been put in the correct row. The crotchet note at the end is correctly shown as being a quarter of the bar. Therefore, this test has passed.	
TR42	We create two clips:   To get this in the arranger:	4.18



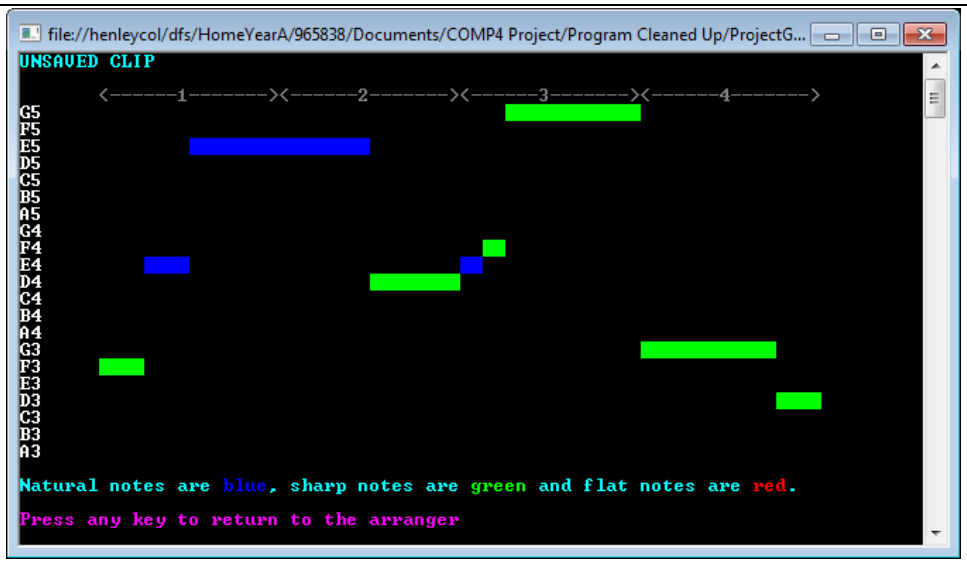
We press 5 on the first note to display the notes:

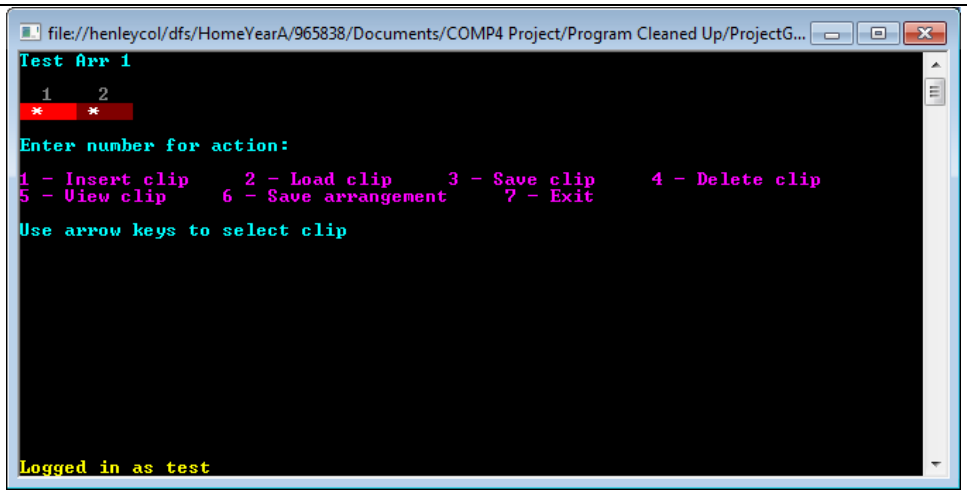


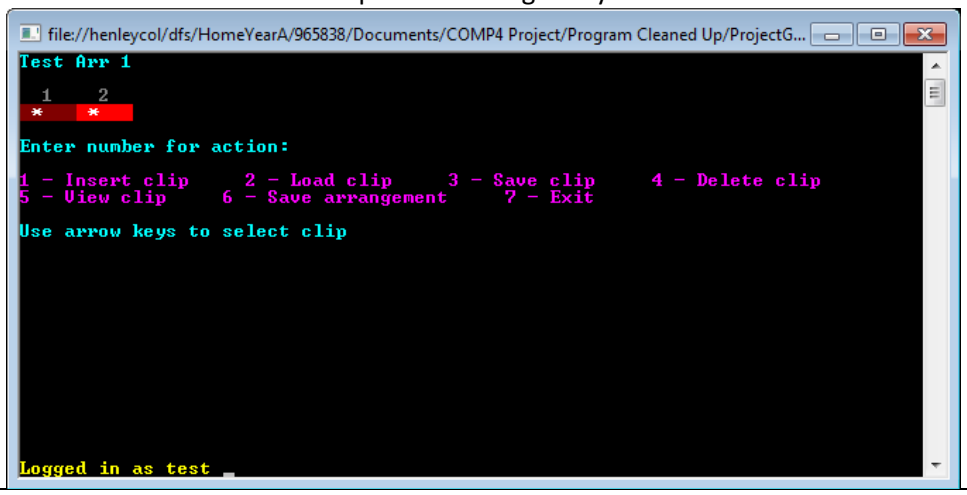
We then select the other note:

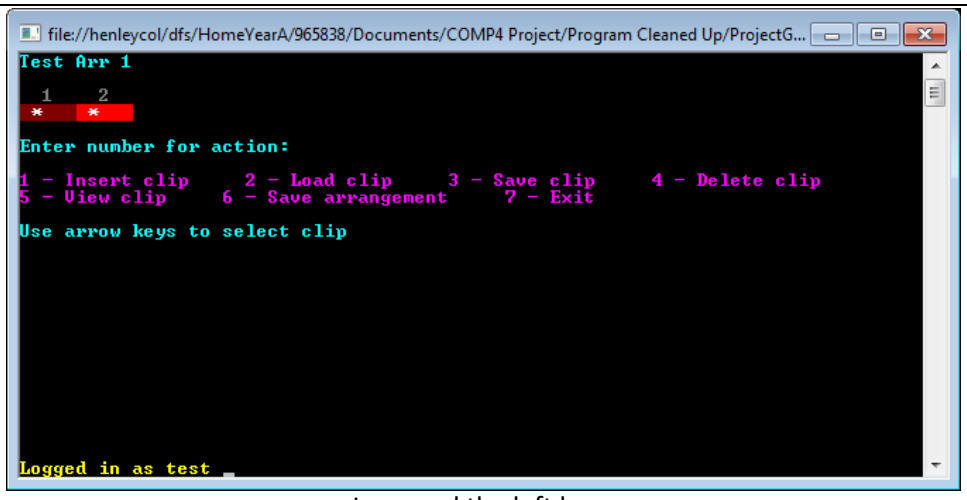
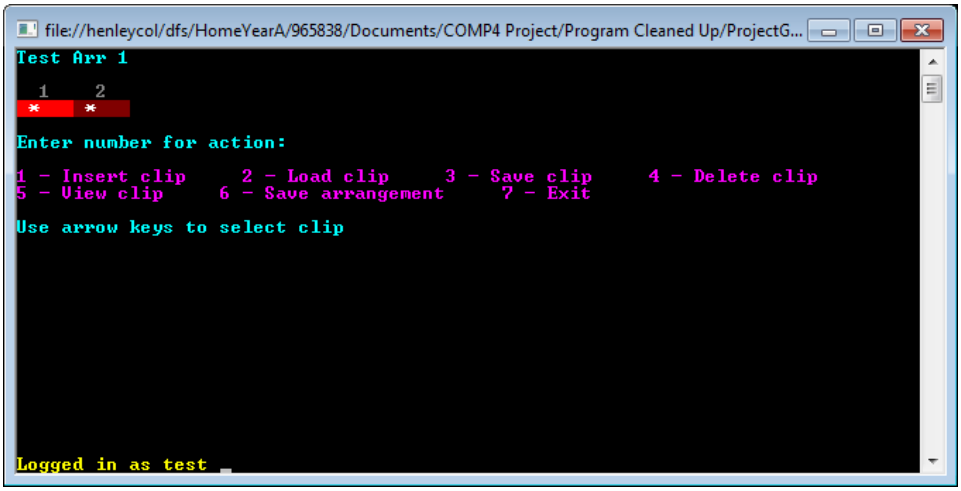
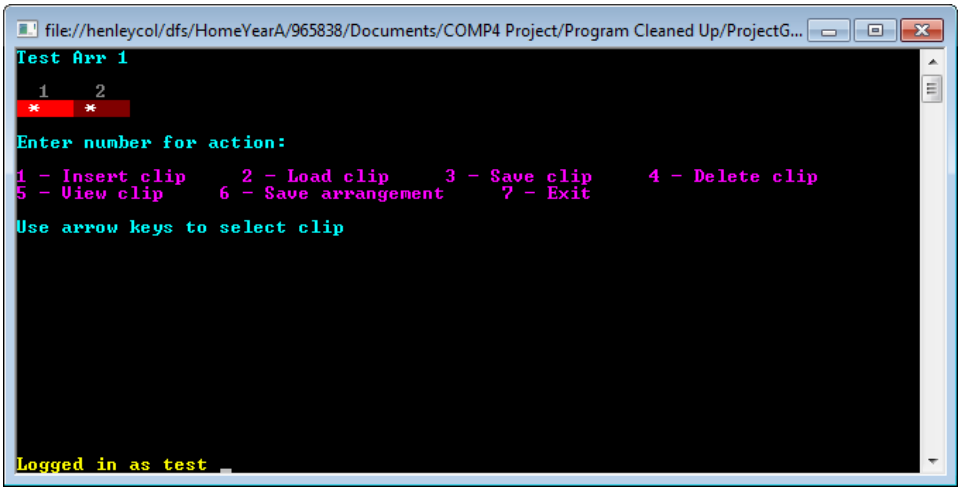


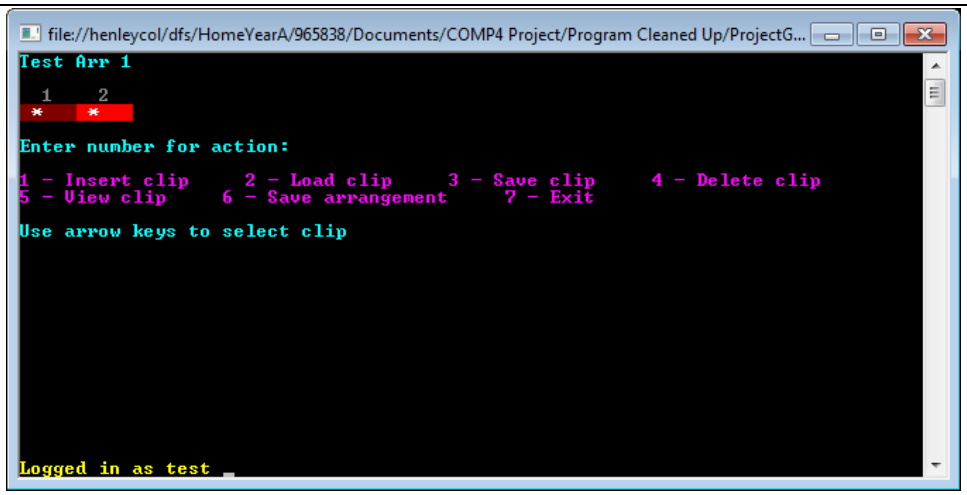
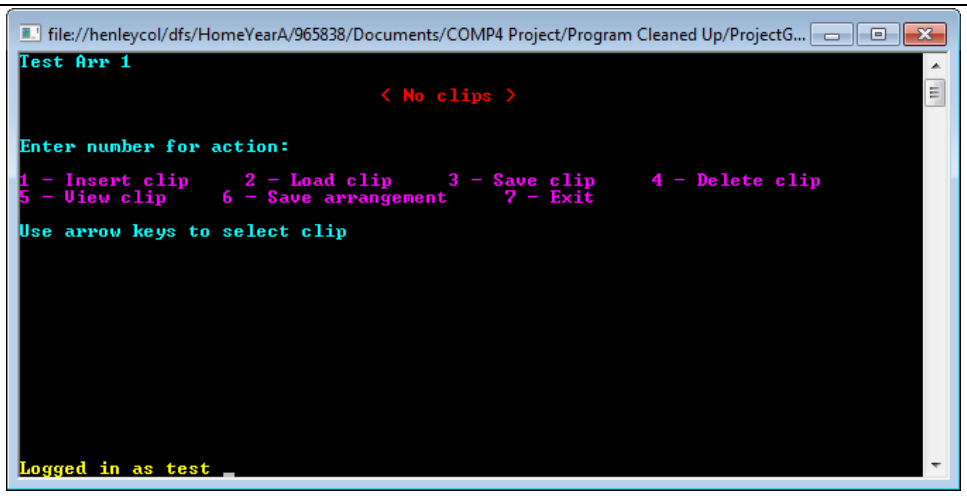
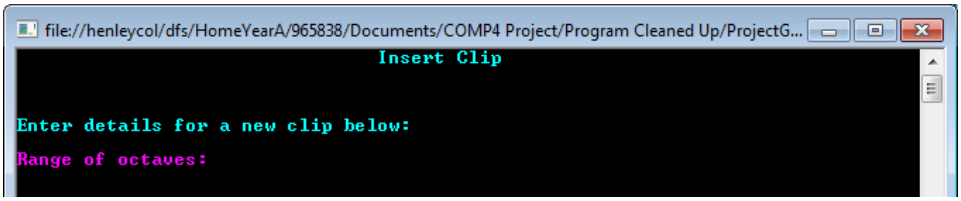
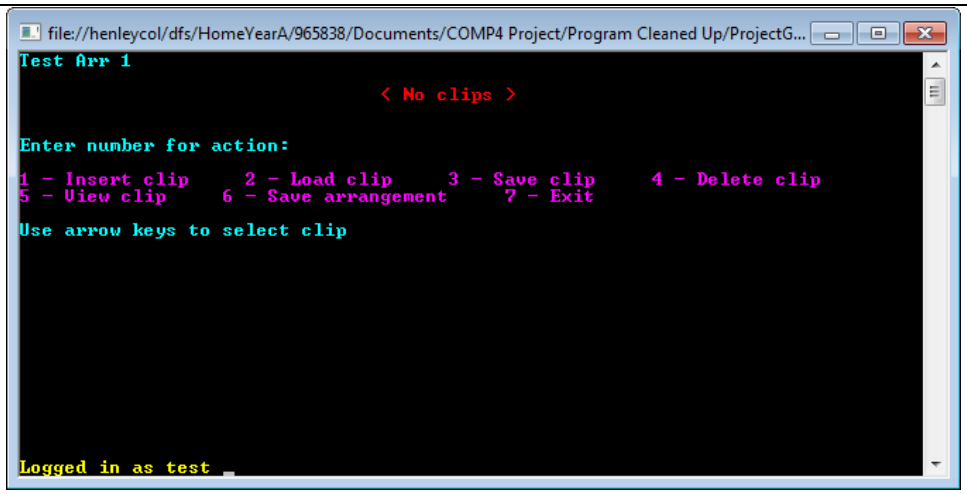
And press 5:

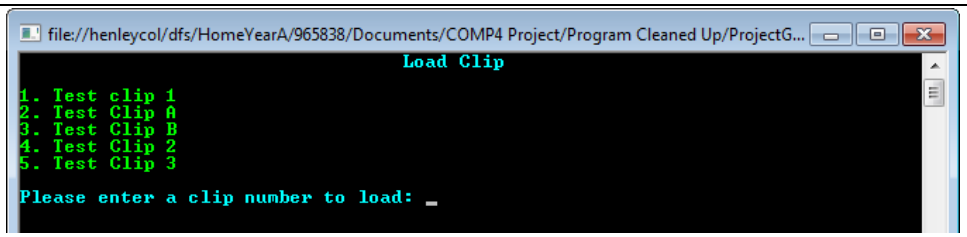
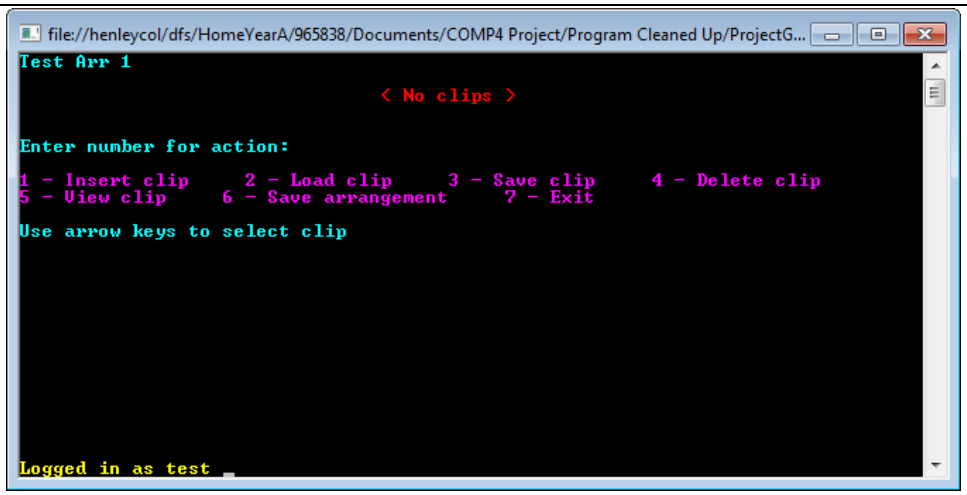
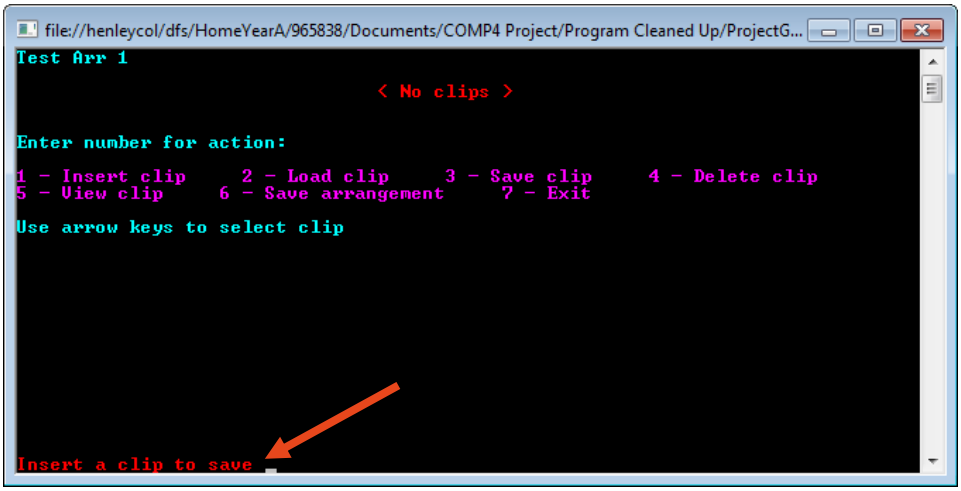
<table>
<tr>
<td></td>
<td> 

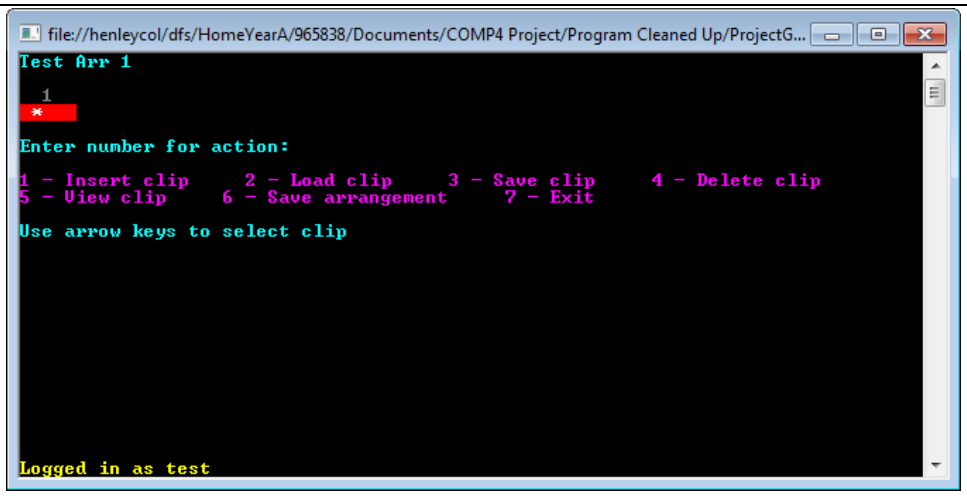
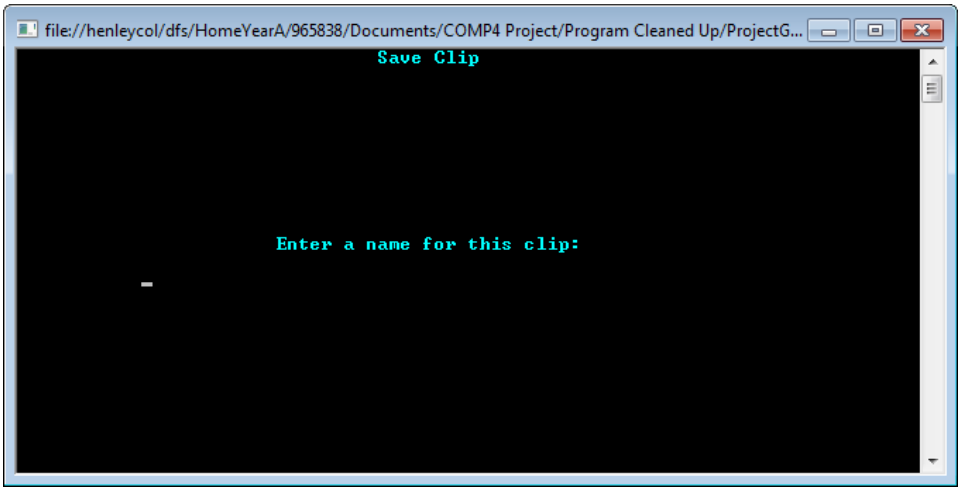
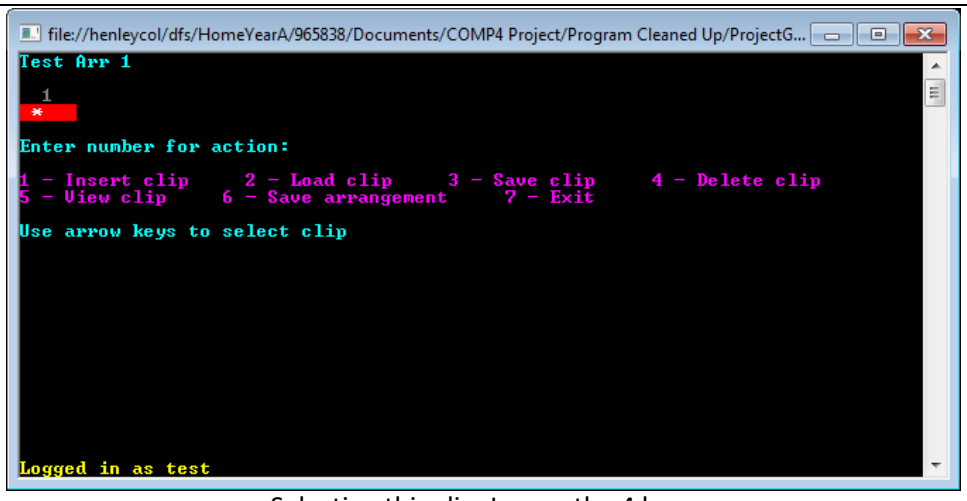
As you can see, the clip with notes in just octave 4 was positioned first, and the clip with notes in octaves 3 – 5 was positioned second. Therefore, this test has passed.</td>
<td></td>
</tr>
<tr>
<td>TR43</td>
<td> 

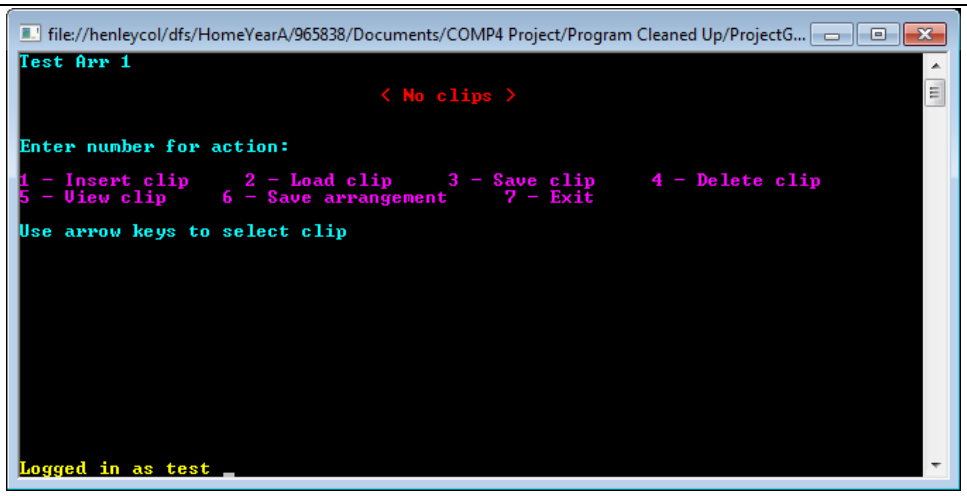
I pressed the right key:

</td>
<td>5.1</td>
</tr>
</table>

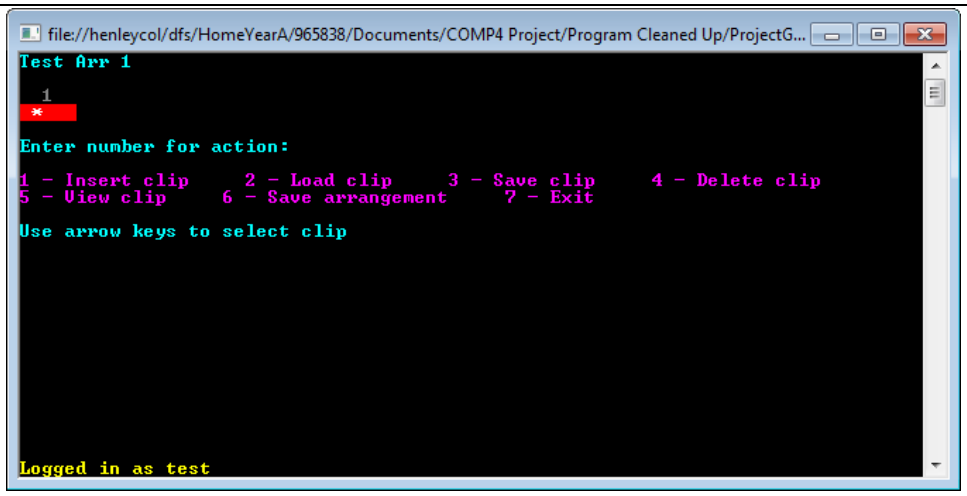
TR44	 I pressed the left key: 	5.2
TR45	I pressed the left key here:  Pressing the left key does nothing, which is what we expect.	5.3
TR46	I pressed the right key here:	5.4

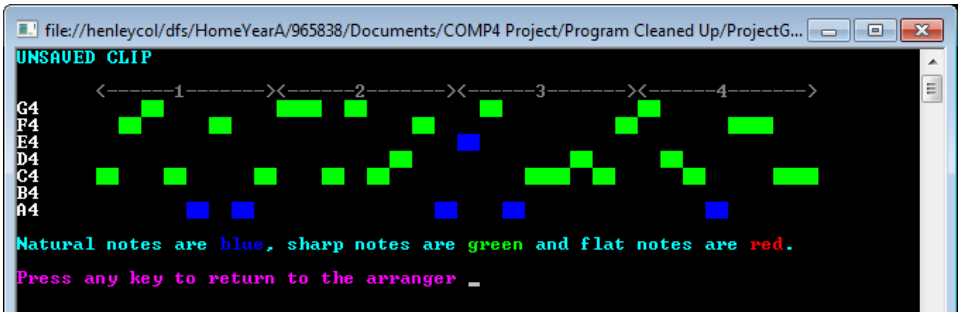
	 Pressing the right key does nothing.	
TR47	 I pressed the 1 key: 	5.5
TR48	 I pressed the 2 key:	5.6

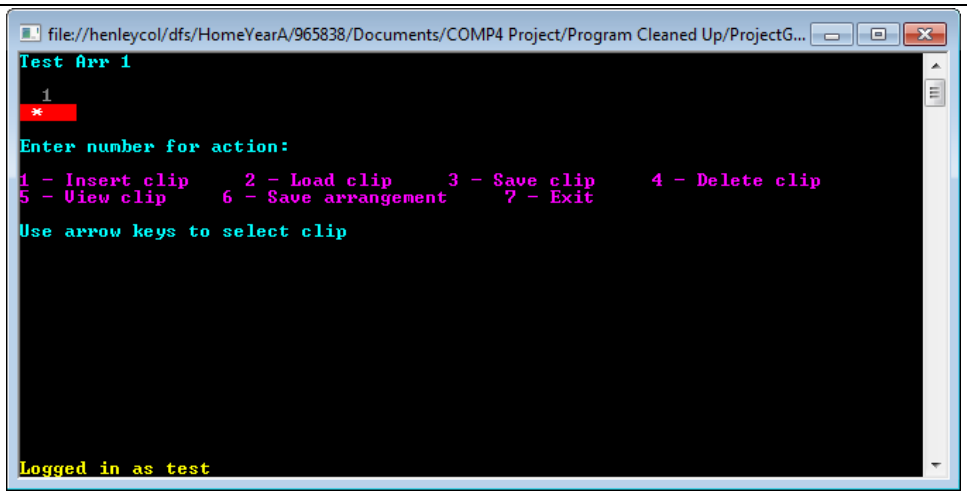
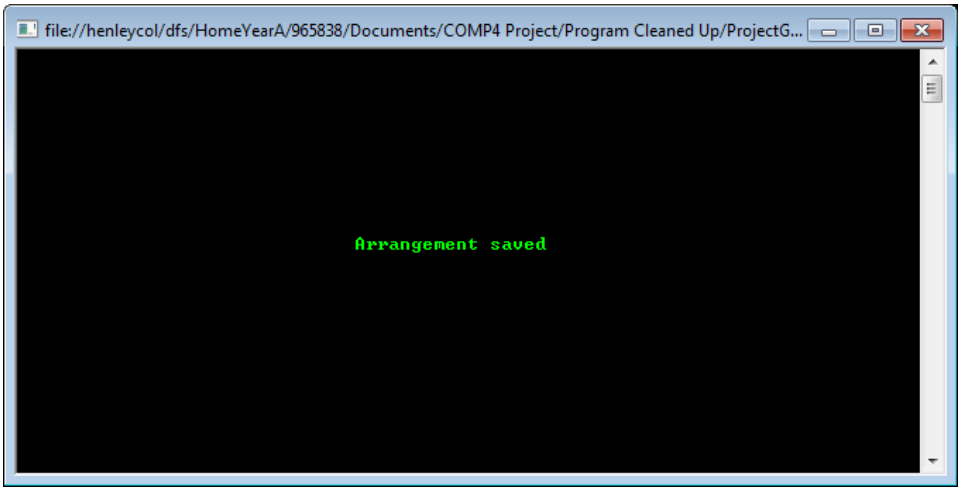
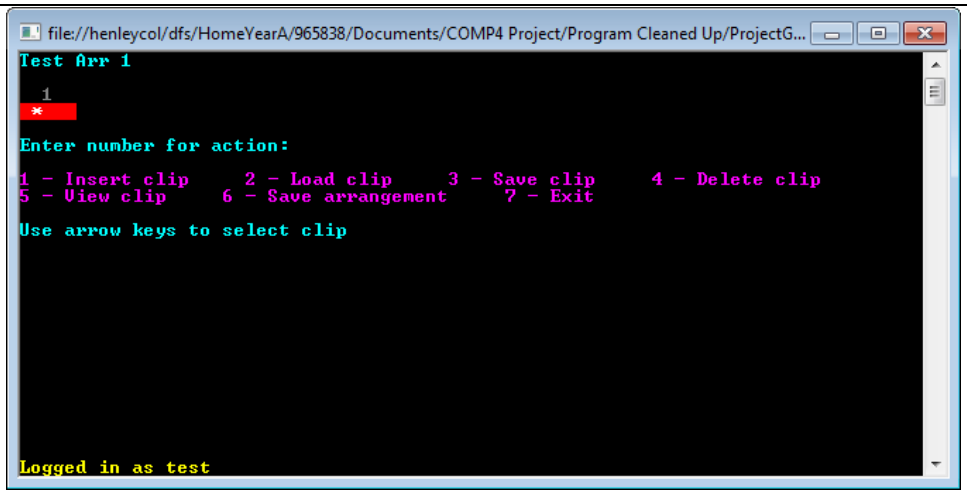
	 This is the list of some test clips that I made at an earlier date. We'll come on to loading clips later. The fact that they're displayed means the test has passed.	
TR49	 I pressed the 3 key:  We have an error. Technically, this error is expected because I do not have any clips in this example to be able to be saved. If I create a clip:	5.7

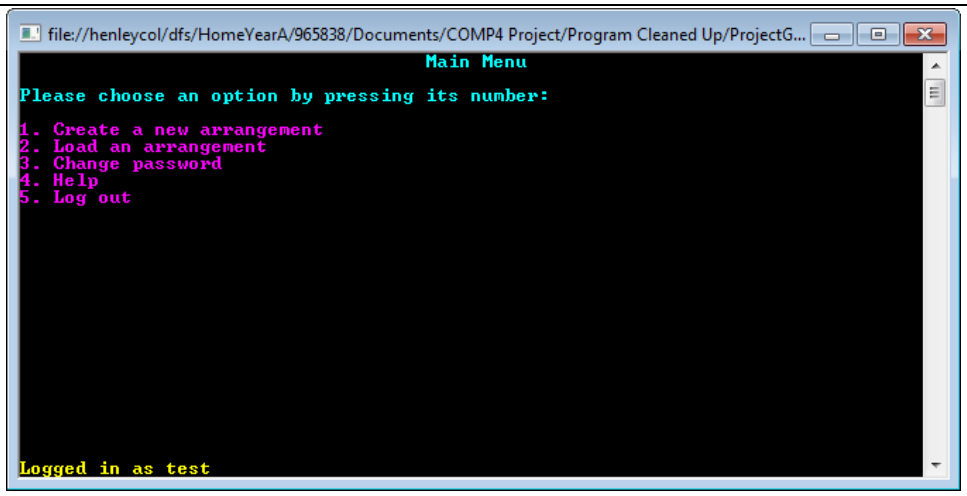
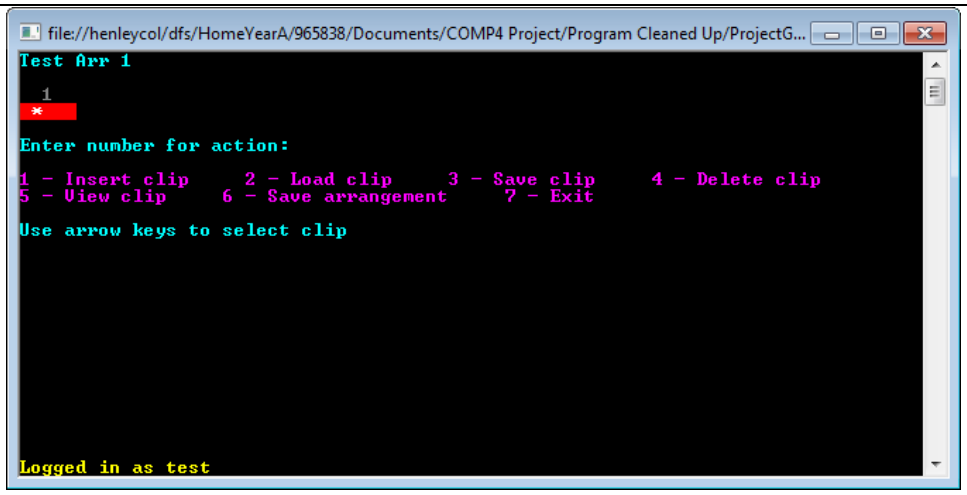
	 And press 3 again:  This is what we want. This test has passed.	
TR50	 Selecting this clip, I press the 4 key:	5.8

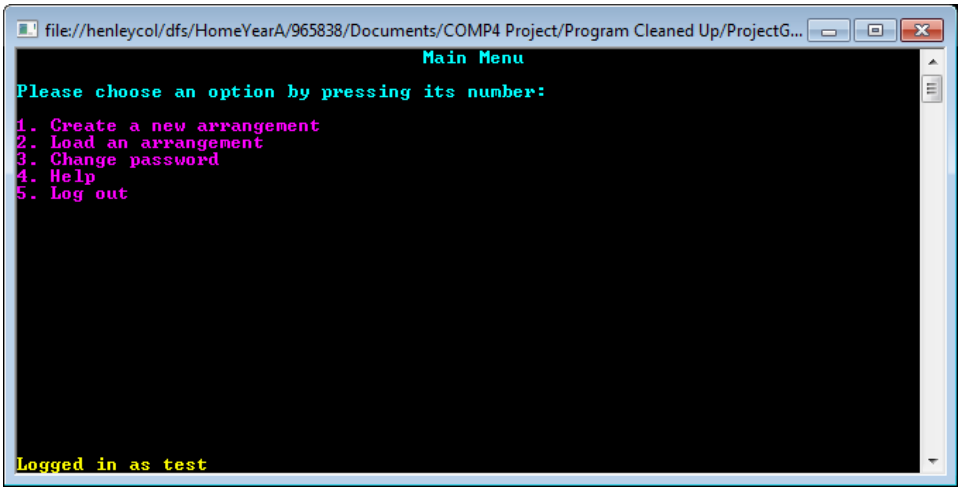
<table>
<tr>
<td></td>
<td> 

The clip is now gone, so this test has passed.</td>
<td></td>
</tr>
<tr>
<td>TR51</td>
<td> 

I pressed the 5 key:

</td>
<td>5.9</td>
</tr>
</table>

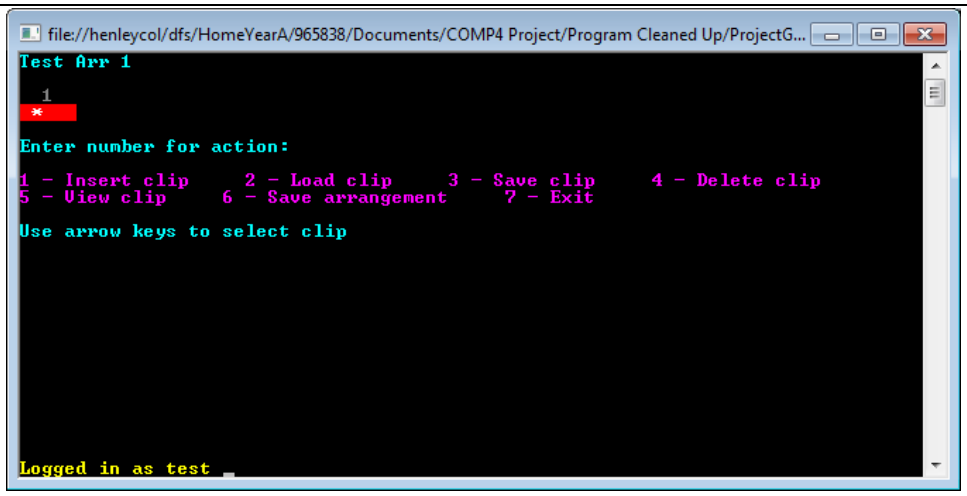
TR52	 I pressed the 6 key: 	5.10
TR53	 I pressed the 7 key to exit to the main menu:	5.11

<table>
<tr><td></td><td> 
</td><td></td></tr>
<tr><td>TR54</td><td> 

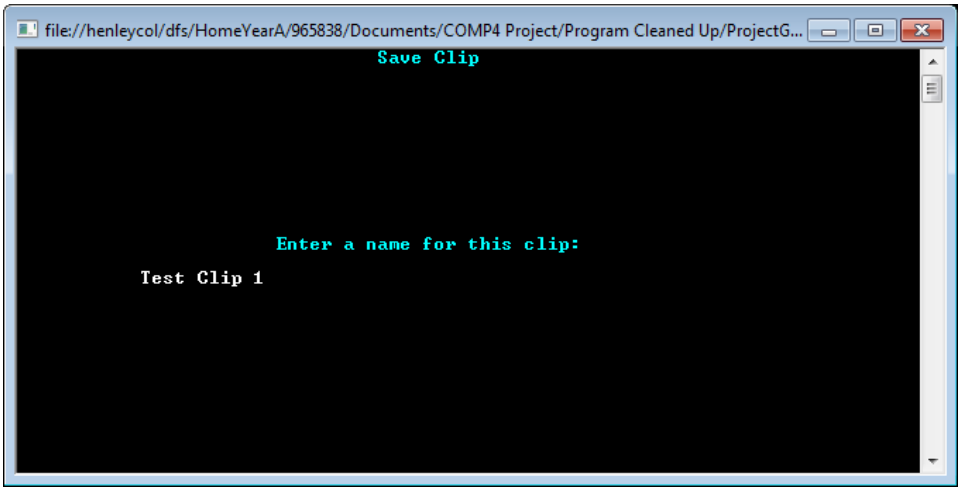
I press the 8 key:


The main menu is displayed, which is not what we expect. Again, I didn't put in a catch to check for an unexpected key press.</td><td>5.12</td></tr>
</table>

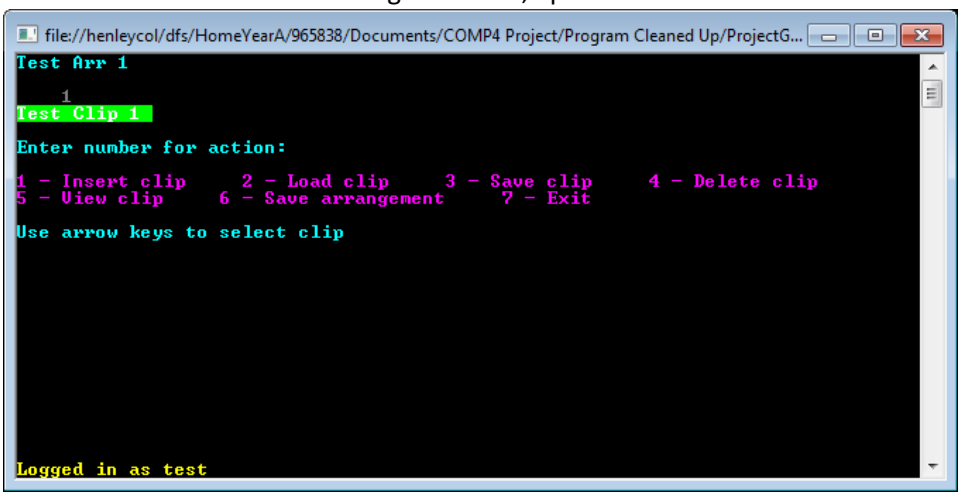
TR55		6.1



I pressed the 3 key on the selected clip:

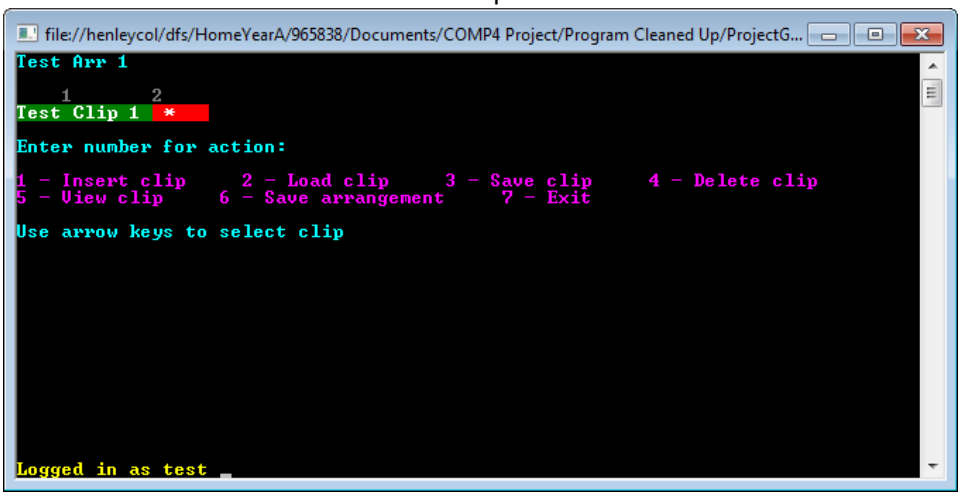


After entering the name, I pressed Enter:

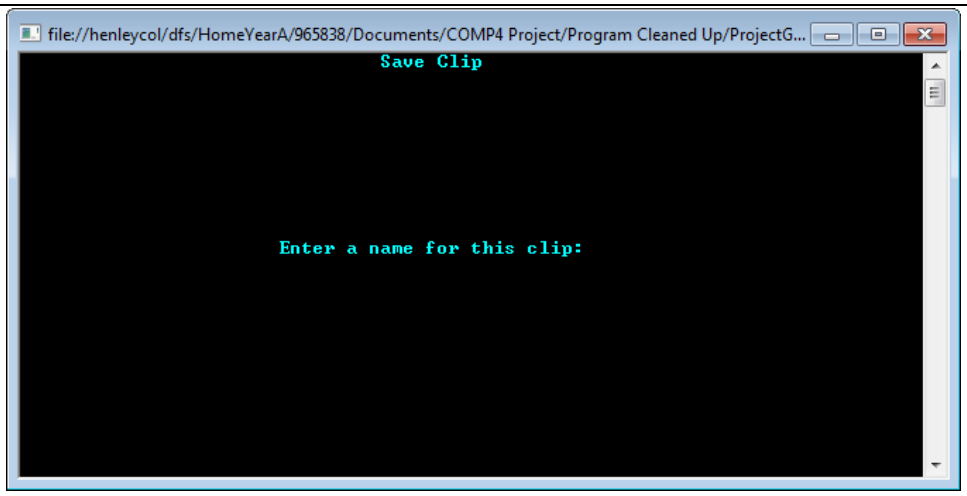


The clip is now green because it has been saved. Now, we must check the Clips table in the database:

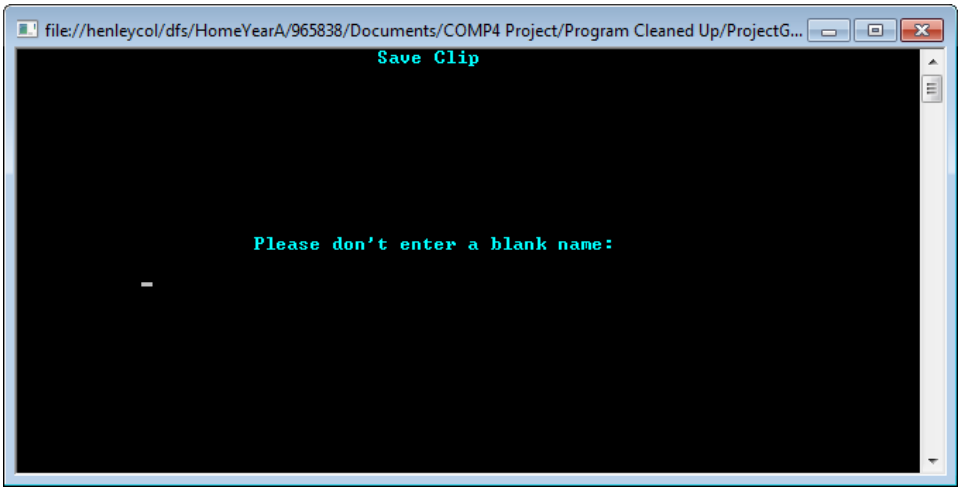
PK_ClipID	ClipName	NoteData	FK_AccountID
2	Test clip 1	XA1B05C4YXA2B04C4YXA2B05C4YXA4B03C4YXA1B05C4YXA1B0...	4
3	Test Clip A	XA2B05C4YXA4B03C4YXA2B03C4YXA4B03C4YXA2B04C4YXA2B0...	4
4	Test Clip B	XA0B01C4YXA4B04C4YXA3B06C4YXA4B04C4YXA4B06C4YXA2B0...	4
5	Test Clip 2	XA4B06C4YXA2B05C4YXA4B02C4YXA1B01C4YXA2B05C4YXA4B0...	4
6	Test Clip 3	XA1B05C4YXA2B04C4YXA2B05C4YXA4B03C4YXA1B05C4YXA1B0...	4
7	Test Clip 1	XA4B11C4YXA4B14C4YXA4B15C4YXA4B11C4YXA4B06C4YXA4B1...	4

<table>
<tr>
<td></td>
<td>A new record has been created after the other clips I'd created at an earlier date. Note that the clip ID is 7. We were using the Test Arr 1 arrangement. Let's check the Arrangement table to see what it's ID is:
<table>
<tr><th>PK_ArrangementID</th><th>ArrangementName</th></tr>
<tr><td>8</td><td>Test Arrangement</td></tr>
<tr><td>9</td><td>Test Arrangement 2</td></tr>
<tr><td>11</td><td>Test Arrangement 3</td></tr>
<tr><td>12</td><td>Test Again</td></tr>
<tr><td>13</td><td>Test Arr 1</td></tr>
<tr><td>15</td><td>Test Arr 2</td></tr>
<tr><td>17</td><td>Test Arr 3</td></tr>
</table>
It has the ID of 13. Now let's check the ClipUses table to see if the clip and the arrangement have been linked together:
<table>
<tr><th>FK_ClipID</th><th>FK_ArrangementID</th></tr>
<tr><td>2</td><td>8</td></tr>
<tr><td>3</td><td>9</td></tr>
<tr><td>4</td><td>9</td></tr>
<tr><td>5</td><td>8</td></tr>
<tr><td>6</td><td>8</td></tr>
<tr><td>7</td><td>13</td></tr>
</table>
The clip ID of 7 and arrangement ID of 13 are related. Therefore, the clip has been saved properly and the test has passed.</td>
<td></td>
</tr>
<tr>
<td>TR56</td>
<td>We create another clip for this next test:


And press the 3 key to save:</td>
<td>6.2</td>
</tr>
</table>



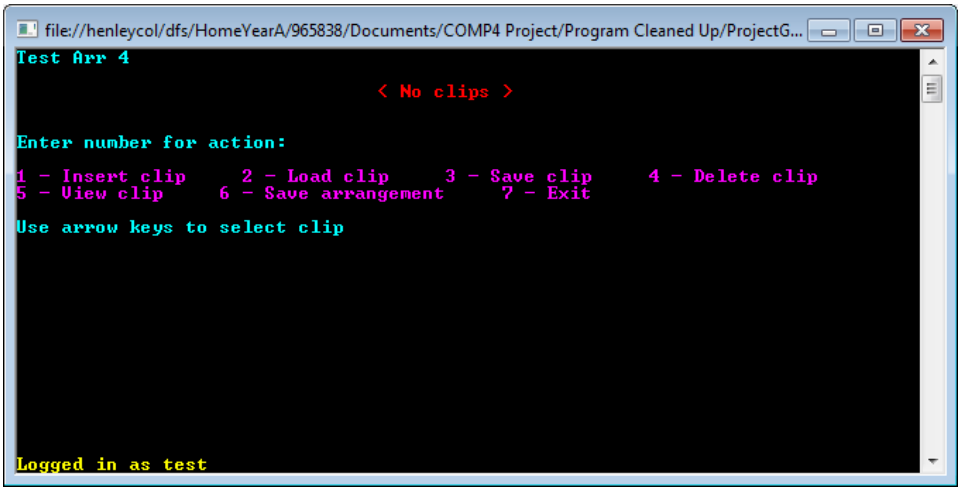
And leave the name blank, and press Enter:



TR57

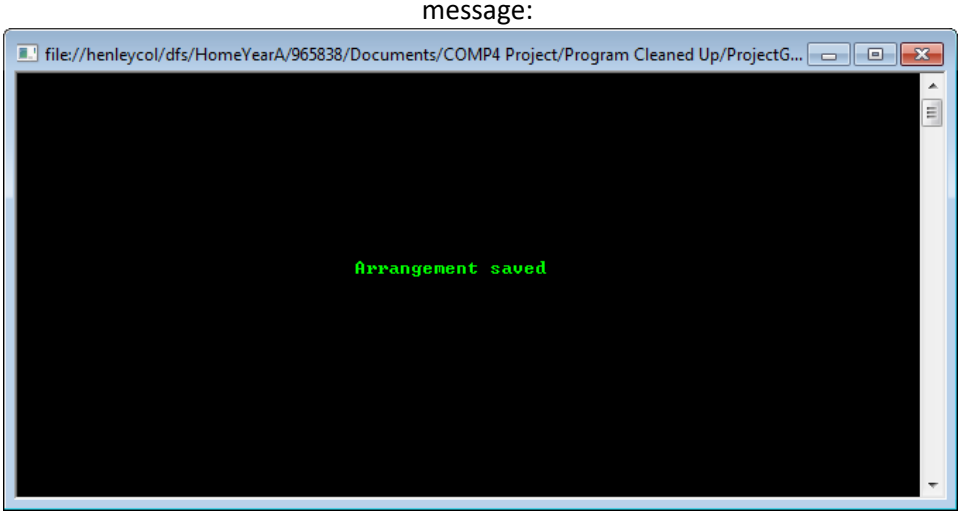
I have created a new blank arrangement:

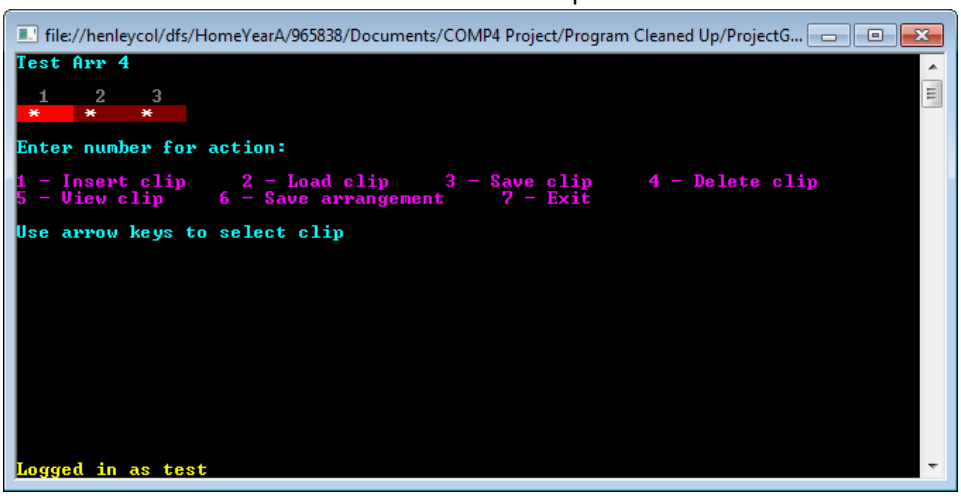
7.1



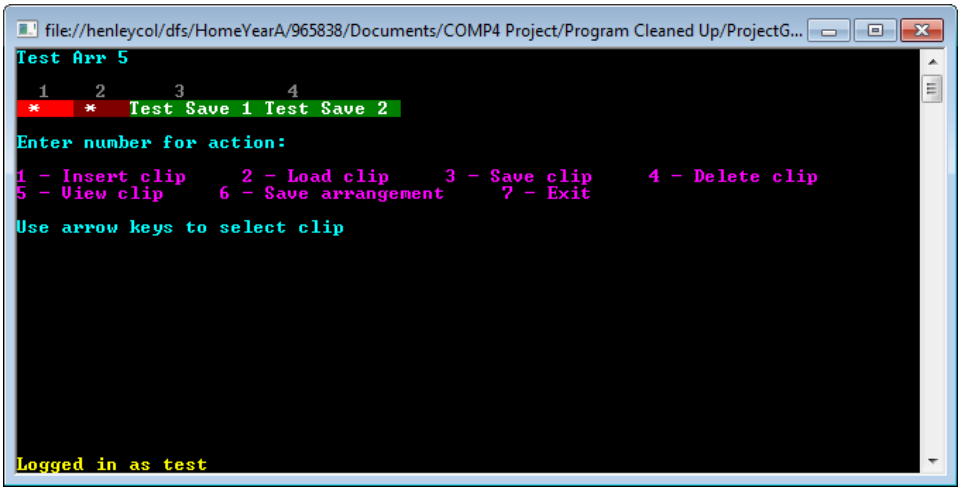
Here is the record in the database:

PK_ArrangementID	ArrangementName	ArrangementData	ClipContentsData	FK_AccountID
8	Test Arrangement	001011D21BE202717462FA08A68	E202717462FXA3B03C4YXA4B02C4	4
9	Test Arrangement 2	0011C24A39A49BE304842317F	E304842317FXA1B02C4YXA4B04C4	4
11	Test Arrangement 3	0031C24BE730148066FBE1367104	E730148066FXA0B15C4YXA3B11C4	4
12	Test Again	0071C24	*NULL*	4
13	Test Arr 1	0031C24BE1474562967F	E1474562967FXA4B11C4YXA4B14C	4
15	Test Arr 2	0011C24	*NULL*	4
17	Test Arr 3	0041D24	*NULL*	4
18	Test Arr 4	0011C24	*NULL*	4

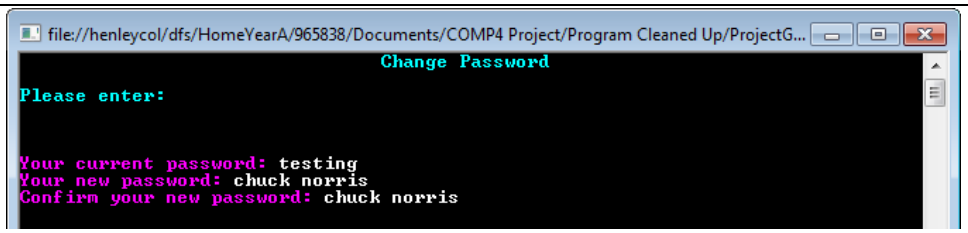
<table>
<tr>
<td></td>
<td>
We can deduce that is a blank arrangement with no clips, either saved or unsaved, inserted. Now, if we press 6 in the program to save, we get this message:


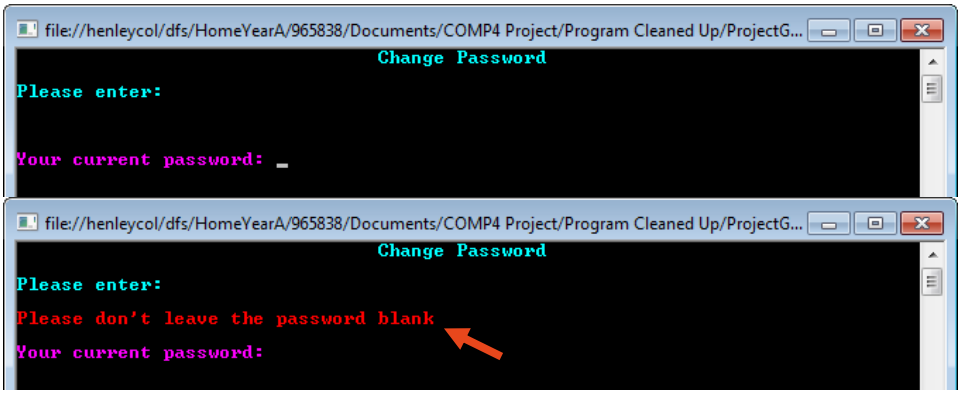
The record in the database should be unchanged:
<table>
<tr><th>PK_ArrangementID</th><th>ArrangementName</th><th>ArrangementData</th><th>ClipContentsData</th><th>FK_AccountID</th></tr>
<tr><td>8</td><td>Test Arrangement</td><td>001011D21BE202717462FA08A68</td><td>E202717462FXA3B03C4YXA</td><td>4</td></tr>
<tr><td>9</td><td>Test Arrangement 2</td><td>0011C24A39A49BE304842317F</td><td>E304842317FXA1B02C4YXA</td><td>4</td></tr>
<tr><td>11</td><td>Test Arrangement 3</td><td>0031C24BE730148066FBE1367104</td><td>E730148066FXA0B15C4YXA</td><td>4</td></tr>
<tr><td>12</td><td>Test Again</td><td>0071C24</td><td>NULL</td><td>4</td></tr>
<tr><td>13</td><td>Test Arr 1</td><td>0031C24BE1474562967F</td><td>E1474562967FXA4B11C4YX</td><td>4</td></tr>
<tr><td>15</td><td>Test Arr 2</td><td>0011C24</td><td>NULL</td><td>4</td></tr>
<tr><td>17</td><td>Test Arr 3</td><td>0041D24</td><td>NULL</td><td>4</td></tr>
<tr><td>18</td><td>Test Arr 4</td><td>0011C24</td><td></td><td>4</td></tr>
</table>
…which it is. This test has passed.
</td>
<td></td>
</tr>
<tr>
<td>TR58</td>
<td>
I have created 3 clips:


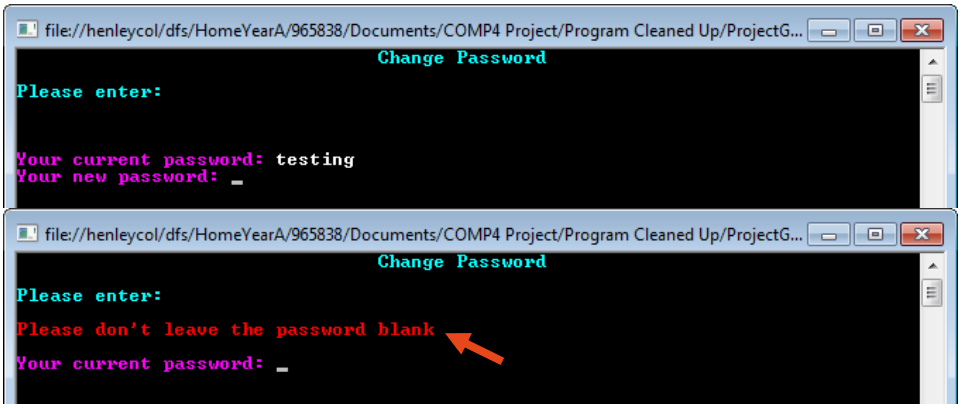
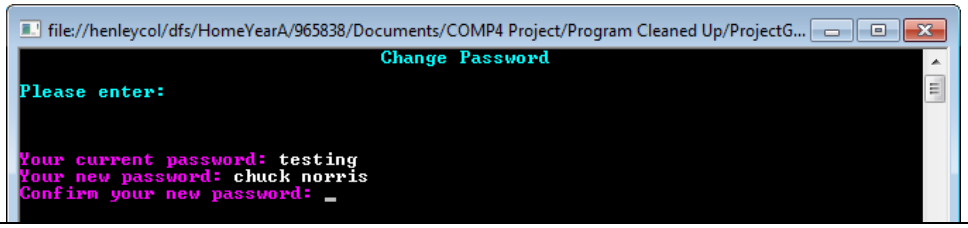
Once I press 6 to save the arrangement, this is what the arrangement table now contains (I've reordered it to show this new one first):
<table>
<tr><th>PK_ArrangementID</th><th>ArrangementName</th><th>ArrangementData</th></tr>
<tr><td>18</td><td>Test Arr 4</td><td>0011C24BE2039280099FBE1739728085FBE503529823F</td></tr>
</table>
<table>
<tr><th>ClipContentsData</th><th>FK_AccountID</th></tr>
<tr><td>E2039280099FXA2B03C4YXA3B02C4YXA3B01C4YXA2B04C4YXA...</td><td>4</td></tr>
</table>
We can see that the clip data has now been serialised, with the 3 keys in the ArrangementData field and the note data in the ClipContentsData field. This test has passed.
</td>
<td>7.2</td>
</tr>
</table>

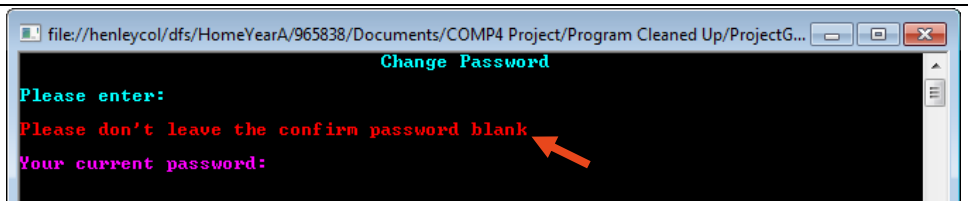
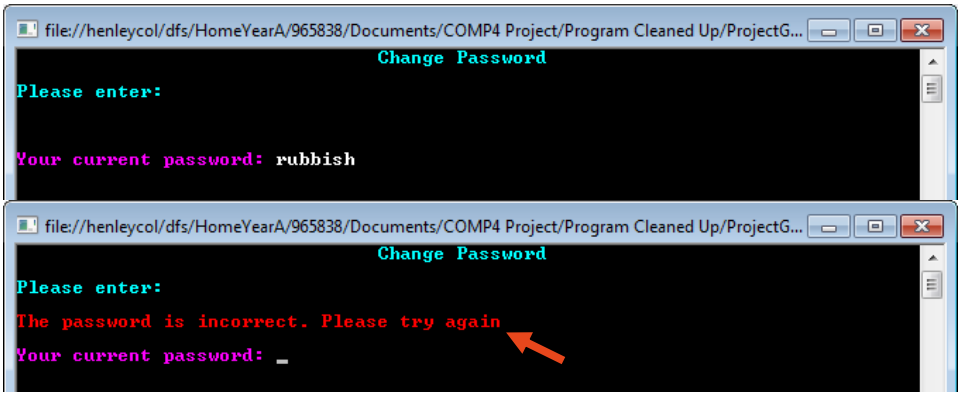
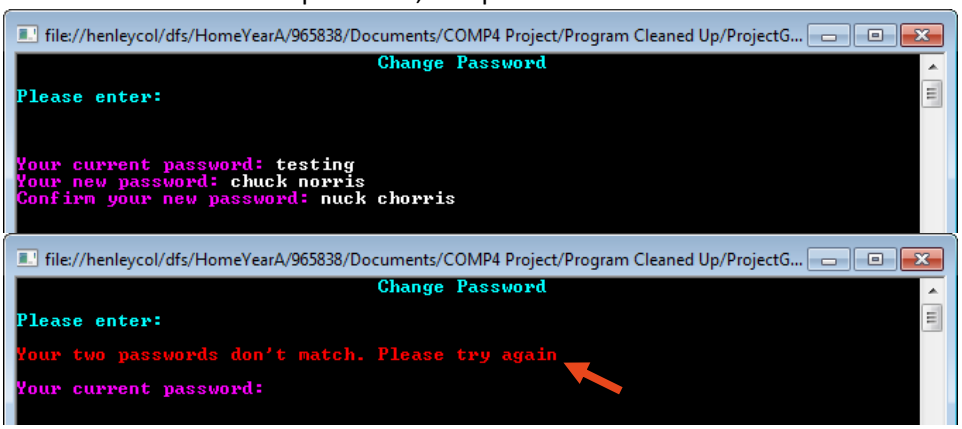
<table>
<tr>
<td>TR59</td>
<td>
I have created a new arrangement with the following clips:


I pressed 6 to save it, and this is the record in the arrangement table:
<table>
<tr><th>PK_ArrangementID</th><th>ArrangementName</th><th>ArrangementData</th></tr>
<tr><td>19</td><td>Test Arr 5</td><td>0011C24BE223602441FBE574070352FA819A919</td></tr>
</table>
<table>
<tr><th>ClipContentsData</th><th>FK_AccountID</th></tr>
<tr><td>E223602441FXA2B05C4YXA4B01C4YXA</td><td>4</td></tr>
</table>
We can see in the ArrangementData the two BE-F keys for the unsaved clips, and the A819 and A919 clips that were saved. Let's check the ClipUses table:
<table>
<tr><th>FK_ClipID</th><th>FK_ArrangementID</th></tr>
<tr><td>2</td><td>8</td></tr>
<tr><td>3</td><td>9</td></tr>
<tr><td>4</td><td>9</td></tr>
<tr><td>5</td><td>8</td></tr>
<tr><td>6</td><td>8</td></tr>
<tr><td>7</td><td>13</td></tr>
<tr><td>8</td><td>19</td></tr>
<tr><td>9</td><td>19</td></tr>
</table>
You can see that we have two clip uses that were in arrangement 19, which is what we had. If we check the Clips table:
<table>
<tr><th>PK_ClipID</th><th>ClipName</th><th>NoteData</th><th>FK_AccountID</th></tr>
<tr><td>9</td><td>Test Save 2</td><td>XA2B04C4YXA2B03C4YXA2B05C4YXA2B06C4YXA2B04C4YXA2B0...</td><td>4</td></tr>
<tr><td>8</td><td>Test Save 1</td><td>XA1B06C4YXA1B06C4YXA1B03C4YXA1B01C4YXA1B03C4YXA3B0...</td><td>4</td></tr>
</table>
You can see that the two clips that we saved have had their note data properly saved. This test has passed.
</td>
<td>7.3</td>
</tr>
<tr>
<td>TR60</td>
<td>
Before we change the password, this is the hash that the Accounts table contains:
<table>
<tr><th>PK_AccountID</th><th>Username</th><th>Password</th></tr>
<tr><td>4</td><td>test</td><td>z4DNiu1ILV0VJ9fccvzv+E5jJlkoSER9LcCw6H38mpA=</td></tr>
</table>
Once we enter the details for a new password:
</td>
<td>8.1</td>
</tr>
</table>

<table>
<tr><td></td><td> 

…and press Enter, this is what the table now shows:
<table>
<tr><th>PK_AccountID</th><th>Username</th><th>Password</th></tr>
<tr><td>4</td><td>test</td><td>CfVzxNNBzLgrs8ni85ED/sZsUomXXpwNQjE6loTHNdA=</td></tr>
</table>
The hash has changed, so this test has passed.</td><td></td></tr>
<tr><td>TR61</td><td>In the change password wizard, I left the current password field blank and pressed Enter:


This is the expected error.</td><td>8.2</td></tr>
<tr><td>TR62</td><td>I left the new password field blank and pressed Enter:

</td><td>8.3</td></tr>
<tr><td>TR63</td><td>I left the confirm new password blank and pressed Enter:

</td><td>8.4</td></tr>
</table>

		
TR64	I entered the wrong current password (the correct one is testing) and pressed Enter:  	8.5
TR65	I entered mismatching passwords for the new password and confirm new password, and pressed Enter:  	8.6

System Maintenance Guide

1 Introduction

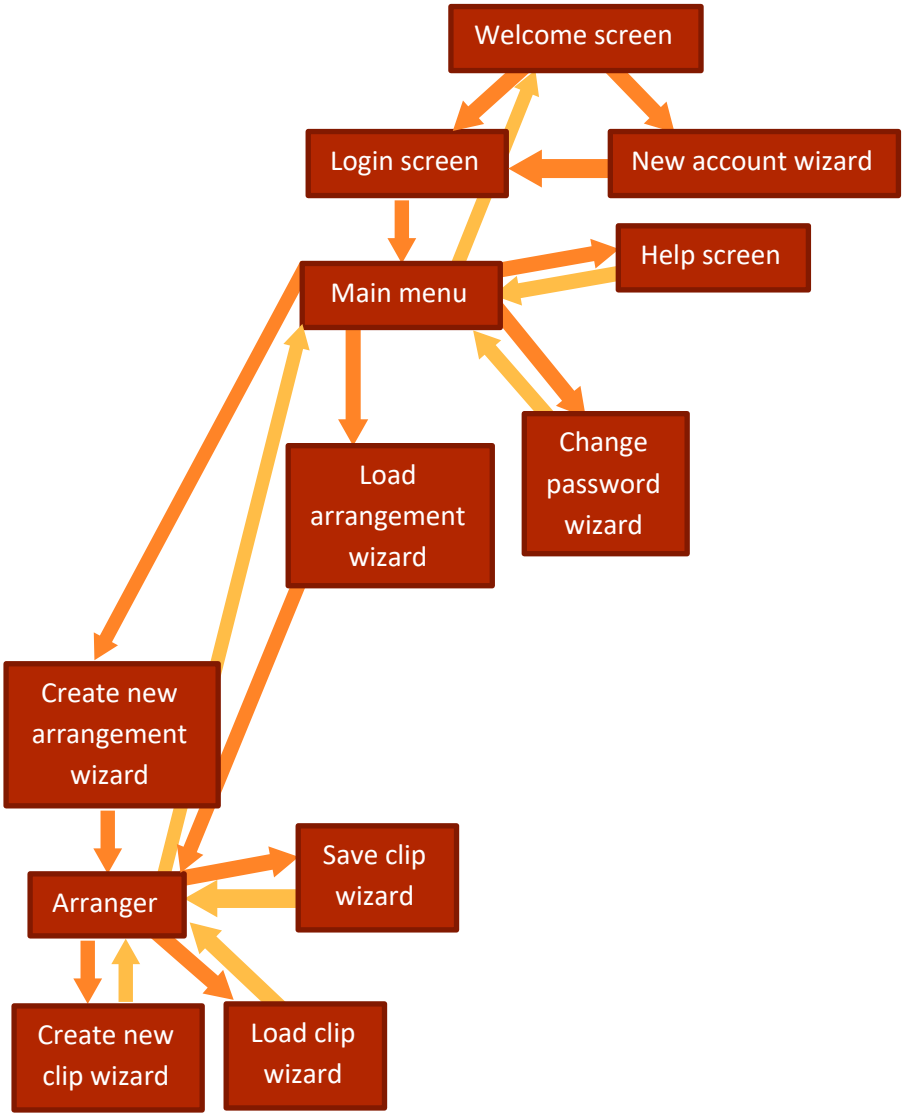
My final solution contains much functionality that was not stated or stated in a limited manner in the Design. The end product was also far more complicated than I anticipated, so I used about 3 times as many methods as originally planned.

In this section, I will be explaining in detail all of the classes and methods used, in particular highlighting code which relates to the note generation algorithm that I went over in the Design.

2 System Overview

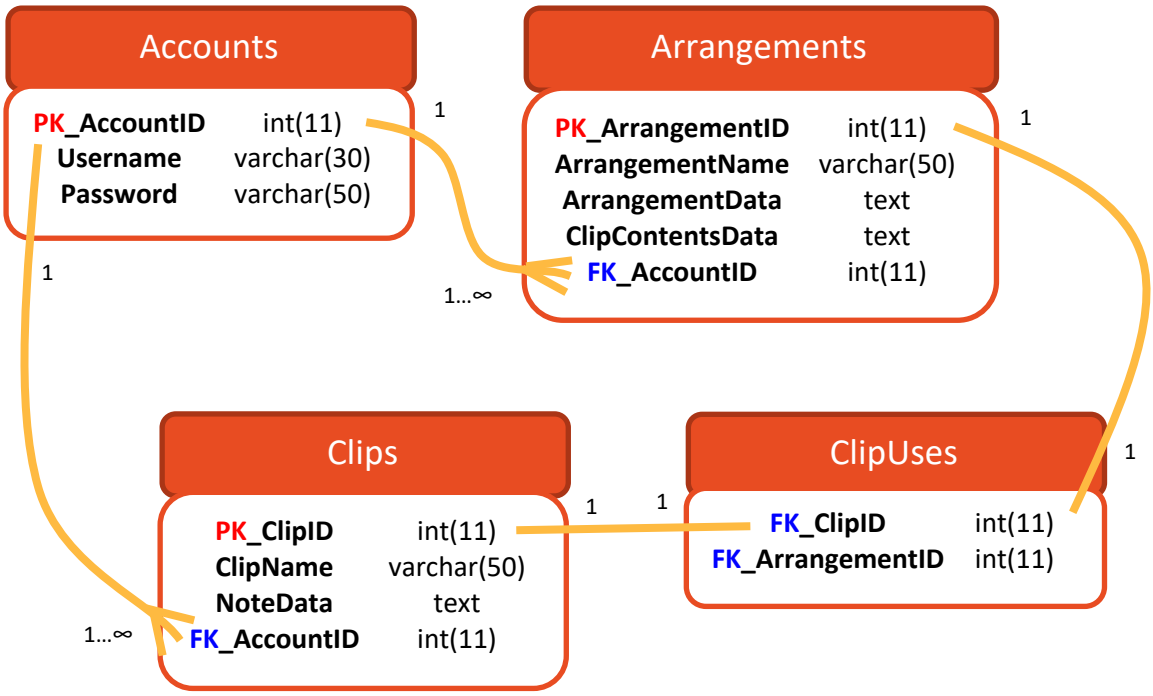
2.1 Local Structure

The solution consists of a single console application which displays a number of different forms. Below is a diagram showing how the user transitions from one form to another. The different paths show the decisions that the user can make on each form.



2.2 Database Structure

The program connects to an EasyPHP server running MySQL. The tables, fields and relationships are as follows. I have listed each of the field names and datatypes. PK means the field is a primary key and FK means it's a foreign key.



Accounts stores the usernames and passwords for the users of the program.

Arrangements holds the data for any arrangements saved by users. It links them to their user account. It can contain note data for unsaved (not given a name) clips and references to note data of saved clips (given a name) in the Clips table. One account can have many arrangements.

Clips contains note data of any individual note clips saved by the user, linking them to their account. One account can have many clips.

ClipUses links any saved clip's use within an arrangement. The two foreign keys form a compound primary key. One record can link only one clip with one arrangement.

3 Encoding Used

Throughout this program, you may see me referring to different codes that represent other values. This section goes over these codes and what they refer to.

3.1 Note Values

Each note has a note value, or length. I have decided to use a single digit to represent the different note values available for each note, so that they can be more easily encoded for the database. When the code is used in the note generation algorithm, it gets converted into a decimal value to give the

actual length of the note. These lengths are used to work out how much space within the clip each note makes up.

Code	Meaning	Decimal Value
0	Semibreve (whole note)	1
1	Dotted minim (3/4 note)	0.75
2	Minim (1/2 note)	0.5
3	Crotchet (1/4 note)	0.25
4	Quaver (1/8 note)	0.125
5	Semiquaver (1/16 note)	0.0625

3.2 Pitch

Each note also has a pitch. Each pitch is normally represented by a letter and accompanying accidental if necessary, such as A, B♭ or C♯. Again, this is not suitable for storing in the database, so I have devised codes for each note's pitch.

This code is two digits. The first digit shows if the note is natural (not sharpened or flattened), sharp or flat.

Code	Meaning
0	Natural
1	Sharp
2	Flat

The second digit represent the main pitch, spanning a whole octave. It can be:

Code	Meaning
1	C
2	D
3	E
4	F
5	G
6	A
7	B

Therefore, a note with the code 01 means C natural and a note with the code 23 means E flat (E♭).

3.3 Serialisation

3.3.1 Overview

Arrangement data and note data are stored in the database as text, but getting that to work was not easy.

Arrangements can be saved using clips that are unsaved, i.e. the notes are only in the arrangement, and saved clips, where the note data is stored in the Clips table and only a reference is made to the clip's ID in the arrangement.

Much of what I go through here is a repeat of what's in the **Technical Solution** so that you don't have to go back to it to understand the processes here.

3.3.2 Arrangement Data

Here is an example of arrangement data stored in the ArrangementData field in the database:

001011C24

The first 0 means tonic note next, followed by 01, which our C code.

The 1 that follows means major/minor next, followed by a C, which means major. If that was a D, it would mean minor.

The 2 means time signature next, followed by a 4, which means we're using a 4/4 time signature.

So far, this is the only data we have, but more will be added once we insert clips.

Next, we look at how clips are inserted.

3.3.3 Storing Unsaved Clips

Here is an example of an arrangement with unsaved clips, taken from the **Technical Solution**:

PK_ArrangementID	ArrangementName	ArrangementData
8	Test Arrangement	001011C24BE1948331075FBE313193004FBE1674691964F

ClipContentsData	FK_AccountID
E1948331075FXA3B03C4YXA4B02C4YXA0B05C4YXA4B02C4YXA...	4

Figure 5.32 *– arrangement record after saving the clips*

First, let's deconstruct the text in the ArrangementData field:

001011C24BE1948331075FBE313193004FBE1674691964F

We can ignore this part, as this is the main arrangement properties (key and time signature) that I went through earlier.

B means the start of an unsaved clip. The fact that there are three Bs shows that there are 3 clips, as there are.

This is a key which makes a reference to the clip's note data in the ClipContentsData field. E means the start of the key and F means the end of the key. Between these two letters is a randomly-generated integer; each time a key is generated, there is a check done to make sure that that integer hasn't been generated already for another key, which means each key is unique.

Now, let's take a look at the ClipContentsData field:

E1948331075FXA3B03C4YXA4B02C4YXA0B05C4YXA4B02C4YXA3B03C4YXA0B05C4YXA2B06C4YXA1B03C4YE313193004FXA4B06C4YXA2B05C4YXA4B02C4YXA1B01C4YXA2B05C4YXA4B01C4YXA1B02C4YXA2B03C4YXA4B06C4YXA3B01C4YXA4B03C4YXA4B03C4YE1674691964FXA1B05C4YXA2B04C4YXA2B05C4YXA4B03C4YXA1B05C4YXA1B06C4YXA2B04C4YXA4B05C4Y

We can see that the three keys that we identified are contained within this text. After each key is the data representing the notes within the clip that the key belongs to. Let's look at the data for the first clip (the text in blue):

XA3B03C4YXA4B02C4YXA0B05C4YXA4B02C4YXA3B03C4YXA0B05C4YXA2B06C4YXA1B03C4Y

The Xs mark the start of a note and the Ys mark the end of a note. The text in black is note properties. Let's look at one of these notes:

XA3B03C4Y

A means that the following number is the code for the note value. 3 is the code for a crotchet (1/4 note).

B means the following two numbers mean the note's pitch. 03 is the code for E natural.

C means the following number represents the octave. This note is in octave 4.

3.3.4 Storing Saved Clips

If our arrangement contains saved clips, the fields look slightly different:

PK_ArrangementID	ArrangementName	ArrangementData
8	Test Arrangement	001011C24BE1948331075FBE313193004FBE1674691964F
9	Test Arrangement 2	0011C24A39A49BE304842317F

ClipContentsData	FK_AccountID
E1948331075FXA3B03C4YXA4B02C4YXA0B05C4YXA4B02C4YXA...	4
E304842317FXA1B02C4YXA4B04C4YXA0B01C4YXA1B05C4YXA4...	4

Figure 5.41 *– arrangements table after saving the arrangement with saved clips*

The first row is for the first arrangement that we made earlier. Let's focus on the ArrangementData field for the second one:

0011C24A39A49BE304842317F

We can see that the field contains the properties of the arrangement, and the key for the 3rd clip that was unsaved, but for the first two clips, we have something different.

A means the start of a saved clip. The number after is the compound primary key of the saved clip in the ClipUses table.

Let's look at the ClipUses table:

FK_ClipID	FK_ArrangementID
2	8
3	9
4	9

Figure 5.42 *– ClipUses table*

A39 refers to the second clip in this table.

A49 refers to the third clip.

The clip ID is then looked up and the note data is loaded from the Clips table.

It is set up like this rather than the reference being straight to the PK_ClipID value, because otherwise, we would have a many-to-many relationship.

3.3.5 Marking Out the Position of Each Note

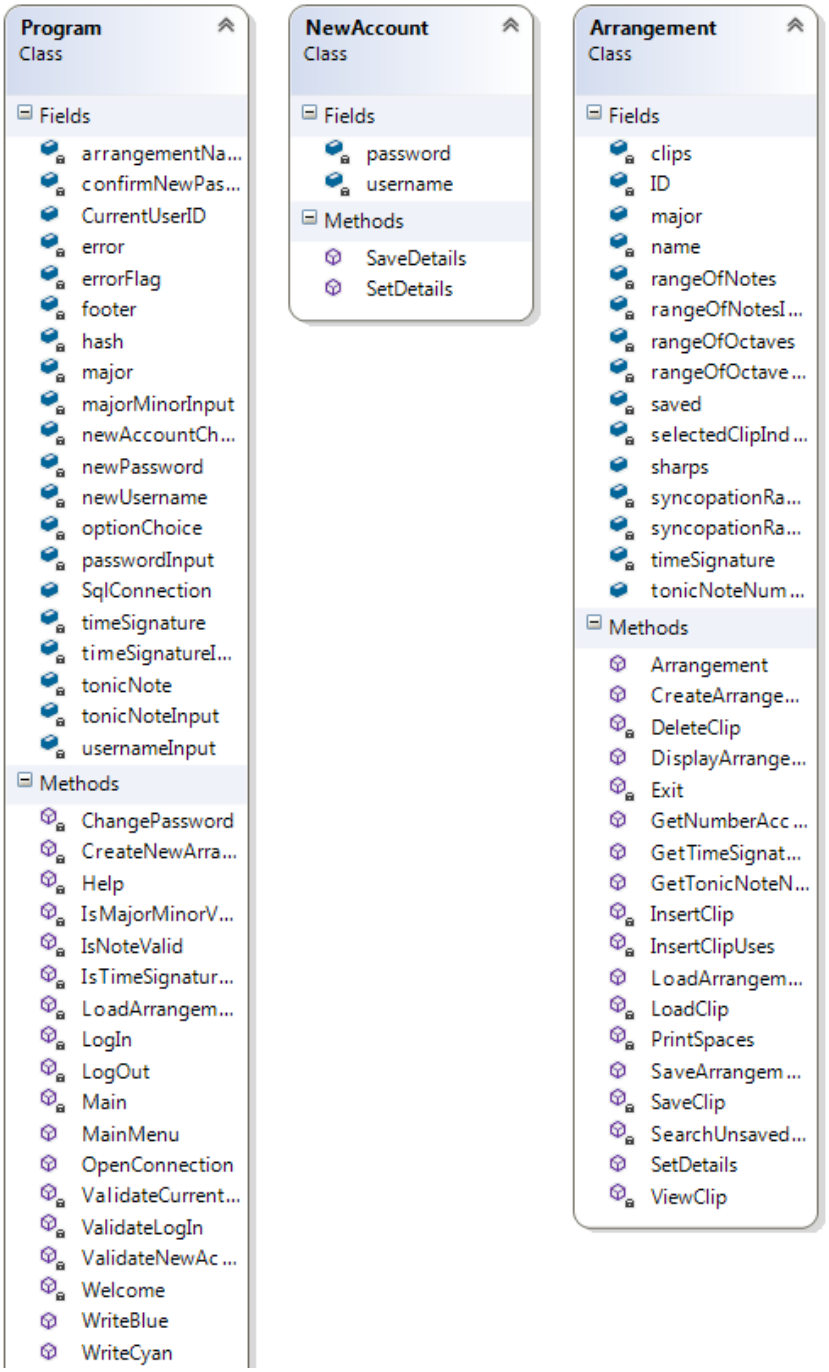
In order to display the notes, the position of each note must be marked out in a series of strings. Each string represents one note. The strings are then processed in order to display the correct colours on the screen according to the characters within the string.

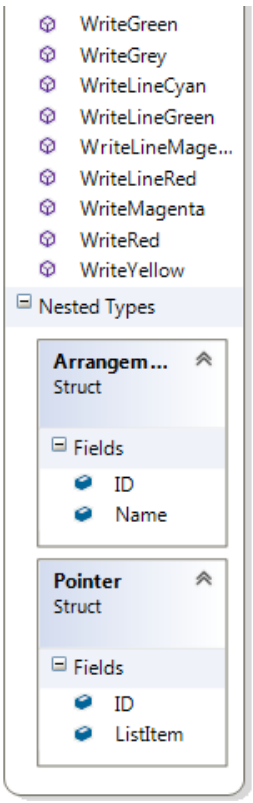
Within each string can contain the following characters:

- Space – there is no note located here.
- # - there is a natural note located here.
- / - there is a sharp note located here.
- * - there is a flat note located here.

4 Class Descriptions

4.1 Class Diagram




WriteGreen
WriteGrey
WriteLineCyan
WriteLineGreen
WriteLineMage...
WriteLineRed
WriteMagenta
WriteRed
WriteYellow
Nested Types
Arrangem...
Struct
Fields
ID
Name
Pointer
Struct
Fields
ID
ListItem

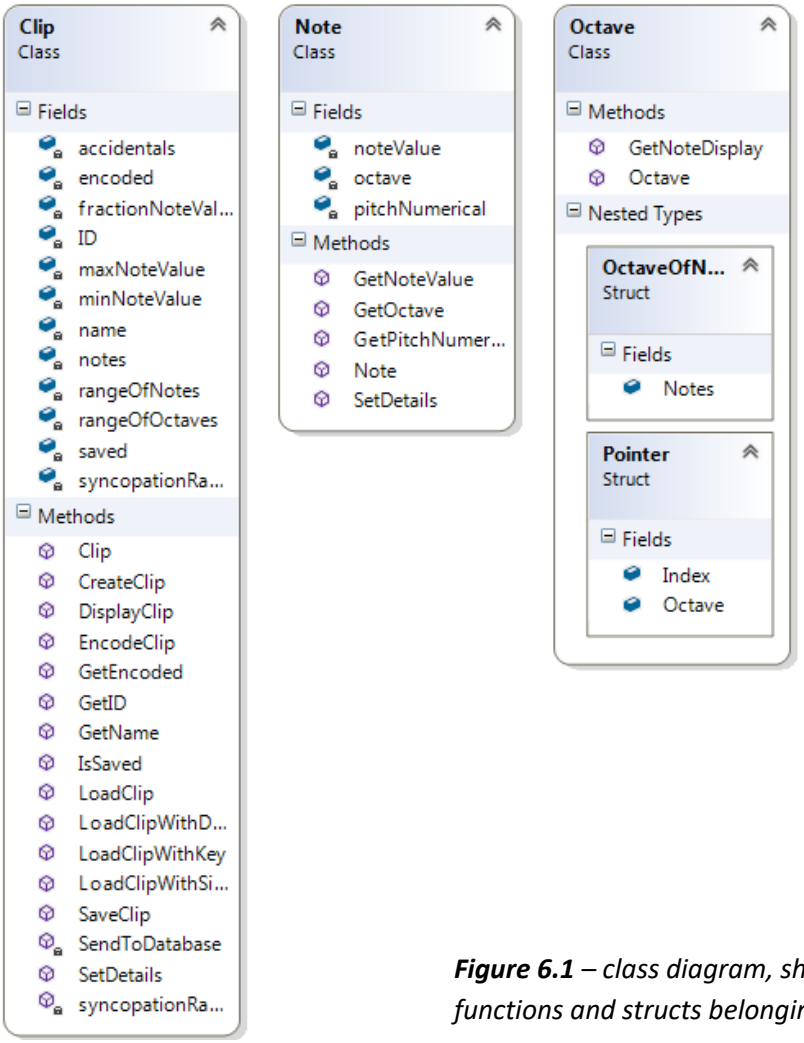


Figure 6.1 *– class diagram, showing all variables, methods, functions and structs belonging to each class in the program*

4.2 Class Analysis

In this section, I will be reminding us of the variables, methods and functions that we intended to include in each class from the **Design** section on page 39 (with the heading 'Designed'), followed by what extra ones I included (with the heading 'New'). I have given each item an ID number. Those which were designed start with D and those which are new start with N.

The variables, methods and functions are listed in dictionaries. Most of these are in alphabetical order, but I have changed the order of some of them, as they require knowledge of previous variables to make sense.

For each variable, method or function, I have made cross-references to the page location of the source code in which they are used, which is located in the **Appendix** section (page 187) towards the back of the document.

I have listed some items in both the Designed and New sections. This is because I changed the name, datatype or scope. Where this occurs, I have included a cross-reference to the original item ID and a justification as to why I changed the property. I will not include a page reference for each original item that's been changed, but will include one instead for the updated version.

3.2.1 Program Class

3.2.1.1 Overview

This handles the initial user inputs, such as the welcome page, new account wizard and main menu items. It also contains the SQL connection and the methods that print to the console in different colours.

3.2.1.2 Variable Dictionary

3.2.1.2.1 Designed

ID	Name	Datatype	Scope	Purpose	Page in Appendix
D1	SqlConnection	MySql Connection	public	The object allowing for connection to the database. This is public as it is accessed in all classes.	188
D2	CurrentUserID	int	public	The ID of the current user account being accessed.	188
D3	newAccountChoice	console key	private	When logging in, if the first key that the user inputs is F1, this is a prompt to the new account screen. Without this variable, the new account screen can't be shown.	188
D4	usernameInput	string	private	Used for logging in.	189
D5	passwordInput	string	private	Used for logging in and checking the current password for changing a password.	189
D6	newUsername	string	private	Used for creating a new account.	189
D7	newPassword	string	private	Used for creating a new account.	189
D8	confirmNewPassword	string	private	Used for creating a new account, where password needs to be entered again for validation.	189
D9	optionChoice	console key	private	Given by user on the main menu.	189
D10	tonicNoteInput	string	private	Used for creating a new arrangement. Inputs key note.	189
D11	majorMinorInput	string	private	Used for creating a new arrangement. Inputs whether the key is major or minor.	189
D12	timeSignature	int	private	Used for creating a new arrangement. User inputs number of beats in a bar.	189

D13	arrangementIDChoice	int	private	Used when loading an arrangement, where user chooses ID number of one they'd like to load. *This was renamed to 'choice'*.	201
D14	oldPassword	string	private	Used for changing password, where existing password is validated. *This was not used.*	*N/A*
D15	newPassword	string	private	Used for changing password, where new password is entered.	189
D16	confirmNewPassword	string	private	Used for changing password, where new password is confirmed.	189
D17	rangeOfOctaves	int	private	Used for creating a new clip – user inputs range of octaves for the clip to include. *This was moved to the Clip class.*	*(See **D35**)*
D18	rangeOfNotes	int	private	Used for creating a new clip – user inputs. *This was moved to the Clip class.*	*(See **D36**)*
D19	syncopation Randomisation	int	private	Used for creating a new clip – user enters a score from 1 to 10 based on how much they'd like their syncopation to change. See **Section 3.3** for more info on what this means. *This was moved to the Clip class.*	*(See **D37**)*
D20	clipIDChoice	int	private	Used for loading a clip – user chooses ID of clip they want to load. *This was renamed to 'choice'.*	219

3.2.1.2.2 New

ID	Variable Name	Datatype	Scope	Purpose	Page in Appendix
N1	arrangementName Input	string	private	Takes input from the user for naming an arrangement.	189
N2	error	string	private	This is by default blank. When an error occurs on a form which uses errors (such as the create new arrangement wizard), the error is stored in this variable. It is always displayed on these forms, even when there's no error.	189
N3	footer	string	private	Stores the current value to be	189

				displayed in the footer, which is shown on some of the pages. It normally shows the current username logged in but does display errors when required.	
N4	errorFlag	bool	private	Used to mark where an error has occurred on most of the forms, so that the footer is displayed in red.	189
N5	hash	Hash Algorithm	private	Object that contains the SHA-256 hash function needed for the password.	189
N6	major	bool	private	When creating an arrangement, we need to mark if the key signature is major or minor (major is true, minor false).	189
N7	timeSignatureInput	string	private	Essentially the same as **D12**, but I've created an extra variable for the input so that it can be validated before being stored as an actual value. It's string so that we can validate any erroneous data that gets input without crashing.	189
N8	timeSignature	short	private	Stores the valid time signature value for when we create a new arrangement.	189
N9	tonicNote	string	private	Stores the tonic note value for creating a new arrangement. It uses the value from **D10** once it's been validated.	189

3.2.1.3 Method/Function Dictionary

3.2.1.3.1 Designed

I did not design any functions or methods for the Program class.

3.2.1.3.2 New

ID	Name	Return Type	Scope	Purpose	Page in Appendix
N10	ChangePassword	void	private	Shows the change password wizard, where the user can change their password and the Accounts table is updated with a new hash of their input.	202
N11	CreateNewArrangement	void	private	Shows the create new arrangement wizard. The user enters properties and an arrangement object and new record in the Arrangements table are created.	194

N12	Help	void	private	Shows the help screen.	197
N13	IsMajorMinorValid	bool	private	Returns if the user's input for major/minor is valid or not.	195
N14	IsNoteValid	bool	private	Returns if the user's input for the tonic note is valid or not. If it is valid, it converts the input note into its code form and stores it in the tonicNote variable.	196
N15	IsTimeSignatureValid	bool	private	Returns if the user's input for time signature is valid or not.	195
N16	LoadArrangement	void	private	Shows the load arrangement wizard – the names and IDs of all of the arrangement's belonging to the user are selected from the Arrangements table. The user chooses and the selected arrangement's data is retrieved, and a new arrangement object is created.	200
N17	ValidateLogIn	void	private	Shows the login screen and takes input from the user for username and password.	191
N18	LogIn	void	private	Checks the user's input information from the Accounts table. If it's correct, the user is shown the main menu. If not, ValidateLogIn is run again with an error message.	199
N19	LogOut	void	private	From the main menu, this method clears the CurrentUserID and runs the welcome method.	197
N20	Main	void	private	The first method to be run in the class. SqlConnection and hash are instantiated and the Welcome method is called.	189
N21	MainMenu	void	public	Displays the main menu and takes input from the user for the action to happen next.	192
N22	OpenConnection	void	public	Attempts to open SqlConnection. If it's already open, close the existing connection and open a new one.	200
N23	ValidateCurrentPassword	bool	private	When changing a password, this returns if the user's input for their current password is true or not.	204
N24	Welcome	void	private	Displays the welcome screen and gives the user the option to create a new account or log in.	189
N25	WriteBlue	void	public	Writes to the console in blue (all of the writing methods here change the colour back to white afterwards).	198

N26	WriteCyan	void	public	Writes in cyan.	199
N27	WriteGreen	void	public	Writes in green.	198
N28	WriteGrey	void	public	Writes in grey.	198
N30	WriteLineCyan	void	public	Writes a line in cyan.	198
N31	WriteLineGreen	void	public	Writes a line in green.	199
N32	WriteLineMagenta	void	public	Writes a line in magenta.	198
N33	WriteLineRed	void	public	Writes a line in red.	198
N34	WriteMagenta	void	public	Writes in magenta.	198
N35	WriteRed	void	public	Writes in red.	198
N36	WriteYellow	void	public	Writes in yellow.	198

3.2.1.4 Struct Dictionary

- ArrangementRow – when displaying the list of arrangements to choose from when the user uses the load arrangement wizard, this is used to store each row of the list, the ID representing the number in the list and the name holding the name of the arrangement.
- Pointer – for each arrangement row in the load wizard list, a pointer links the arrangement's number in the list (ListItem) with its PK_ArrangementID in the database (ID).

3.2.2 NewAccount Class

3.2.2.1 Overview

This class is responsible for storing properties of a new account and inserting a new record with the properties into the Accounts table.

3.2.2.2 Variable Dictionary

3.2.2.2.1 Designed

ID	Variable Name	Datatype	Scope	Purpose	Page in Appendix
D21	password	string	private	For storing new username to be saved.	204
D22	username	string	private	For storing new password to be saved.	204

3.2.2.2.2 New

There are no new variables in this class.

3.2.2.3 Method Dictionary

3.2.2.3.1 Designed

ID	Name	Return Type	Scope	Purpose	Page in Appendix
D23	SetDetails	void	public	Stores values in username and password using input from Program class.	204
D24	SaveDetails	bool	public	Inserts a new record into the Accounts table using values in username and password.	205

3.2.2.3.2 New

There are no new methods or functions in this class.

3.2.3 Arrangement Class

3.2.3.1 Overview

This is the template for an arrangement object. It aggregates a list of clip objects and contains the necessary methods for creating, loading clips from the database, saving the arrangement and deleting clips.

3.2.3.1 Variable Dictionary

3.2.3.1.1 Designed

ID	Variable	Datatype	Scope	Purpose	Page in Appendix
D25	clips	list of Clip objects	protected	Contains the clips within an arrangement.	*(See **N37**)*
D26	saved	bool	protected	To flag whether or not the arrangement has been saved.	*(See **N45**)*
D27	tonicNoteNumerical	int	protected	The numerical equivalent of the string tonic note which was input by the user. I am using numbers as they are easier to process than characters.	*(See **N50**)*
D28	majorOrMinor	bool	protected	Determines if the key is major or minor.	*(See **N39**)*
D29	timeSignature	int	protected	Determines the number of beats in a bar.	*(See **N49**)*

3.2.3.1.2 New

ID	Name	Datatype	Scope	Purpose	Page in Appendix
N37	clips	list of Clip objects	private	Same as **D25** but is private instead of protected (there's no need for it to be protected as there's no inheritance).	205
N38	ID	int	private	The PK_ArrangementID value of the arrangement in the Arrangements table in the database. This is always present for an arrangement object as it always has a record in the Arrangements table.	206
N39	major	bool	public	Same as **D28**, but it needs to be public as it is accessed by the Clips class.	205
N40	name	string	private	Stores the name of the arrangement.	205

N41	rangeOfNotes	int	private	Stores the range of notes property for a clip that's about to be created (range of notes is the number of notes within an octave allowed to be generated).	206
N42	rangeOfNotes Input	string	private	Takes the input from the user for range of notes to be converted to the int above.	206
N43	rangeOfOctaves	int	private	Stores the range of octaves property for a new clip (this is the range of octaves allowed for the notes, where the lower limit is 4 – the input, and the upper limit is 4 + the input).	206
N44	rangeofOctaves Input	string	private	Takes the input from the user for the range of octaves above.	206
N45	saved	bool	private	Same as **D26** but private instead of protected.	206
N46	selectedClip Index	int	private	Stores an index for the clips list to mark which clip in the arranger is currently selected by the user.	206
N47	sharps	bool	public	Stores if the key signature uses sharps (true) or flats (false). It is public because it is used in the Octave class. Originally, this was going to be its own function (**D34**), but instead, the value is set using the GetNumberAccidentals function (**N54**).	205
N48	syncopation Randomisation	int	private	Stores the syncopation randomisation property for a new clip (this is how often the note value of each next note being generated should be changed. I will show how this works later).	206
	syncopation Randomisation Input	string	private	Takes the input from the user for the variable above.	206
N49	timeSignature	int	private	Same as **D29** but private instead of protected.	206
N50	tonicNote Numerical	string	public	Same as **D27** but public instead of protected (it is used in the Clip class) and string instead of int, as the tonic note codes can start with 0s.	205

3.2.3.2 Method/Function Dictionary

3.2.3.2.1 Designed

ID	Name	Return Type	Scope	Purpose	Page in Appendix
D30	SetDetails	void	public	Stores values for tonic note, major/minor and time signature using inputs.	206
D31	CreateArrangement	void	public	Creates an arrangement object taking input from the user and inserts a new record into the Arrangements table using the data.	220
D32	LoadArrangement	void	public	Selects arrangement data from the Arrangements table into the object using a given PK_ArrangementID value chosen by the user.	213
D33	SaveArrangement	void	public	Serialises the arrangement object's clip data and updates the record in the Arrangements table.	220
D34	Sharps	bool	public	Returns whether the key signature uses sharps or not (flats). *This was not used.*	*(See **N47**)*

2.2.3.2.2 New

ID	Name	Return Type	Scope	Description	Page in Appendix
N51	Arrangement	N/A	public	Constructor for the Arrangement object.	206
N52	DeleteClip	void	private	Deletes the clip currently selected by the user from the list.	219
N53	Display Arrangement	void	public	Displays the arranger: all of the clips in the list and the options. It prompts the user for a key input for what to perform next.	206
	Exit	void	private	Exits the current arrangement being worked on and the main menu is shown.	213
N54	GetNumber Accidentals	int	public	This returns the number of accidentals (sharps or flats) in the arrangement's key signature. It's public because it's used by the Clips class.	211
N55	GetTime Signature	int	public	Returns the time signature of the arrangement.	213
N56	GetTonicNote Numerical	string	public	Returns the tonic note code of the arrangement.	211
N57	InsertClip	void	private	Takes input from the user for properties for a new clip, creates and sets up a Clip object and adds it to the clip list.	209

N58	InsertClipUses	void	private	Inserts a new record into the ClipUses table using the clip ID and arrangement ID values.	218
N59	LoadClip	void	private	Takes user input for choice of clip to load in the load clip wizard, and selects the clip data for a clip using a selected clip ID, creates and sets up a new clip object and adds it to the clip list.	218
N60	PrintSpaces	void	private	Prints the right number of spaces onto the screen when displaying the arranger.	209
N61	SaveClip	void	private	Takes user input for the name for the selected clip, serialises a clip's note data and adds it to a new record to the Clips table, also using the InsertClipUses method to update the ClipUses table.	217
N62	SearchUnsaved Clip	void	private	Searches an unsaved clip's note data for data about a particular clip, using a given key.	216
N63	ViewClip	void	private	Call the select clip to display its notes.	213

3.2.4 Clip Class

3.2.4.1 Overview

This is the template for a clip object. It aggregates a list of note objects and contains methods for generating notes, displaying the notes to the user, saving the notes to the database and loading notes from the database using serialised note data.

3.2.4.2 Variable Dictionary

3.2.4.2.1 Designed

ID	Name	Datatype	Scope	Purpose	Page in Appendix
D35	rangeOfOctaves	int	private	Stores the range of octaves chosen by the user.	222
D36	rangeOfNotes	int	private	Stores the range of notes within the octave chosen by the user.	222
D37	syncopation Randomisation	int	private	Stores the randomisation factor for the syncopation of the notes, chosen by the user. See **Section 3.3** for more on what this means.	222
D38	notes	list of Note objects	private	Stores the notes within the clip.	222
D39	accidentals	int[]	private	Stores the numerical values of the notes which need to be changed depending on how many notes there are in the key signature.	*(See **N64**)*

3.2.4.2.2 New

ID	Name	Datatype	Scope	Purpose	Page in Appendix
N64	accidentals	short[]	private	Same as **D39** but uses the short datatype (16-bit integer), as the numbers it stores are small.	222
N65	encoded	string	private	Stores the serialised note data when the clip is about to be saved to the database.	222
N66	fractionNoteValues	double[]	private	Stores the different note values that a note can use, ranging from a whole note to a 1/16 note. Each note value is represented as a decimal, so an 1/8 note is 0.125.	222
N67	ID	int	private	If a clip is saved, then it is assigned a PK_ClipID value in the Clips table. This variable stores that value.	222
N68	maxNoteValue	short	private	Stores the maximum index of **N66** (highest note value) that the clip is allowed to contain.	223
N69	minNoteValue	short	private	Stores the minimum index of **N66** (lowest note value) that the clip is allowed to contain.	223
N70	name	string	private	If the clip is saved, it is assigned a name by the user. This variable stores said name.	222
N71	saved	bool	private	Marks if the clip has been saved in the Clips table or not.	222

3.2.4.3 Method/Function Dictionary

3.2.4.3.1 Designed

ID	Name	Return Type	Scope	Purpose	Page in Appendix
D40	syncopationRandomiser	bool	private	Decides whether or not the note value needs changing.	226
D41	SetDetails	void	public	Sets the values of the range of octaves, range of notes and syncopation randomisation factor, using given inputs.	223
D42	CreateClip	void	public	Selects pitches, note values and octaves at random, checks if they are valid according to the arrangement's properties, then instantiates note objects, which are added to the Notes list (**D38)**.	224
D43	LoadClip	void	public	Loads note data from the Clips table in the database into the clip object for processing into	232

				individual clips.	
D44	DisplayClip	void	public	Uses the note data to display the notes onto the screen in a readable format.	228
D45	SaveClip	void	public	Save the clip to the Clips table using a given name.	227

3.2.4.3.2 New

ID	Name	Return Type	Scope	Purpose	Page in Appendix
N72	Clip	N/A	public	Constructor for the clip object.	223
N73	EncodeClip	void	public	Serialises the note data and stores it in the encoded string (**N65**).	227
N74	GetEncoded	string	public	Returns the encoded string (**N65**) to another function.	228
N75	GetID	int	public	Returns the ID value (**N67**) of the clip.	234
N76	GetName	string	public	Returns the name of the clip (**N70**).	223
N77	IsSaved	bool	public	Returns whether or not the clip is saved (using **N71**).	223
N78	LoadClipWithData	void	public	Takes the serialised note data that's been loaded from the database and creates individual notes from it. This is used for both saved and unsaved clips.	229
N79	LoadClipWithKey	void	public	Loads serialised note data from the Clips table using a given compound primary key. This key looks up a record in the ClipUses table, which links the clip's PK_ClipID to the PK_ArrangementID of the arrangement in which it is used.	232
N80	LoadClipWithSingleKey	void	public	Does the same as **N79**, but uses just the PK_ClipID value to find the clip rather than a compound key.	233
N81	SendToDatabase	void	private	Sends the serialised note data (**N65**) to the Clips table.	233

3.2.5 Note Class

3.2.5.1 Overview

This is a template for the note object. It holds the pitch, octave and length (note value) of each note and has no behaviour other than access modifiers.

3.2.5.2 Variable Dictionary

3.2.5.2.1 Designed

ID	Name	Datatype	Scope	Purpose	Page in Appendix
D46	isRest	bool	private	Stores whether or not the note object represents a note or a rest. If it's true, it's a rest. If false, it's a note. *This was not used as I didn't end up including functionality for rests.*	*N/A*
D47	pitchNumerical	int	private	The numerical value of the note's pitch.	234
D48	noteValue	int	private	The number code for the note value (length) of the note.	234

3.2.5.2.2 New

ID	Name	Datatype	Scope	Purpose	Page in Appendix
N82	octave	int	private	To store the octave of the note.	234

3.2.5.3 Method/Function Dictionary

3.2.5.2.1 Designed

ID	Name	Return Type	Scope	Purpose	Page in Appendix
D49	SetDetails	void	public	Sets the values of the pitch, note value and octave.	234
D50	GetStringNoteValue	string	public	Return the note value of the note as a string.	*(See **N84**)*

3.2.5.2.2 New

ID	Name	Return Type	Scope	Purpose	Page in Appendix
N83	Note	N/A	public	Constructor for Note object.	234
N84	GetNoteValue	int	public	The same as **D50**, but returns the note value code as an integer rather than a string. There is no need to convert the numerical note value into a string.	234
N85	GetOctave	int	public	Returns the octave of the note.	235
N86	GetPitchNumerical	string	public	Returns the pitch code of the note.	234

3.2.6 Octave Class

3.2.6.1 Overview

This contains the functions for formatting the note objects of a clip into a series of serialised strings that represent each octave, before they are displayed to the user in the clip viewer.

This class was not planned and was only added to separate the note displaying code away from the rest of the program. Hence, no variables or methods were designed.

In addition, no class-level variables were even used.

3.2.6.2 Method/Function Dictionary

ID	Name	Return Type	Scope	Purpose	Page in Appendix
N87	Octave	N/A	public	Constructor for the Octave object.	235
N88	GetNoteDisplay	string	public	Returns a mapping string showing where the notes are located within the clip. This is sent to the Clip class, where it is processed and the notes are displayed.	235

4.2.6.3 Struct Dictionary

- OctaveOfNotes – this contains an array of 7 strings. Each string represents a note within an octave. The position of each note is marked out using the characters as described in **Section 2.3.5 (page 155)**.
- Pointer – this is used to assign indices to the different octaves contained in a clip, so that they can be stored in an array.

5 Algorithms

In my program, I used one standard algorithm and two custom algorithms.

This section shows pieces of code within my program where these algorithms are used, with comments.

Only one of the algorithms out of these 3 were designed, and that is the note generation algorithm (5.2).

5.1 SHA-256 Hashing

In order to hash the user's password, both when creating a new account, logging into an existing one, or when they are changing the password, I make use of the SHA-256 hash function within C#. This is on page 191:

```
// The password (newPassword) is converted into an array of pure unicode bytes
byte[] plainTextBytes = System.Text.Encoding.UTF8.GetBytes(newPassword);
// hash is the object containing hash functions. We create another array of bytes
where we take our original array and perform the hash function for all the bytes
byte[] hashBytes = hash.ComputeHash(plainTextBytes);
// We convert the bytes into a string
string hashedPassword = Convert.ToBase64String(hashBytes);
```

5.2 Note Generation

Within the Clip class is one of the most important parts of the program; the part that generates the notes and adds them to the Notes list.

In the **Design Section 3.3.2 (page 57)**, I designed this algorithm using the following pseudocode:

```
private int: rangeOfOctaves
private int: rangeOfNotes
private int: syncopationRandomisation
private list (of type Note): notes
private static int[] accidentals = {4, 1, 5, 2, 6, 3, 7}
private static

public method Constructor {No parameters}

public method SetDetails {Parameters = (int: RangeOfOctaves, int: RangeOfNotes, int:
SyncopationRandomisation)}:
        rangeOfOctaves ← RangeOfOctaves
        rangeOfNotes ← rangeOfNotes
        syncopationRandomisation ← SyncopationRandomisation

public method CreateClip {No parameters}:
        int[]: octaveRangeAvailable ← {4 - rangeOfOctaves, 4 + rangeOfOctaves}
        int: highestNoteAvailable
        if Arrangement.tonicNoteNumerical = 1
                highestNoteAvailable ← 7
        else:
                highestNoteAvailable ← Arrangement.tonicNoteNumerical - 1
        int: numberAccidentals ← Number of accidentals in key signature
        bool: sharpen
        int: previousNoteValue
        static float[]: fractionNoteValues ← [1, 3/4, 1/2, 1/4, 1/8, 1/16]
        int[]: accidentalsToApply[numberAccidentals]
        if Arrangement.Sharps() = true:
                sharpen ← true
                for (int i ← 0 to (numberAccidentals – 1)):
                        accidentalsToApply[i] ← accidentals[i]
        else (key contains flats):
                sharpen ← false
                for (int i ← 6 (decreasing) to (7 – numberAccidentals)):
                        accidentalsToApply[i] ← accidentals[i]
        float: spaceLeft ←  Arrangement.timeSignature * 1/4
        infinite loop:
                int: pitch ← Random number between 1 and 7
                int: octave ← Random number between 1 and 10
                int: noteValue
                if syncopationRandomiser(syncopationRandomisation) = true:
                        noteValue ← Random number between 0 and 5
                else:
                        noteValue ← previousNoteValue
                if spaceLeft >= fractionNoteValues[noteValue]:
                        if (octave >= octaveRangeAvailable[0]) & (octave <=
                        octaveRangeAvailable[1]):
                                if (pitch >= Arrangement.tonicNoteNumerical) & (pitch <=
                                highestNoteAvailable):
                                        if accidentalToApply contains pitch:
```

```
                                        if sharpen = true:
                                            pitch ← pitch + 10
                                        else:
                                            pitch ← pitch + 20
                                    previousNoteValue ← noteValue
                                    notes ← New note
                                    New note .SetDetails(pitch, noteValue)
                                    spaceLeft ← spaceLeft – fractionNoteValues[noteValue]
                                else:
                                    break
                            else:
                                break

                    else:

                        break
```

```
private bool function syncopationRandomiser {int: syncopationRandomisation}:

    int: randomNumber ← Random integer from 1 to syncopationRandomisation:

        if randomNumber = 1:
            return true
        else:
            return false
```

It works by selecting note values, pitches and octaves at random, and seeing if they fit in the range allowed by the properties in the arrangement and clip, which are given by the user.

If they are valid, then a new note object is created and added to the note list. If they are not, the process starts again.

This is the relevant code in the Clip class (page 224):

```
public void CreateClip(ref bool sharps, ref bool major)

{
    // This creates an array storing two values - the lowest octave available is 4
minus the range of octaves input, and the highest octave available is 4 plus the range
of octaves
    // So if the range of octaves is 1, the lowest octave is 3 and the highest is 5
    int[] octaveRangeAvailable = { 4 - rangeOfOctaves, 4 + rangeOfOctaves };
    // This variable is used for the note value of the highest note available
    int highestNoteAvailable;
    // Copy the tonic note (the main note without any accidentals) from the
Arrangement class
    string mainTonicNote = Arrangement.tonicNoteNumerical[1].ToString();
    // This varible takes the main tonic note (if the tonic note is a C#, the main
tonic note is a C) as an integer so it can be compared
    int TonicNoteNumerical = Convert.ToInt32(mainTonicNote);
    // If the tonic note is a C, make the highest note a B
    if (TonicNoteNumerical == 1)
    {
        highestNoteAvailable = 7;
    }
    // If it's not a C, make the highest note the note below the tonic note
    // So if the tonic note is an E (3), the highest note available is a D (2)
```

```
    else
    {
        highestNoteAvailable = TonicNoteNumerical - 1;
    }
    // This variable gets the number of accidentals of the key signature by checking
how many there are in the arrangement's key signature
    int numberAccidentals = Arrangement.GetNumberAccidentals(ref sharps, ref major);
    Random randomGenerator = new Random();
    // The note value to use (note length) is stored and is changed if the syncopation
randomiser function (which is further below) decides if it needs changing
    int previousNoteValue = randomGenerator.Next(minNoteValue, maxNoteValue);
    // This array will hold the pitch values of the notes that need sharpening or
flattening
    int[] accidentalsToApply = new int[numberAccidentals];
    // If the key signature contains sharps, apply sharp accidentals
    // This is done by adding the accidental pitch values to the accidentalsToApply
array    in forwards order (as that is how sharps are applied in key signatures)
    if (sharps)
    {
        for (int i = 0; i < numberAccidentals; i++)
        {
            accidentalsToApply[i] = accidentals[i];
        }
    }
    // Otherwise, if the key contains flats, apply flat accidentals
    // This is done by adding the accidental pitch values to the accidentalsToApply
array in reverse order
    else
    {
        short counter = 0;
        for (int i = 6; i >= 7 - numberAccidentals; i--)
        {
            accidentalsToApply[counter] = accidentals[i];
            counter++;
        }
    }
    // This variable stores the amount of space left for the notes
    // The intial value is the number of the key signature (if the key signature is 4,
then the space left is 4)
    double spaceLeft = Arrangement.GetTimeSignature();
    // The spaceLeft is taken away using the decimal note values each time a note is
added
    double spaceToTakeAway;
    // These variables are used as properties for a new note
    int pitch;
    int octave;
    int noteValue;
   string pitchString;
    // This is the note object variable that we will be using to create a new note and
add it to the list
    Note newNote;
    // This loop will decide the properties for a new note
    // It will then check if the note is valid according to the arrangement's key and
time signature
    // If it is invalid, the process starts again
    // If it is valid, the new note is added to the clip list
    do
    {
        // A pitch is randomly selected, from the note C (1) up to B (7)
        pitch = randomGenerator.Next(1, 7);
        // The octave is randomly selected from 1 to 10
        octave = randomGenerator.Next(1, 10);
```

```
        // If the syncopation randomiser decides that the note value needs changing
        if (syncopationRandomiser())
        {
            // Change the note value
            noteValue = randomGenerator.Next(minNoteValue, maxNoteValue);
        }
        // If it decides not
        else
        {
            // Use the note value of the previous note
            noteValue = previousNoteValue;
        }
        // Adding of the note will only be processed further if there is enough space
left, using the spaceLeft variable to check
        if (spaceLeft >= fractionNoteValues[noteValue])
        {
            // It will only go further if the octave is greater or equal to the lowest
octave available, and less than or equal to the highest octave available
            if ((octave >= octaveRangeAvailable[0]) && (octave <=
octaveRangeAvailable[1]))
            {
                // If the note generated needs sharpening or flattening according to
the key signature
                if (accidentalsToApply.Contains(pitch))
                {
                    // If the key signature uses sharps, a 1 is added to the start of
the pitch to show that it is sharp
                    if (sharps)
                    {
                        pitch = pitch + 10;
                    }
                    // If the key signature uses flats, a 2 is added to show the pitch
is flat
                    else
                    {
                        pitch = pitch + 20;
                    }

                }
                // Once note's properties are decided, add it to the notes list
                // If the note is natural, then a 0 needs to be added to the end, as
the integer value doesn't start with 0
                if (pitch - 10 < 0)
                {
                    pitchString = "0" + pitch.ToString();
                }
                // Otherwise, the value is just converted
                else
                {
                    pitchString = pitch.ToString();
                }
                // We store the note value as the previous note value so that we can
use it for the next note
                previousNoteValue = noteValue;
                // A new note object is created and the properties are set
                newNote = new Note();
                newNote.SetDetails(pitchString, noteValue, octave);
                // We add this new note object to the note list
                notes.Add(newNote);
                // Reduce amount of space left so that new notes can be made
                spaceToTakeAway = fractionNoteValues[noteValue];
                spaceLeft = spaceLeft - spaceToTakeAway;
```

```
            }
        }
    } while (spaceLeft > 0);

}

// This function decides if the note value should be changed
private bool syncopationRandomiser()
{
    Random random = new Random();
    // A random number is generated between 1 and the syncopation randomisation value
that the user chose
    int randomNumber = random.Next(1, syncopationRandomisation);
    // If the number is one, then return true
    if (randomNumber == 1)
    {
        return true;
    }
    // If not, return false
    else
    {
        return false;
    }
}
```

5.3 Displaying the Notes

This part is spanned across two different classes: the Clip class and the Octave class.

This part serialises the clip's notes into a single string which maps out the position of the note using characters (see **Section 2.3.5 (page 155)** for a key to the characters used). Then, each of the characters is searched and the console prints out different spaces of colour depending on what character is currently selected.

This is the relevant code (page 228):

5.3.1 In the Clip Class

```
public void DisplayClip(ref int timeSignature)
        {
            // Check which is the highest and lowest octaves within the notes list
            // Initially set these variables to the values of the first note. They
will be changed later
            int highestOctave = notes[0].GetOctave();
            int lowestOctave = notes[0].GetOctave();
            int currentOctave;

            // Loop through all of the other clips
            for (int i = 1; i < notes.Count; i++)
            {
                // Select the octave of the current note
                currentOctave = notes[i].GetOctave();
                // If the octave of the current note is higher than the highest octave
value, set that as the highest octave
                if (currentOctave > highestOctave)
                {
                    highestOctave = currentOctave;
                }
                // If the current octave is lower than the lowest octave value, set
that as the lowest octave
```

```
                else if (currentOctave < lowestOctave)
                {
                    lowestOctave = currentOctave;
                }
                // This will loop so that we have the lower and upper octave limits
            }

            Octave octave = new Octave();
            // We get the serialised note position data using the function in the
Octave class
            string toDisplay = octave.GetNoteDisplay(ref highestOctave, ref
lowestOctave, ref notes);

            // We clear the console
            Console.Clear();
            // If the clip is unsaved, we just say UNSAVED CLIP at the top, otherwise
we show the clip's name
            if (this.name == null)
            {
                Program.WriteLineCyan("UNSAVED CLIP");
            }
            else
            {
                Program.WriteLineCyan(this.name);
            }
            Console.WriteLine();
            Console.Write("       ");
            // We print the bar indicators at the top of the viewer
            for (int i = 1; i <= timeSignature; i++)
            {
                // We print this spanning divider for the number of the time signature
                Program.WriteGrey("<------" + i + "------->");
            }
            Console.WriteLine();
            // We loop through each character in the serialised note position data
            foreach (char i in toDisplay)
            {
                Console.BackgroundColor = ConsoleColor.Black;
                switch (i)
                {
                    // If the note is natural, print a blue space
                    case '#':
                        Console.BackgroundColor = ConsoleColor.Blue;
                        Console.Write(" ");
                        break;
                    // If the note is sharp, orint a green space
                    case '/':
                        Console.BackgroundColor = ConsoleColor.Green;
                        Console.Write(" ");
                        break;
                    // If the note is flat, print a red space
                    case '*':
                        Console.BackgroundColor = ConsoleColor.Red;
                        Console.Write(" ");
                        break;
                    // Otherwise, print whatever is there
                    default:
                        Console.Write(i);
                        break;
                }
            }
            // Show the instructions below the notes
```

```
            Program.WriteCyan("\nNatural notes are ");
            Program.WriteBlue("blue");
            Program.WriteCyan(", sharp notes are ");
            Program.WriteGreen("green");
            Program.WriteCyan(" and flat notes are ");
            Program.WriteRed("red");
            Program.WriteCyan(".\n");
            Program.WriteMagenta("\nPress any key to return to the arranger ");
            Console.ReadKey();
        }

// This function decides if the note value should be changed
        private bool syncopationRandomiser()
        {
            Random random = new Random();
            // A random number is generated between 1 and the syncopation
randomisation value that the user chose
            int randomNumber = random.Next(1, syncopationRandomisation);
            // If the number is one, then return true
            if (randomNumber == 1)
            {
                return true;
            }
            // If not, return false
            else
            {
                return false;
            }
```

5.3.2 In the Octave Class

```
// This struct contains holders of notes of one octave
        private struct OctaveOfNotes
        {
            public string[] Notes;
        }

// This struct contains a number of an octave and its index in the list of octaves
later
        private struct Pointer
        {
            public int Index;
            public int Octave;
        }

public string GetNoteDisplay(ref int highestOctave, ref int lowestOctave, ref
List<Note> notes)
        {
            string toDisplay = "";

            // List to hold all of the octaves required
            List<OctaveOfNotes> listOfOctaves = new List<OctaveOfNotes>();

            // We keep a value of the number of octaves. We keep is a 1 less than what
the actual number of octaves is so that we can use it as an index pointer for the
arrays later
            int numberOctaves = (highestOctave - lowestOctave) + 1;

            // Generate correct number of octaves
            for (int i = 0; i < numberOctaves; i++)
            {
```

```
                // We instantiate a new octave with 7 rows for each of the 7 notes
                OctaveOfNotes newOctave = new OctaveOfNotes();
                newOctave.Notes = new string[7];
                // We add this octave to the list of octaves
                listOfOctaves.Add(newOctave);
            }

            // Assign the correct octave numbers to each octave created
            int count = 0;
            Pointer[] pointers = new Pointer[numberOctaves];
            // Octaves will be listed in descending order
            // We start labelling octaves with the highest octave first going down to
the lowest one
            for (int i = highestOctave; i >= lowestOctave; i--)
            {
                // Give each note row its corresponding note label at the beginning
                listOfOctaves[count].Notes[0] = "G" + i + "     ";
                listOfOctaves[count].Notes[1] = "F" + i + "     ";
                listOfOctaves[count].Notes[2] = "E" + i + "     ";
                listOfOctaves[count].Notes[3] = "D" + i + "     ";
                listOfOctaves[count].Notes[4] = "C" + i + "     ";
                listOfOctaves[count].Notes[5] = "B" + i + "     ";
                listOfOctaves[count].Notes[6] = "A" + i + "     ";

                // Give a pointer to this octave
                pointers[count].Index = count;
                pointers[count].Octave = i;

                // Increment counter for next octave
                count++;
            }
            // Loop through each of the notes and assign them to the correct octave
            foreach (Note currentNote in notes)
            {
                // Variables used for assigning the notes
                int octaveListIndex = new int();
                int noteInOctave = new int();
                int numberSpacesToUse = new int();
                int octave = currentNote.GetOctave();
                // Find which octave array it needs to be assigned to in the list
                for (int i = 0; i < pointers.Length; i++)
                {
                    if (pointers[i].Octave == octave)
                    {
                        octaveListIndex = pointers[i].Index;
                    }
                }
                // Find which pitch within the note needs to be assigned to
                string pitch = currentNote.GetPitchNumerical();
                // Extract accidental and main pitch
                string accidental = pitch[0].ToString();
                // Change accidental code to #, / and * to avoid confusion with octave
numbers
                switch (accidental)
                {
                    case "0":
                        accidental = "#";
                        break;
                    case "1":
                        accidental = "/";
                        break;
                    case "2":
```

```
                        accidental = "*";
                        break;
                }
                string mainNote = pitch[1].ToString();
                // Choosing which index in one of the octaves needs to be assigned to
                switch (mainNote)
                {
                    // If the note is a C
                    case "1":
                        // Cs are stored in index 4 in each of the octaves
                        noteInOctave = 4;
                        break;
                    // If the note is a D
                    case "2":
                        // Ds are stored in index 3 in each of the octaves
                        noteInOctave = 3;
                        break;
                    // If the note is an E
                    case "3":
                        noteInOctave = 2;
                        break;
                    // If F
                    case "4":
                        noteInOctave = 1;
                        break;
                    // If G
                    case "5":
                        noteInOctave = 0;
                        break;
                    // If A
                    case "6":
                        noteInOctave = 6;
                        break;
                    // If B
                    case "7":
                        noteInOctave = 5;
                        break;
                }
                // Decide how many characters to print based on how long the note
value is
                int noteValue = currentNote.GetNoteValue();
                // Extract note value based on its corresponding code
                switch (noteValue)
                {
                    // If the note is 1 beat
                    case 0:
                        // A note that is 1 beat will take up 16 spaces
                        numberSpacesToUse = 16;
                        break;
                    // If the note is 3/4 beat, it will take up 12 spaces
                    case 1:
                        numberSpacesToUse = 12;
                        break;
                    // If the note is 1/2 beat, it will take up 8 spaces
                    case 2:
                        numberSpacesToUse = 8;
                        break;
                    // 1/4 beat
                    case 3:
                        numberSpacesToUse = 4;
                        break;
                    // 1/8 beat
```

```
                    case 4:
                        numberSpacesToUse = 2;
                        break;
                    // 1/16 beat
                    case 5:
                        numberSpacesToUse = 1;
                        break;
                }
                // Print out accidental code for number of spaces needed
                for (int i = 0; i < numberSpacesToUse; i++)
                {
                    // For each row where our note is contained, its accidental code
(#, / or *) will be assigned
                    // For each row where our note is not contained, spaces will be
added
                    for (int j = 0; j < listOfOctaves.Count; j++)
                    {
                        // Find index where the note is located
                        if (j == octaveListIndex)
                        {
                            for (int k = 0; k < 7; k++)
                            {
                                if (k == noteInOctave)
                                {
                                    // If the current position in the row is correct
for the note that we're adding, we add the accidental code to show the note's position
                                    listOfOctaves[j].Notes[k] += accidental;
                                }
                                else
                                {
                                    // Otherwise, we just add a space
                                    listOfOctaves[j].Notes[k] += " ";
                                }
                            }
                        }
                        else
                        {
                            // For every row for the notes other than the one we're
working on, we add a space
                            for (int k = 0; k < 7; k++)
                            {
                                listOfOctaves[j].Notes[k] += " ";
                            }
                        }
                    }
                }
            }
            // Loop through each of the rows and add to toDisplay
            foreach (OctaveOfNotes currentOctave in listOfOctaves)
            {
                for (int i = 0; i < 7; i++)
                {
                    // Inserts each row separated by line
                    toDisplay += currentOctave.Notes[i] + "\n";
                }
            }
            return toDisplay;
        }
```

Evaluation

1 Introduction

In this section, I will be revisiting parts of the **Analysis** section (page 8), namely the user requirements and the SMART objectives, to analyse whether or not I met these objectives and if not, explaining why.

I will also be discussing how effective the solution is at meeting its needs, as well as analysing feedback given from my client and making suggestions for future development.

2 User Requirements

These are the acceptable user requirements that I listed in the **Analysis**, stating whether they are met or not, as well as why there were met or not met?

Requirement	Was It Met?	How?
The program is database-driven using multiple accounts.	YES	All of the user's accounts are stored in the Accounts table and all of their arrangements and clips are serialised and stored in the other tables.
Work flow needs to be fast and efficient by not including any unnecessary features and by allowing for fast shortcuts.	YES	Getting to each section of the program requires only single key presses, which are not difficult to learn and the keys are shown on the screen. The program is fast to respond.
The program should allow for composition with various keys that can be chosen by the user when entering details for each clip.	NO	It is only possible to choose a key signature for the whole arrangement rather than individual clips.
The program must be able to use clips of notes with fast retrieval from the database. Clips are only downloaded if necessary.	NO	While I did attempt to implement the clip saving part of the system, there were some bugs that prevented those clips from being referenced in arrangement data properly.
The program should be able to create chords as well as monophonic melodies.	NO	Clips can only contain single notes at a time in sequence. Including multiple notes together would require a new object to represent a chord, and each chord could have any number of notes, but each one would need to be checked if the intervals between each of the notes is valid. This would require about 5 times more code for the note generation, and I considered it impractical to put in that much more work for just one extra feature, given the amount of time I had to create this project. Creating chords could be a feature to include in a later version of the software.

3 SMART Objectives

Here, I have listed all of the objectives that I wrote in the **Analysis**, explaining why they've been met or not. The IDs represent the numbers of the objectives in the **Analysis** section:

ID	Objective	Comments
1	To allow the user to log in	This was met because the user can log in.
2	To be able to hash a password	This was met because the SHA-256 algorithm can create a hash in the database.
3	To allow the user to create a new account	This was met because the account wizard takes data from the user and stores it in the database.
4	To display the main menu	This was met because once the user is logged in, the main menu shows.
5	To allow the user to choose an option from the main menu	This was met because pressing a key goes to the appropriate screen.
6	To allow the user to input details for a new arrangement	This was met because the input is taken and saved in a new Arrangement object.
7	To be able to connect to the database	This was met because all of the SQL queries work.
8	To be able to send queries to the database	This was met because the data in the tables changes when queries are executed.
9	To be able to format the select query responses so that they fit the format required within the program	This was met because serialised note data can be extracted from the query responses and be formatted for display to the user.
10	To create a new arrangement record in the arrangement table in the database	This was met because arrangement records are made when a new one is created.
11	To display a blank arrangement page to the screen	This was met because the arranger displays and the user can use it.
12	To let the user choose a previously-saved arrangement	This was met because the arrangement lists are displayed and one can be loaded.
13	To load a previously-saved arrangement	This was not met because arrangements with unsaved clips load fine but ones with saved clips load with no clips. This is the bug that I wasn't able to fix.
14	To display the previously-saved arrangement to the screen	This was met because clips can be displayed and browsed.
15	To allow the user to input a new password	This was met because the password can be changed.
16	To save new user details	This was met because the records in the database can be updated.
17	To display the help screen	This was met because the help screen shows.
18	To be able to log out	This was met because the current user ID is cleared and the welcome screen shows for a new user to log in.
19	To be able to insert a new clip into an arrangement	This was met because a clip object can be created and added to the clips list.
20	To be able to generate a melody sequence for a clip using the details given by the user	This was met because notes can be created correctly inside the clip.
21	To be able to generate a chord sequence using the detail given by the user	This was not met because the note generation algorithm doesn't allow for chords to be created.

22	To ensure that any notes generated in either melody or chord generation are in the correct key and time signature to the project	This was met because notes which need to be sharpened or flattened have it done correctly.
23	To be able to save a clip to the database	This was met because clips' note data can be saved in the Clips table.
24	To be able to load a saved clip into an arrangement	This was met because the user can choose a clip from the list and the correct note data is loaded into a new object.
25	To be able to save an arrangement to the database	This was not met because arrangements with unsaved clips save OK, when there is a saved clip, when a reference is made to it in an arrangement, the reference key corrupts and the clip can't be used in that arrangement. This is part of the bug that I mentioned earlier.
26	To be able to exit an arrangement	This was met because the user is returned to the main menu.
27	To be able to colour-code different areas of the display	This was met because colours work successfully.

4 Effectiveness of the Solution

The program does meet its main objective: generating notes to display to the user so that they can play and save them. However, the way in which the options are presented are not up to modern standards.

The options like range of octaves and syncopation randomisation are not what the average young musician would expect and therefore they may not get the results that they want in the end.

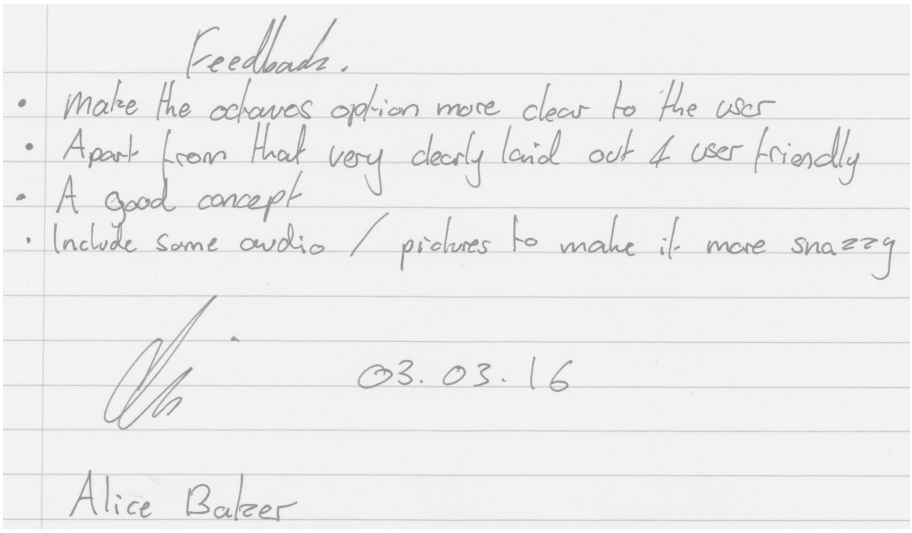
The whole console environment is in my option not very nice to look at and is reminiscent of office applications from the 1980s, not modern composition software. When I showed my program to my peers and my client, they seemed fairly confused at first.

However, the fact that notes can be made shows that this program is a proof of concept.

5 Client Feedback

5.1 Written Comments

I asked my client to use my program and give some general feedback comments:



ID	Comment
1	Make the octaves option more clear to the user.
2	Apart from that very clearly laid out & user friendly.
3	A good concept.
4	Include some audio / pictures to make it more snazzy.

5.2 My Response to the Comments

I originally stated in the **Analysis** section that I was going to use a spiral model for development, i.e. checking the client's opinion on each stage of the development of the solution before moving on. I decided soon afterwards that was not feasible, partly because I didn't have the time but mostly because Alice is a fellow student and I didn't want to distract her from her studies or her social life. The questionnaire and feedback were exchanged on a very informal basis and doing it periodically could have become an annoyance. A spiral model would be more suitable for a professional-level client-developer relationship.

However, I still could have kept Alice more updated with my program's features because it still could have been of interest to her; I should not be surprised about receiving some negative feedback, which I did.

Comment 1 highlights how I don't clearly explain that the range of octaves is the number of octaves allowed above and below octave 4. I could have explained this by including a description next to the field when the user is prompted to enter it. Realistically though, the range of octaves value is not a good idea for music. Certain values for range of octaves gave notes between octaves 1 and 9; ranges of that magnitude are quite extreme for most styles of music and would be difficult to play on a keyboard. Therefore, a better idea would be to have the user choose the lowest octave allowed and the highest octave allowed.

Comments 2, 3 and 4 refer to the user interface. While using the console to clearly lay out the information was feasible, it would be even more user-friendly for a modern Windows application to use forms instead of a console. This would make use of the graphical Windows interface and would allow a mouse to be used. It would also be more expected of by the target users, which are young musicians, who are used to software like Sibelius and MuseScore, which both use Windows form environments. Less serialisation and string manipulation would be required, as would be easier to draw clips and notes to the screen by using panels, labels and objects, rather than pure text. It would also be easier for them to input data using radio buttons and text boxes.

Comment 4 says that there is no audio functionality in the program. Sibelius and MuseScore both allow you to play your arrangements with instrument sounds. This is because all of the notes are stored as MIDI (musical instrument digital interface) data and the programs contain sample libraries for the instruments. When the MIDI files are played, the correct sample for each note is played. I did not incorporate this feature into my program because attaching a MIDI engine and getting all of the notes stored as standard MIDI data would probably take as long as the whole program without that took to write. I would also need to find royalty-free samples for the instrument sounds or, failing that, record my own. The effort required for this isn't practical or necessary for this project. Having said that, it would make the user experience easier and more interesting, because they can preview what each clip sounds like instantly once creating, rather than playing the notes on a piano/keyboard which they may or may not have.

6 Future Development

I have called this version of the program Alpha 1.1. If I was to write Alpha 1.2, I would make, or consider to make, the following changes:

- Fix the saved clip bugs that I mentioned earlier.
- Redesign the program as a Windows form application instead of a console.
- Give the option to create chords as well as monophonic melodies.
- Make the meaning of the different properties clearer to the user.
- Make the properties of the clips more specific (e.g. give a discrete range of octaves).
- Make the process of saving the data less complicated (simplify the serialisation or save by some other means).
- Include a MIDI engine to be able to save the work as MIDI files.
- Include samples so that the clips can be played back using the MIDI data.

Appendix

A1 Transcript of User Interview

Client name: Alice Baker

1) Do you regularly compose music?

Yes

2a) Do you feel like you could be composing more music?

Yes

2b) *(If yes)* What's wrong with your current creative process?

I don't usually have enough time for composing music. Other things take priority.

3) What would be your specific objectives to improve your composition skills?

- Spend more time on composing.
- Listen to a wider variety of music genres and new bands to be more creative and have a wider ranges of influences.

4a) Do you currently use any software to help you with your composition and what is it?

Cubase

4bi) *(If yes)* What features of that software are beneficial to you?

- It is quick and easy to play in (with the keyboard) ideas in case you forget them later, so you can work on them later.
- It is very easy to be creative and make stuff up as you go along until you end up with something awesome.

4bii) What features of that software do you think are beneficial to others?

- It is very user-friendly.

5) What features would you want from a new computer program to help you with your composition skills?

- The ability to real-time record and manipulate sounds.

6) My idea for a program is designed to randomly generate a melody with given details from the user. If the program gave you a choice of details for a certain section of music to generate a melody for that section, what details would you want to add or think are most important to add? *(e.g. key, major/minor, note values)*

- Key and note values (maybe you could give specific notes like C, E & G, and make a melody out of those specific notes).

- You should also have the ability to create chords and harmonies, not just monophonic melodies.

7) How would you want the operating environment of this new program arranged (e.g. how should the menus look)?

- Keep it simple and user-friendly.
- Use colour.

8) Would you prefer the program to be offline and only allow one user on one computer to use it, or use online accounts in a database to allow multiple people to use it?

Database and multiple users

A2 Program Source Code Listings

A2.1 Program Code

A2.1.1 Program Class

```
/// Please note, if brackets () are used, this refers to the name of a function

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using MySql.Data.MySqlClient;
using System.Security.Cryptography;

namespace ProjectGracenoteAlpha1Point1
{
    /// <summary>
    /// This is the default class, which contains the main database connections and
the main menu items.
    /// It instantiates arrangement objects if a user wishes to create a new one or
load one from the database.
    /// It also contains the method for changing the user's password.
    /// </summary>
    class Program
    {
        /// <summary>
        /// Public variables - the SQL connection object, needed for querying the
database, and the ID of the current user logged in
        /// They are public because they need to be used throughout the program
        /// </summary>
        public static MySqlConnection SqlConnection;
        public static int CurrentUserID;

        /// <summary>
        /// Private variables, used for inputs on the login screen and new account
wizard. These are:
        /// Data inputs for when entering usernames, passwords, arrangement
information or key presses
        /// Error messages and flags
        /// A hash object (needed for hash functions)
        /// </summary>
        private static ConsoleKeyInfo newAccountChoice;
```

```
        private static string usernameInput;
        private static string passwordInput;
        private static string newUsername;
        private static string newPassword;
        private static string confirmNewPassword;
        private static ConsoleKeyInfo optionChoice;
        private static string arrangementNameInput;
        private static string tonicNoteInput;
        private static string tonicNote;
        private static string majorMinorInput;
        private static bool major;
        private static string timeSignatureInput;
        private static short timeSignature;
        private static string error;
        private static string footer;
        private static bool errorFlag;
        private static HashAlgorithm hash;

        /// <summary>
        /// The Main function is the first to run
        /// It sets up the database connections and leads to Welcome()
        /// </summary>
        static void Main(string[] args)
        {
            // Instantiate connection and set correct credentials
            SqlConnection = new MySqlConnection("SERVER=127.0.0.1;
DATABASE='ProjectProNote'; UID='root'; PASSWORD=''");
            hash = new SHA256Managed();
            Welcome();
        }

        /// <summary>
        /// This shows the welcome window and gives the user two options:
        /// If they choose to create a new account, they press the F1 key and we go to
ValidateNewAccount()
        /// </summary>
        static void Welcome()
        {
            Console.ForegroundColor = ConsoleColor.White;
            Console.WriteLine("\n\n\n\n\n\n\n");
            WriteLineCyan("                          Welcome to Project ProNote\n");
            WriteLineRed("          WARNING: This is version Alpha 1.1. There will be
bugs!");
            Console.WriteLine();
            Console.WriteLine();
            WriteLineCyan("                      Press F1 to create a new account");
            WriteLineCyan("              or any other key to log in to an existing
account");
            // Check input key for creating new account
            newAccountChoice = Console.ReadKey();
            // If it is F1 then create new account
            if (newAccountChoice.Key == ConsoleKey.F1)
            {
                ValidateNewAccount();
            }
            else
            {
                ValidateLogIn(0);
            }
        }

        /// <summary>
```

```
        /// This displays the new account wizard and prompts the user to enter a new
username and password
        /// They must confirm their password and it must be identical to the first
password
        /// They must not leave any spaces blank
        /// If data is invalid, the screen is reset and an appropriate error message
is shown
        /// If there are no problems, the password is converted into an SHA-256 hash
        /// A NewAccount object is created; the details are set and saved (see
NewAccount.SaveDetails())
        /// Once done, are sent to ValidateLogIn
        /// </summary>
        private static void ValidateNewAccount()
        {
            error = "";
            while (true)
            {
                Console.Clear();
                WriteLineCyan("                                New Account Wizard");
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine("\n" + error + "\n");
                Console.ForegroundColor = ConsoleColor.White;
                WriteLineCyan("Please enter the following details");
                Console.WriteLine();
                // User enters username
                WriteMagenta("Username: ");
                newUsername = Console.ReadLine();
                // Check if username isn't blank
                if (newUsername == "")
                {
                    error = "Please don't leave the username blank";
                }
                else
                {
                    // User enters password
                    WriteMagenta("Password: ");
                    newPassword = Console.ReadLine();
                    if (newPassword == "")
                    {
                        error = "Please don't leave the password blank";
                    }
                    else
                    {
                        // User enters confirm password
                        WriteMagenta("Confirm password: ");
                        confirmNewPassword = Console.ReadLine();
                        if (confirmNewPassword == "")
                        {
                            error = "Please don't leave the confirm password blank";
                        }
                        else
                        {
                            // If both passwords do not match
                            if (confirmNewPassword != newPassword)
                            {
                                error = "Your two passwords do not match";
                            }
                            // If there are no problems
                            else
                            {
                                // The password (newPassword) is converted into an
array of pure unicode bytes
```

```
                                byte[] plainTextBytes =
System.Text.Encoding.UTF8.GetBytes(newPassword);
                                // hash is the object containing hash functions. We
create another array of bytes where we take our original array and perform the hash
function for all the bytes
                                byte[] hashBytes = hash.ComputeHash(plainTextBytes);
                                // We convert the bytes into a string
                                string hashedPassword =
Convert.ToBase64String(hashBytes);

                                // We instantiate the new account object
                                NewAccount newAccount = new NewAccount();
                                newAccount.SetDetails(ref newUsername, ref
hashedPassword);
                                if (!newAccount.SaveDetails())
                                {
                                    error = "Account could not be created. Please try
again later";
                                }
                                else
                                {
                                    ValidateLogIn(1);
                                }
                            }
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Lets the user log into an account using username and password
        /// If it's run from the welcome screen, it just says Login
        /// If it's run from the new account screen, it says Account successfully
created. Log in below
        /// Loops to ensure all data is valid before proceeding
        /// If both fields are blank, clear fields and show error
        /// If fields, hash the password and run LogIn()
        /// LogIn() may run ValidateLogIn again if there the data is invalid, passing
an error code number
        /// If the username doesn't exist, say so
        /// If the password's incorrect, say so
        /// </summary>
        private static void ValidateLogIn(short mode)
        {
            error = "";
            // Loop until details are entered correctly
            while (true)
            {
                Console.Clear();
                Console.WriteLine("\n\n\n\n\n\n\n");
                // If any errors need to be shown before the user enters any details,
they are shown here
                if (mode == 1)
                {
                    WriteLineCyan("                Account successfully created. Log
in below");
                }
                else if (mode == 2)
                {
                    WriteLineCyan("           The username you entered doesn't exist.
Please try again:");
```

```
                }
                else if (mode == 3)
                {
                    WriteLineCyan("                    The password is incorrect. Please
try again:");
                }
                // If there are no problems (mode would be 0)
                else
                {
                    WriteLineCyan("                                          Login");
                }
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine("\n" + error + "\n");
                Console.ForegroundColor = ConsoleColor.White;
                // User enters password
                WriteMagenta("                          Username: ");
                usernameInput = Console.ReadLine();
                // Check if username is not empty
                if (usernameInput == "")
                {
                    error = "                     Please don't leave the username
blank";
                }
                else
                {
                    // User enters password
                    WriteMagenta("                           Password: ");
                    passwordInput = Console.ReadLine();
                    if (passwordInput == "")
                    {
                        error = "                     Please don't leave the password
blank";
                    }
                    else
                    {
                        // Hash password before using it
                        byte[] plainTextBytes =
System.Text.Encoding.UTF8.GetBytes(passwordInput);
                        byte[] hashBytes = hash.ComputeHash(plainTextBytes);
                        string hashedPasswordInput =
Convert.ToBase64String(hashBytes);

                        LogIn(ref usernameInput, ref hashedPasswordInput);
                        break;
                    }
                }
            }
        }

        /// <summary>
        /// This displays the main menu to the user and gives them a choice on what to
proceed with next by getting them to press a key
        /// It also displays the current username's menu at the bottom of the screen
        /// It leads to CreateNewArrangement(), LoadArrangement(), ChangePassword(),
Help() or LogOut() depending on their choice
        /// </summary>
        /// <param name="username">The reference to this parameter gets passed round
most functions of this program to show the username at the bottom of most of the
screens</param>
        public static void MainMenu(ref string username)
        {
            errorFlag = false;
```

```
            footer = "Logged in as " + username + " ";
            // Runs until valid option is input
            do
            {
                Console.Clear();
                WriteLineCyan("                                        Main Menu");
                Console.WriteLine();
                WriteLineCyan("Please choose an option by pressing its number:");
                Console.WriteLine();
                WriteLineMagenta("1. Create a new arrangement");
                WriteLineMagenta("2. Load an arrangement");
                WriteLineMagenta("3. Change password");
                WriteLineMagenta("4. Help");
                WriteLineMagenta("5. Log out");
                Console.WriteLine("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
                // If there is no error, then show the footer as yellow
                if (errorFlag == false)
                {
                    Console.ForegroundColor = ConsoleColor.Yellow;
                }
                // If there is an error, show the footer as red
                else
                {
                    Console.ForegroundColor = ConsoleColor.Red;
                }
                Console.Write(footer);
                Console.ForegroundColor = ConsoleColor.White;
                optionChoice = Console.ReadKey();
                // Perform select case on pressed key
                switch (optionChoice.Key)
                {
                    case ConsoleKey.D1:
                        CreateNewArrangement(ref username);
                        break;
                    case ConsoleKey.D2:
                        LoadArrangement(0, ref username);
                        break;
                    case ConsoleKey.D3:
                        ChangePassword(ref username);
                        break;
                    case ConsoleKey.D4:
                        Help(ref username);
                        break;
                    case ConsoleKey.D5:
                        LogOut();
                        break;
                    default:
                        // If no valid option chosen, error shown
                        errorFlag = true;
                        footer = "Please enter a valid option ";
                        break;
                }
            } while (errorFlag);
        }

        /// <summary>
        /// This loads the create new arrangement wizard. The user enters data for a
new arrangement
        /// This is looped until all data is valid
        /// They cannot make any fields blank, or an error is shown
        /// The tonic note must be a valid musical note
        /// They must enter either major/minor
```

```
        /// They must enter a whole number greater than 0 for time signature
        /// At the end, it creates an Arrangement object, sets its properties and runs
the DisplayArrangement function within it
        /// </summary>
        private static void CreateNewArrangement(ref string username)
        {
            bool errorFlag2 = false;
            error = "";
            do
            {
                Console.Clear();
                WriteLineCyan("                                Create a New Arrangement");
                Console.WriteLine();
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine(error);
                Console.ForegroundColor = ConsoleColor.White;
                Console.WriteLine();
                WriteLineCyan("Please enter:");
                Console.WriteLine();
                WriteMagenta("Arrangement name: ");
                // Input arrangement name
                arrangementNameInput = Console.ReadLine();
                // Validate input for each solution
                // If name is blank, show error
                if (arrangementNameInput == "")
                {
                    error = "Please don't leave the name blank";
                }
                else
                {
                    // Instantiate arrangement object
                    WriteMagenta("Tonic note (use # for sharp and b for flat): ");
                    // Input tonic note
                    tonicNoteInput = Console.ReadLine();
                    // If the tonic note is blank, say so
                    if (tonicNoteInput == "")
                    {
                        error = "Please don't leave the tonic note blank";
                    }
                    // If the note is invalid, say so
                    // In the meantime, this function attempts to convert the tonic
note into its numerical value
                    else if (!IsNoteValid(ref tonicNoteInput, ref tonicNote))
                    {
                        error = "Invalid tonic note. Please try again";
                    }
                    else
                    {
                        // Input major/minor
                        WriteMagenta("Major/minor: ");
                        majorMinorInput = Console.ReadLine();
                        // If the major/minor is blank or it's not valid, show error
                        // In the meantime, if valid, store the value
                        if (!IsMajorMinorValid(majorMinorInput))
                        {
                            error = "Please choose either major or minor";
                        }
                        else
                        {
                            // Input time signature
                            WriteMagenta("Time signature (top number only, bottom
number is 4): ");
```

```
                            timeSignatureInput = Console.ReadLine();
                            // If time signature is invalid, show error
                            // If valid, store value
                            if (!IsTimeSignatureValid(ref timeSignatureInput, ref
timeSignature))
                            {
                                error = "Please enter a whole number greater than 0
for time signature";
                            }
                            else
                            {
                                // If all details entered are correct, create
arrangement
                                errorFlag2 = true;
                                Arrangement arrangement = new Arrangement();
                                arrangement.SetDetails(ref arrangementNameInput, ref
tonicNote, ref major, ref timeSignature);
                                arrangement.CreateArrangement();
                                // Once created, display the arrangement to the user
                                arrangement.DisplayArrangement(ref username, 0);
                            }
                        }
                    }
                }
            } while (!errorFlag2);
        }

        /// <summary>
        /// This returns to the CreateArrangement() function whether or not the time
signature entered by the user is valid
        /// </summary>
        private static bool IsTimeSignatureValid(ref string timeSignatureInput, ref
short timeSignature)
        {
            bool isNumberValid = Int16.TryParse(timeSignatureInput, out
timeSignature);
            // Check if the input is blank, or if the input is not a number. If so,
return false
            if (timeSignatureInput == "" || !isNumberValid)
            {
                return false;
            }
            // If the input is a valid number, return true
            else
            {
                return true;
            }
        }

        /// <summary>
        /// This returns to the CreateArrangement() function whether or not the
major/minor input by the user is valid
        /// We pass in majorMinorInput by val so that it can be converted to lowercase
        /// </summary>
        private static bool IsMajorMinorValid(string majorMinorInput)
        {
            bool flag = true;
            majorMinorInput = majorMinorInput.ToLower();
            if (majorMinorInput == "major")
            {
                major = true;
            }
```

```
            else if (majorMinorInput == "minor")
            {
                major = false;
            }
            else
            {
                flag = false;
            }
            return flag;
        }

        /// <summary>
        /// This function returns to the CreateArrangement() function whether or not
the note entered by the user is valid
        /// At the same time, we set the value of the tonic note variable used for
setting the arrangement's properties
        /// We convert the note to its numerical code
        /// </summary>
        private static bool IsNoteValid(ref string tonicNoteInput, ref string
tonicNote)
        {
            bool flag = false;
            // Take note input and convert to lowercase
            string tonicNoteInputLower = tonicNoteInput.ToLower();
            // The note must be less than 3 characters, otherwise it is not a note
            if (tonicNoteInputLower.Length < 3)
            {
                // Take the first character of the input note. This should be a note
letter from A to G. Convert it to its numerical value from 01-07
                switch (tonicNoteInputLower[0])
                {
                    // As each of these letters are valid, if the chosen note is one
of these, make flag true
                    case 'c':
                        tonicNote = "1";
                        flag = true;
                        break;
                    case 'd':
                        tonicNote = "2";
                        flag = true;
                        break;
                    case 'e':
                        tonicNote = "3";
                        flag = true;
                        break;
                    case 'f':
                        tonicNote = "4";
                        flag = true;
                        break;
                    case 'g':
                        tonicNote = "5";
                        flag = true;
                        break;
                    case 'a':
                        tonicNote = "6";
                        flag = true;
                        break;
                    case 'b':
                        tonicNote = "7";
                        flag = true;
                        break;
                }
```

```
                // If the note is valid, check if it has a sharp or flat, and convert
that accordingly
                // If it has a sharp or flat, the input string will have 2 characters
                if (flag)
                {
                    if (tonicNoteInputLower.Length == 2)
                    {
                        switch (tonicNoteInputLower[1])
                        {
                            // Sharps are represented as 1, flats as 2
                            case '#':
                                tonicNote = "1" + tonicNote;
                                break;
                            case 'b':
                                tonicNote = "2" + tonicNote;
                                break;
                            default:
                                flag = false;
                                break;
                        }
                    }
                    // If there are no accidentals, put a 0 on the front
                    else
                    {
                        tonicNote = "0" + tonicNote;
                    }
                }
            }
            return flag;
        }

        /// <summary>
        /// This displays my email address and a copyright mark
        /// Once the user presses a key, we are returned to MainMenu()
        /// </summary>
        private static void Help(ref string username)
        {
            Console.Clear();
            WriteLineCyan("                                        Help\n\nIf you have
any questions, feel free to send me an email:\nross.hugh.duncan@gmail.com\n\n© Ross
Duncan 2015-2016\n");
            WriteMagenta("Press any key to return to the main
menu\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
            WriteYellow("Logged in as " + username + " ");
            Console.ReadKey();
            MainMenu(ref username);
        }

        /// <summary>
        /// This clears the current user ID variable and loads Welcome() to let
another user log in
        /// </summary>
        private static void LogOut()
        {
            // We use -1 to mark the user as being logged out
            CurrentUserID = -1;
            Console.Clear();
            Welcome();
        }

        /// These next few functions are used for printing text in various colours
        /// They are public because they are used in other classes
```

```
// Function for writing text as green, then changing back to white
public static void WriteGreen(string toDisplay)
{
    Console.ForegroundColor = ConsoleColor.Green;
    Console.Write(toDisplay);
    Console.ForegroundColor = ConsoleColor.White;
}

// Function for cyan text
public static void WriteLineCyan(string toDisplay)
{
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine(toDisplay);
    Console.ForegroundColor = ConsoleColor.White;
}

public static void WriteYellow(string toDisplay)
{
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.Write(toDisplay);
    Console.ForegroundColor = ConsoleColor.White;
}

public static void WriteLineRed(string toDisplay)
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(toDisplay);
    Console.ForegroundColor = ConsoleColor.White;
}

public static void WriteGrey(string toDisplay)
{
    Console.ForegroundColor = ConsoleColor.DarkGray;
    Console.Write(toDisplay);
    Console.ForegroundColor = ConsoleColor.White;
}

public static void WriteLineMagenta(string toDisplay)
{
    Console.ForegroundColor = ConsoleColor.Magenta;
    Console.WriteLine(toDisplay);
    Console.ForegroundColor = ConsoleColor.White;
}

public static void WriteMagenta(string toDisplay)
{
    Console.ForegroundColor = ConsoleColor.Magenta;
    Console.Write(toDisplay);
    Console.ForegroundColor = ConsoleColor.White;
}

public static void WriteRed(string toDisplay)
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.Write(toDisplay);
    Console.ForegroundColor = ConsoleColor.White;
}

public static void WriteBlue(string toDisplay)
{
    Console.ForegroundColor = ConsoleColor.Blue;
```

```
            Console.Write(toDisplay);
            Console.ForegroundColor = ConsoleColor.White;
        }

        public static void WriteCyan(string toDisplay)
        {
            Console.ForegroundColor = ConsoleColor.Cyan;
            Console.Write(toDisplay);
            Console.ForegroundColor = ConsoleColor.White;
        }

        public static void WriteLineGreen(string toDisplay)
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine(toDisplay);
            Console.ForegroundColor = ConsoleColor.White;
        }

        /// <summary>
        /// This selects the account ID, username and password from the Accounts table
in the database
        /// If it cannot find the username that the user enters, or if the password is
incorrect, it runs ValidateLogIn() again with an error
        /// If the details are correct, we are passed to MainMenu()
        /// </summary>
        private static void LogIn(ref string username, ref string password)
        {
            // Retrieve details
            // First, check that username exists. If it doesn't, an exception will
thrown, which we need to catch
            MySqlCommand findDetails = SqlConnection.CreateCommand();
            findDetails.CommandText = "select PK_AccountID, Username, Password from
Accounts where Username = '" + username + "'";
            MySqlDataReader findDetailsReader;
            string retrievedPassword = "";
            OpenConnection();
            findDetailsReader = findDetails.ExecuteReader();
            // Extract password into string and current user ID into integer
            // (whilst the reader is reading from the database)
            if (findDetailsReader.Read())
            {
                retrievedPassword = findDetailsReader.GetString(2);
                // If the username is retrievable but the password is incorrect
                if (retrievedPassword == password)
                {
                    // Set current user ID
                    CurrentUserID = findDetailsReader.GetInt32(0);
                    findDetailsReader.Close();
                    SqlConnection.Close();
                    // Show menu the main menu
                    MainMenu(ref username);
                }
                // No?
                else
                {
                    findDetailsReader.Close();
                    SqlConnection.Close();
                    ValidateLogIn(3);
                }

            }
            // If the username cannot be read (it is invalid), show error
```

```
            else
            {
                findDetailsReader.Close();
                ValidateLogIn(2);
            }
            MainMenu(ref username);
        }

        /// <summary>
        /// This is used throughout the program to open the SQL connection
        /// If it is closed, open it
        /// If it has been left open, close the existing connection and open a new one
        /// </summary>
        public static void OpenConnection()
        {
            try
            {
                SqlConnection.Open();
            }
            catch
            {
                SqlConnection.Close();
                SqlConnection.Open();
            }
        }

        /// <summary>
        /// These two structs are used for storing a list of arrangement names when we
are choosing from a list to load from
        /// They contain pointers to the arrangement IDs in the database
        /// </summary>

        public struct ArrangementRow
        {
            public int ID;
            public string Name;
        }

        public struct Pointer
        {
            public int ListItem;
            public int ID;
        }

        /// <summary>
        /// This retrieves the names and indices of all of the arrangements saved
under the user's account and displays them to the user
        /// The user then chooses
        /// If they make an invalid choice, show an error
        /// If they make a valid choice, create a new Arrangement object and load the
data from the selected arrangement record in the database, and display it
        /// </summary>
        private static void LoadArrangement(int mode, ref string username)
        {
            MySqlCommand command = SqlConnection.CreateCommand();
            command.CommandText = "select PK_ArrangementID, ArrangementName from
Arrangements where FK_AccountID = " + CurrentUserID;
            MySqlDataReader reader;
            List<ArrangementRow> list = new List<ArrangementRow>();
            ArrangementRow newRow;
            // Read list of arrangements from table
            OpenConnection();
```

```
            reader = command.ExecuteReader();
            // Loop until all arrangement names have been retrieved and store each row
in the list
            while (reader.Read())
            {
                newRow.ID = reader.GetInt32(0);
                newRow.Name = reader.GetString(1);
                list.Add(newRow);
            }
            SqlConnection.Close();

            // The different arrangements will be displayed as a numbered list.
However, each number will need to point to an ID within the arrangement table
            // Therefore, I need to create a list of pointers
            List<Pointer> pointers = new List<Pointer>();
            Pointer newPointer;
            Console.Clear();
            WriteLineCyan("                                Load Arrangement\n");
            // First check if the user has no saved arrangements
            // If that's true, show an error
            if (list.Count == 0)
            {
                WriteLineRed("\nThere are no arrangements saved to this
account.\nPress any key to return to the main menu.");
                WriteYellow("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\nLogged in as " +
username + " ");
                Console.ReadKey();
                MainMenu(ref username);
            }
            // If that's not the case, then run the rest of the code as usual
            else
            {
                // Print each of the arrangement names in the list
                for (int i = 0; i < list.Count(); i++)
                {
                    // Must be +1 to prevent 0-indexing
                    newPointer.ListItem = i + 1;
                    newPointer.ID = list[i].ID;
                    pointers.Add(newPointer);
                    WriteLineGreen((i + 1) + ". " + list[i].Name);
                }
                switch (mode)
                {
                    // Normal mode, where there is no errors
                    case 0:
                        // Prompt user to enter choice
                        WriteCyan("\nEnter number of arrangement to load: ");
                        break;
                    case 1:
                        WriteCyan("\nInvalid choice. Please try again: ");
                        break;
                }
                int choice;
                try
                {
                    choice = Convert.ToInt32(Console.ReadLine());
                    // Translate choice number into its corresponding arrangement ID
                    int id = new int();
                    // Flag is needed to check if the choice is valid
                    bool flag = false;
                    for (int i = 0; i < pointers.Count; i++)
                    {
```

```
                        if (pointers[i].ListItem == choice)
                        {
                            id = pointers[i].ID;
                            flag = true;
                            break;
                        }
                    }
                    // If the choice is invalid, prompt to choose the arrangement
again
                    if (!flag)
                    {
                        LoadArrangement(1, ref username);
                    }
                    // If the choice is valid, create a new arrangement object and
load the data
                    else
                    {
                        Arrangement loadingArrangement = new Arrangement();
                        loadingArrangement.LoadArrangement(ref id, ref username);
                    }
                }
                catch
                {
                    // If the user makes an invalid input, run the function again but
with showing an error
                    LoadArrangement(1, ref username);
                }
            }
        }

        /// <summary>
        /// This opens the change password wizard
        /// This loops until the user has entered the right data
        /// They must enter their old password and new one twice. If they don't, the
form is reset and an error is shown
        /// </summary>
        private static void ChangePassword(ref string username)
        {
            bool flag = false;
            string error = "";
            string currentPassword;
            do
            {
                Console.Clear();
                WriteLineCyan("                                    Change
Password\n\nPlease enter:\n");
                WriteLineRed(error + "\n");
                WriteMagenta("Your current password: ");
                // Enter current password
                currentPassword = Console.ReadLine();
                // If the password is empty
                if (currentPassword == "")
                {
                    error = "Please don't leave the password blank";
                }
                else
                {
                    // If the password is not correct
                    if (!ValidateCurrentPassword(ref currentPassword))
                    {
                        error = "The password is incorrect. Please try again";
                    }
```

```
                    else
                    {
                        // Enter new password
                        WriteMagenta("Your new password: ");
                        newPassword = Console.ReadLine();
                        // Check if empty
                        if (newPassword == "")
                        {
                            error = "Please don't leave the password blank";
                        }
                        else
                        {
                            // Confirm new password
                            WriteMagenta("Confirm your new password: ");
                            confirmNewPassword = Console.ReadLine();
                            if (confirmNewPassword == "")
                            {
                                error = "Please don't leave the confirm password
blank";
                            }
                            else if (confirmNewPassword != newPassword)
                            {
                                error = "Your two passwords don't match. Please try
again";
                            }
                            else
                            {
                                flag = true;
                            }
                        }
                    }
                }
            } while (flag == false);

            // If no errors, first hash password then send to database
            byte[] plainTextBytes = System.Text.Encoding.UTF8.GetBytes(newPassword);
            byte[] hashBytes = hash.ComputeHash(plainTextBytes);
            string hashedPassword = Convert.ToBase64String(hashBytes);

            MySqlCommand sendCommand = SqlConnection.CreateCommand();
            sendCommand.CommandText = "update Accounts set Password = \"" +
hashedPassword + "\" where PK_AccountID = " + CurrentUserID;
            OpenConnection();
            sendCommand.ExecuteNonQuery();
            SqlConnection.Close();
            System.Diagnostics.Stopwatch stopwatch = new
System.Diagnostics.Stopwatch();
            WriteLineGreen("\nYour password has been changed");
            stopwatch.Start();
            // Wait for one second before displaying success message
            while (stopwatch.ElapsedMilliseconds < 1000)
            {
                // Wait
            }
            MainMenu(ref username);
        }

        /// <summary>
        /// This hashes the password input by the user on the change password wizard
and checks it against the user's current password in the database with a select
statement
```

```
        /// If it is identical, return true back to ChangePassword(), otherwise return
false
        /// </summary>
        private static bool ValidateCurrentPassword(ref string currentPassword)
        {
            bool flag = false;

            // Hash password
            byte[] plainTextBytes =
System.Text.Encoding.UTF8.GetBytes(currentPassword);
            byte[] hashBytes = hash.ComputeHash(plainTextBytes);
            string hashedCurrentPassword = Convert.ToBase64String(hashBytes);

            MySqlCommand command = SqlConnection.CreateCommand();
            command.CommandText = "select Password from Accounts where PK_AccountID =
" + CurrentUserID;
            string realPassword = "";
            MySqlDataReader reader;
            OpenConnection();
            reader = command.ExecuteReader();
            while (reader.Read())
            {
                realPassword = reader.GetString(0);
            }
            SqlConnection.Close();
            // Check if password is valid
            if (realPassword == hashedCurrentPassword)
            {
                flag = true;
            }

            return flag;
        }
    }
}
```

A2.2.2 NewAccount Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using MySql.Data.MySqlClient;

namespace ProjectGracenoteAlpha1Point1
{
    /// <summary>
    /// This class contains the method to create a new account
    /// </summary>
    class NewAccount
    {
        // Properties
        private string username;
        private string password;

        /// <summary>
        /// Access modifier
        /// Passes username and password into object by reference
        /// </summary>
        public void SetDetails(ref string Username, ref string Password)
        {
```

```
            username = Username;
            password = Password;
        }

        /// <summary>
        /// Runs insert SQL statement using username and password to create new record
in Accounts table
        /// Returns boolean flag back to Program to show if successful or not
        /// </summary>
        public bool SaveDetails()
        {
            bool flag = true;
            // Using connection in Program, send to database
            MySqlCommand sender = new MySqlCommand("insert into Accounts (Username,
Password) values (\"" + username + "\", \"" + password + "\")",
Program.SqlConnection);
            // If there are any problems with the connection, show error (error is
shown in Program)
            try
            {
                Program.SqlConnection.Open();
                sender.ExecuteNonQuery();
                Program.SqlConnection.Close();
            }
            catch
            {
                flag = false;
            }
            return flag;
        }
    }
}
```

A2.2.3 Arrangement Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using MySql.Data.MySqlClient;

namespace ProjectGracenoteAlpha1Point1
{
    /// <summary>
    /// This class used as a template for the arrangement object.
    /// Its behaviour includes the methods for creating, loading and displaying an
arrangement, as well as creating, loading and deleting clips (modifying the clip list)
    /// </summary>
    class Arrangement
    {
        /// <summary>
        /// These are all property variables for an arrangement, used for when the
user in inputting data or data is being retrieved from the database
        /// </summary>
        private List<Clip> clips;
        private string name;
        // tonicNoteNumerical and sharps must be public and static as they need to be
accessed from the Clip class
        public static string tonicNoteNumerical;
        public bool sharps;
        public bool major;
```

```
        private static int timeSignature;
        // Marks if the arrangement is saved in the database or not
        private bool saved;
        // Stores the index of the clip currently selected
        private int selectedClipIndex = 0;
        // Used for storing the PK_ArrangementID of an arrangement once it's assigned
to the database
        private int ID;
        private string rangeOfOctavesInput;
        private string rangeOfNotesInput;
        private string syncopationRandomisationInput;
        private int rangeOfOctaves;
        private int rangeOfNotes;
        private int syncopationRandomisation;

        // Constructor
        public Arrangement()
        {
            clips = new List<Clip>();
            saved = false;
            ID = 0;
        }

        /// <summary>
        /// Access modifier
        /// Allows functions outside of the Arrangement class to access the properties
        /// </summary>
        public void SetDetails(ref string Name, ref string TonicNote, ref bool Major,
ref short TimeSignature)
        {
            name = Name;
            tonicNoteNumerical = TonicNote;
            major = Major;
            timeSignature = TimeSignature;
        }


        /// <summary>
        /// This section displays the arrangement to the user, namely the title, the
clip blocks and the options
        /// It takes input from the user of what actiont to perform next
        /// </summary>
        /// <param name="username">This is a reference to the variable storing the
name of the user currently logged in, which is shown on the screen</param>
        /// <param name="error">This is a single-digit code representing what message
the arranger should display depending on whether or not that is an error</param>
        public void DisplayArrangement(ref string username, short error)
        {
            bool flag = true;
            bool flag2 = true;
            string footer = "";
            // If there is an error, display error message
            if (error == 0)
            {
                footer = "Logged in as " + username + " ";
            }
            // If there are no clips available to display, say so
            else if (error == 1)
            {
                footer = "There are no clips to display ";
                flag2 = false;
            }
```

```
            // if the user tries to enter a clip but it's already saved
            else if (error == 2)
            {
                footer = "This clip is already saved ";
                flag2 = false;
            }
            else if (error == 3)
            {
                footer = "Insert a clip to save ";
                flag2 = false;
            }
            // Run until the user has chosen a valid option to continue
            do
            {
                Console.Clear();
                Program.WriteLineCyan(name + "\n");
                // Clip display section
                // If no clips are within arrangement yet, say so
                if (clips.Count == 0)
                {
                    Program.WriteLineRed("                                    < No clips
>");
                }
                // If there are clips, show them appropriately
                else
                {
                    Clip currentClip;
                    // Display bar numbers for each clip
                    for (int i = 0; i < clips.Count; i++)
                    {
                        // Display bar number halfway along clip name
                        // If there is no name for the clip, use 4 characters
                        int length;
                        if (clips[i].GetName() == null)
                        {
                            length = 4;
                        }
                        else
                        {
                            length = clips[i].GetName().Length;
                        }
                        int halfLength = length / 2;
                        // If clip name is an odd number of characters, miss out one
space
                        if (length % 2 == 0)
                        {
                            PrintSpaces(halfLength);
                        }
                        else
                        {
                            PrintSpaces(halfLength - 1);
                        }
                        Program.WriteGrey((i + 1).ToString());
                        PrintSpaces(halfLength);
                    }
                    Console.Write("\n");
                    // Now, display the actual clips
                    // The clip selected will be displayed as a different colour to
the other clips
                    for (int i = 0; i < clips.Count; i++)
                    {
                        currentClip = clips[i];
```

```
                        // If the clip is saved, display as green
                        if (currentClip.IsSaved())
                        {
                            // If the clip is the one currently selected, display as a
lighter green than the unselected clips
                            if (i == selectedClipIndex)
                            {
                                Console.BackgroundColor = ConsoleColor.Green;
                            }
                            else
                            {
                                Console.BackgroundColor = ConsoleColor.DarkGreen;
                            }
                            Console.Write(currentClip.GetName() + " ");
                        }
                        // If not saved, show as red
                        else
                        {
                            // If the clip is the one currently selected, display as a
lighter green than the unselected clips
                            if (i == selectedClipIndex)
                            {
                                Console.BackgroundColor = ConsoleColor.Red;
                            }
                            else
                            {
                                Console.BackgroundColor = ConsoleColor.DarkRed;
                            }
                            Console.Write(" *   ");
                        }
                    }
                    Console.BackgroundColor = ConsoleColor.Black;
                }
                Console.WriteLine();
                Program.WriteLineCyan("\nEnter number for action:\n");
                Program.WriteLineMagenta("1 - Insert clip     2 - Load clip     3 -
Save clip     4 - Delete clip\n5 - View clip     6 - Save arrangement     7 -
Exit\n");
                Program.WriteLineCyan("Use arrow keys to select
clip\n\n\n\n\n\n\n\n\n\n\n\n\n");
                if (flag2)
                {
                    Program.WriteYellow(footer);
                }
                else
                {
                    Program.WriteRed(footer);
                }
                flag = true;
                // Take key input from user to decide the next action
                ConsoleKeyInfo nextAction = Console.ReadKey();
                switch (nextAction.Key)
                {
                    case ConsoleKey.D1:
                        InsertClip(ref username);
                        break;
                    case ConsoleKey.D2:
                        LoadClip(ref username, false);
                        break;
                    case ConsoleKey.D3:
                        SaveClip(ref username);
                        break;
```

```
                    case ConsoleKey.D4:
                        DeleteClip(ref username);
                        break;
                    case ConsoleKey.D5:
                        ViewClip(ref username, ref timeSignature);
                        break;
                    case ConsoleKey.D6:
                        SaveArrangement(ref username);
                        break;
                    case ConsoleKey.D7:
                        Exit(ref username);
                        break;
                    case ConsoleKey.LeftArrow:
                        if (selectedClipIndex > 0)
                        {
                            selectedClipIndex--;
                        }
                        flag = false;
                        break;
                    case ConsoleKey.RightArrow:
                        if (selectedClipIndex < clips.Count - 1)
                        {
                            selectedClipIndex++;
                        }
                        flag = false;
                        break;
                    // If no valid option is input, display error
                    default:
                        flag2 = false;
                        footer = "Please choose a valid option";
                        break;
                }
            } while (!flag);
        }

        /// <summary>
        /// This is used for printing the correct number of spaces in between the bar
numbers
        /// </summary>
        private void PrintSpaces(int halfLength)
        {
            for (int j = 1; j <= halfLength; j++)
            {
                Console.Write(" ");
            }
        }

        /// <summary>
        /// This section is used for inserting a clip. It asks the user for the
details, validates them and gets them to enter them again if they are incorrect.
        /// If all is good, a clip object is created and the CreateClip method within
the object is run, populating it with notes.
        /// Then, the user is returned to the arranger with the new clip displayed.
        /// </summary>
        private void InsertClip(ref string username)
        {
            bool flag = false;
            string error = "";
            do
            {
                Console.Clear();
                Program.WriteLineCyan("                                    Insert Clip");
```

```
                Console.WriteLine();
                Program.WriteLineRed(error);
                Console.WriteLine();
                Program.WriteLineCyan("Enter details for a new clip below:");
                Console.WriteLine();
                // Input range of octaves
                Program.WriteMagenta("Range of octaves: ");
                rangeOfOctavesInput = Console.ReadLine();
                // Validate range of octaves
                if (rangeOfOctavesInput == "")
                {
                    error = "Please don't leave the range of octaves blank";
                }
                else if (!Int32.TryParse(rangeOfOctavesInput, out rangeOfOctaves))
                {
                    error = "Please enter a valid range of octaves (a whole number)";
                }
                else
                {
                    // Input range of notes
                    Program.WriteMagenta("Range of notes (from 1 to 8): ");
                    rangeOfNotesInput = Console.ReadLine();
                    // Validate range of notes
                    if (rangeOfNotesInput == "")
                    {
                        error = "Please don't leave the range of notes blank";
                    }
                    // Check if range of notes is a number, then check if it is
between 1 and 8
                    else if (!Int32.TryParse(rangeOfNotesInput, out rangeOfNotes) ||
rangeOfNotes < 1 || rangeOfNotes > 8)
                    {
                        error = "Please enter a valid range of notes (a whole number
between 1 and 8)";
                    }
                    else
                    {
                        // Input syncopation randomisation
                        Program.WriteMagenta("Syncopation randomisation (from 1 to
10)\n(roughly how much do you want the note values to change?): ");
                        syncopationRandomisationInput = Console.ReadLine();
                        // Validate syncopation randomisation
                        if (syncopationRandomisationInput == "")
                        {
                            error = "Please don't leave the syncopation randomisation
blank";
                        }
                        else if (!Int32.TryParse(syncopationRandomisationInput, out
syncopationRandomisation) || syncopationRandomisation < 1 || syncopationRandomisation
> 10)
                        {
                            error = "Please enter a valid syncopation randomisation
value (a whole number between 1 and 10)";
                        }
                        // If all input is valid, create new clip and return to the
arranger
                        else
                        {
                            flag = true;
                            Clip newClip = new Clip();
                            newClip.SetDetails(ref rangeOfOctaves, ref rangeOfNotes,
ref syncopationRandomisation);
```

```
                            newClip.CreateClip(ref sharps, ref major);
                            clips.Add(newClip);
                        }
                    }
                }
            } while (!flag);
            DisplayArrangement(ref username, 0);
        }

        // This is an access modifier for the tonic note
        public string GetTonicNoteNumerical()
        {
            return tonicNoteNumerical;
        }

        /// <summary>
        /// This section will look up and return the number of accidentals for each
key signature.
        /// I have included conditions for both the major key and their equivalent
minor keys.
        /// This is based on the circle of fifths as shown in the Design Section 3.3.1
        /// </summary>
        /// <param name="sharps">This refers to the flag within the arranger to
determine whether or not the key signature uses sharps or flats, so that it knows what
colour to display the notes as.</param>
        /// <param name="major">The flag used for determining if the key is major or
minor is needed for looking up the correct tonic note</param>
        public static int GetNumberAccidentals(ref bool sharps, ref bool major)
        {
            int numberAccidentals = new int();

            // G major / E minor have 1 sharp
            if ((tonicNoteNumerical == "05" && major) || (tonicNoteNumerical == "03"
&& !major))
            {
                numberAccidentals = 1;
                sharps = true;
            }
            // D major / B minor have 2 sharps
            else if ((tonicNoteNumerical == "02" && major) || (tonicNoteNumerical ==
"07" && !major))
            {
                numberAccidentals = 2;
                sharps = true;
            }
            // A major / F# minor have 3 sharps
            else if ((tonicNoteNumerical == "06" && major) || (tonicNoteNumerical ==
"14" && !major))
            {
                numberAccidentals = 3;
                sharps = true;
            }
            // E major / C# minor have 4 sharps
            else if ((tonicNoteNumerical == "03" && major) || (tonicNoteNumerical ==
"11" & !major))
            {
                numberAccidentals = 4;
                sharps = true;
            }
            // B major / G# minor have 5 sharps
            else if ((tonicNoteNumerical == "07" && major) || (tonicNoteNumerical ==
"15" & !major))
```

```
            {
                numberAccidentals = 5;
                sharps = true;
            }
            // Cb major has 7 flats
            else if (tonicNoteNumerical == "21" & major)
            {
                numberAccidentals = 7;
                sharps = false;
            }
            // Gb and F# major, and Eb and D# minor have 6 sharps
            else if ((tonicNoteNumerical == "25" && major) || (tonicNoteNumerical ==
"14" && major) || (tonicNoteNumerical == "23" && !major) || (tonicNoteNumerical ==
"12" && !major))
            {
                numberAccidentals = 6;
                sharps = true;
            }
            // C# major has 7 sharps
            else if (tonicNoteNumerical == "11" && major)
            {
                numberAccidentals = 7;
                sharps = true;
            }
            // Db major / Bb minor have 5 flats
            else if ((tonicNoteNumerical == "22" && major) || (tonicNoteNumerical ==
"27" && !major))
            {
                numberAccidentals = 5;
                sharps = false;
            }
            // Ab major / F minor have 4 flats
            else if ((tonicNoteNumerical == "26" && major) || (tonicNoteNumerical ==
"04" && !major))
            {
                numberAccidentals = 4;
                sharps = false;
            }
            // Eb major / C minor have 3 flats
            else if ((tonicNoteNumerical == "23" && major) || (tonicNoteNumerical ==
"01" && !major))
            {
                numberAccidentals = 3;
                sharps = false;
            }
            // Bb major / G minor have 2 flats
            else if ((tonicNoteNumerical == "27" && major) || (tonicNoteNumerical ==
"05" && !major))
            {
                numberAccidentals = 2;
                sharps = false;
            }
            // F major / D minor have 1 flat
            else if ((tonicNoteNumerical == "04" && major) || (tonicNoteNumerical ==
"02" && !major))
            {
                numberAccidentals = 1;
                sharps = false;
            }
            // C major / A minor have no sharps/flats
            // If the user chooses a note that's not supported, don't sharpen or
flatten any notes for safety
```

```
            else
            {
                numberAccidentals = 0;
            }

            return numberAccidentals;
        }

        // Access modifier for the time signature
        public static int GetTimeSignature()
        {
            return timeSignature;
        }

        // This is run if the user wants to exit the arrangement. They are returned to
the main menu
        private void Exit(ref string username)
        {
            Program.MainMenu(ref username);
        }

        /// <summary>
        /// This section displays the currently-selected clip to the user.
        /// The index in the list of the clip currently selected is determined by the
selectedClipIndex variable.
        /// If the list of clips is not empty, it will run the method within the
selected clip which displays its notes.
        /// </summary>
        private void ViewClip(ref string username, ref int timeSignature)
        {
            // Check if clip is empty
            // If it is, show error
            if (clips.Count == 0)
            {
                DisplayArrangement(ref username, 1);
            }
            // If not, display clips normally
            else
            {
                clips[selectedClipIndex].DisplayClip(ref timeSignature);
                DisplayArrangement(ref username, 0);
            }
        }

        /// <summary>
        /// If the user chooses to load an arrangement, this method retrieves said
arrangement's data using SQL.
        /// It splits the serialised clip data into clips and decides whether it needs
to create an unsaved clip object or a saved clip object.
        /// </summary>
        /// <param name="IDtoSelect">This is the PK_ArrangementID value that the user
has chosen to load</param>
        public void LoadArrangement(ref int IDtoSelect, ref string username)
        {
            ID = IDtoSelect;
            // First, retrieve arrangement data from database
            MySqlCommand select = Program.SqlConnection.CreateCommand();
            select.CommandText = "select ArrangementName, ArrangementData,
ClipContentsData from Arrangements where PK_ArrangementID = " + IDtoSelect;
            // Store arrangement data and clip contents data in strings
            string arrangementName = "";
            string arrangementData = "";
```

```
            string clipContentsData = "";
            // Execute statement
            Program.OpenConnection();
            MySqlDataReader reader = select.ExecuteReader();
            while (reader.Read())
            {
                arrangementName = reader.GetString(0);
                arrangementData = reader.GetString(1);
                try
                {
                    clipContentsData = reader.GetString(2);
                }
                catch
                {
                    // Do nothing if field is null
                }
            }
            Program.SqlConnection.Close();

            // Name the arrangement after the name in the record
            name = arrangementName;

            // This section makes reference to the encoding. For details on how the
encoding works, see Section 5.2.2 of the Design

            // Decode the data, starting with the arrangement data
            // First, the tonic note
            // The first character in the data should be a 0 (at index 0)
            tonicNoteNumerical += arrangementData[1];
            tonicNoteNumerical += arrangementData[2];
            // This should now give the first two digits of the key tonic note
            // The next character should be a 1 (index 3), followed by the key (index
4)
            // If the character for the key is a C, the key is major. If it is a D,
the key is minor
            if (arrangementData[4] == 'C')
            {
                major = true;
            }
            else
            {
                major = false;
            }
            // The next character should be a 2 (index 5) followed by the time
signature (index 6)
            // As extraction from array gives a char ASCII value, we subtract it to
give the correct time signature
            timeSignature = Convert.ToInt32(arrangementData[6]) - 48;

            // After the basic details, we now need to extract the clips. We must
start at index 7
            int index = 7;
            string newKey = "";
            try
            {
                do
                {
                    // If there is an A in the string, this is the start of a clip
that's been saved, followed by its compound primary key in the ClipUses table
                    if (arrangementData[index] == 'A')
                    {
                        // Try catch block in case the end of the string is reached
```

```
                        try
                        {
                            do
                            {

                                index++;
                                // If the loop has found the start of a new clip, then
exit and start on the next clip
                                if (arrangementData[index] == 'A' ||
arrangementData[index] == 'B')
                                {
                                    // Add this new clip using the key, to look up in
the database
                                    Clip newClip = new Clip();
                                    newClip.LoadClipWithKey(ref newKey);
                                    clips.Add(newClip);
                                    break;
                                }
                                // Otherwise, add the key number to the new key
variable
                                else
                                {
                                    newKey += arrangementData[index];
                                }
                                // Make sure there is no overflow
                            } while (index < arrangementData.Length);
                        }
                        catch
                        {
                            Clip newClip = new Clip();
                            newClip.LoadClipWithKey(ref newKey);
                            clips.Add(newClip);
                        }
                    }
                    // If it is a B, then this is the start of a clip that's not been
saved, followed by the reference code to the clip data in the clip contents data
variable
                    else
                    {
                        if (arrangementData[index] != 'E')
                        {
                            index++;
                        }
                        else
                        {
                            // The next character after the B should be an E
                            string id = "";
                            // After the E, there should be the ID for the unsaved
clip
                            do
                            {
                                id += arrangementData[index];
                                // If F has been reached, then it is the end of that
clip
                                if (arrangementData[index] == 'F')
                                {
                                    // Search for the clip details within the clip
contents data variable
                                    SearchUnsavedClip(id, clipContentsData);
                                    break;
                                }
                                index++;
```

```
                        } while (true);
                        index++;
                    }
                }
            } while (index < arrangementData.Length - 1);
        }
        catch
        {
            // Do nothing in case there is no data
        }
        DisplayArrangement(ref username, 0);
    }

    /// <summary>
    /// This is used for searching for a particular unsaved clip given a BE-F key.
    /// Once the key required is found, a clip is created using the note data
adjacent to the key.
    /// </summary>
    /// <param name="id">This is the key to look for</param>
    /// <param name="clipContentsData">This is the note data to search
through</param>
    private void SearchUnsavedClip(string id, string clipContentsData)
    {
        int i;
        string key = "";
        bool findingKey = false;
        // Loop through the data to find the right clip
        for (i = 0; i < clipContentsData.Length; i++)
        {
            if (findingKey)
            {
                key += clipContentsData[i];
                // If we have reached the end of one key
                if (clipContentsData[i] == 'F')
                {
                    findingKey = false;
                    if (key == id)
                    {
                        // If we have found the right key, we exit the loop
                        break;
                    }
                    // Otherwise, we have to start again
                    else
                    {
                        key = "";
                    }
                }
            }
            else
            {
                if (clipContentsData[i] == 'E')
                {
                    key += clipContentsData[i];
                    findingKey = true;
                }
            }
        }

        // Once the key has been found, extract the data
        i++;
        string data = "";
        do
```

```
            {
                try
                {
                    // Keep adding data until another E is reached or if the end of
the array is reached
                    if (clipContentsData[i] != 'E')
                    {
                        data += clipContentsData[i];
                        i++;
                    }
                    // Otherwise, send the new clip to be added
                    else
                    {
                        Clip newClip = new Clip();
                        newClip.LoadClipWithData(ref data, true);
                        clips.Add(newClip);
                        break;
                    }
                }
                // If the end of the array has been reached then there will be an
index out of bounds error, hence the try catch block
                catch
                {
                    Clip newClip = new Clip();
                    newClip.LoadClipWithData(ref data, true);
                    clips.Add(newClip);
                    break;
                }
            } while (true);
        }

        /// <summary>
        /// This is run when the user wishes to save a clip into the Clips table.
        /// It runs the save method within the selected clip object.
        /// </summary>
        private void SaveClip(ref string username)
        {
            // Check if clip is already saved
            // If it's not saved, then run the save procedure. Otherwise, return to
the arranger and tell the user that the clip is already saved
            try
            {
                if (clips[selectedClipIndex].IsSaved())
                {
                    DisplayArrangement(ref username, 2);
                }
                // Clip cannot be saved again
                else
                {
                    clips[selectedClipIndex].SaveClip();
                    InsertClipUses();
                    DisplayArrangement(ref username, 0);
                }
            }
            catch
            {
                DisplayArrangement(ref username, 3);
            }
        }

        /// <summary>
        /// This inserts a record into the ClipUses table using SQL
```

```
        /// </summary>
        private void InsertClipUses()
        {
            int clipID = clips[selectedClipIndex].GetID();
            MySqlCommand usesCommand = Program.SqlConnection.CreateCommand();
            usesCommand.CommandText = "insert into ClipUses (FK_ArrangementID,
FK_ClipID) values (" + ID + ", " + clipID + ")";
            Program.OpenConnection();
            usesCommand.ExecuteNonQuery();
            Program.SqlConnection.Close();
        }

        /// <summary>
        /// If the user wants to load a clip, this method displays the clips saved
under the user's PK_AccountID, and lets them choose
        /// The clip data is then loaded using SQL and a clip object is created and
added to the list.
        /// </summary>
        /// <param name="mode">This is a code indicating whether or not the method
should display an error.</param>
        private void LoadClip(ref string username, bool mode)
        {
            MySqlCommand command = Program.SqlConnection.CreateCommand();
            command.CommandText = "select PK_ClipID, ClipName from Clips where
FK_AccountID = " + Program.CurrentUserID;
            MySqlDataReader reader;
            // Borrowing the ArrangementRow struct as used in program
            List<Program.ArrangementRow> list = new List<Program.ArrangementRow>();
            Program.ArrangementRow newRow;
            // Read list of arrangements from table
            Program.OpenConnection();
            reader = command.ExecuteReader();
            // Loop until all arrangement names have been retrieved and store each row
in the list
            while (reader.Read())
            {
                newRow.ID = reader.GetInt32(0);
                newRow.Name = reader.GetString(1);
                list.Add(newRow);
            }
            Program.SqlConnection.Close();

            List<Program.Pointer> pointers = new List<Program.Pointer>();
            Program.Pointer newPointer;
            Console.Clear();
            Program.WriteLineCyan("                                    Load Clip\n");
            // The load clip wizard is displayed to the user
            // If there are no clips saved to the user's account, then an error is
shown and they are prompted to press any key to return to the arranger
            if (list.Count == 0)
            {
                Program.WriteLineRed("\nThere are no clips saved to this
account.\nPress any key to return to the arranger.");
                Program.WriteYellow("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\nLogged in
as " + username + " ");
                Console.ReadKey();
                DisplayArrangement(ref username, 0);
            }
            // If there are clips saved to the account, then the name of each one is
displayed in a numbered list
            else
            {
```

```
                // Print each of the arrangement names in the list
                for (int i = 0; i < list.Count(); i++)
                {
                    // Must be +1 to prevent 0-indexing
                    newPointer.ListItem = i + 1;
                    newPointer.ID = list[i].ID;
                    pointers.Add(newPointer);
                    Program.WriteLineGreen((i + 1) + ". " + list[i].Name);
                }
                int choice;
                try
                {
                    if (mode)
                    {
                        Program.WriteCyan("\nInvalid choice. Please try again: ");
                    }
                    else
                    {
                        Program.WriteCyan("\nPlease enter a clip number to load: ");
                    }
                    choice = Convert.ToInt32(Console.ReadLine());
                    // Translate choice number into its corresponding arrangement ID
                    int id = new int();
                    // Flag is needed to check if the choice is valid
                    bool flag = false;
                    for (int i = 0; i < pointers.Count; i++)
                    {
                        if (pointers[i].ListItem == choice)
                        {
                            id = pointers[i].ID;
                            flag = true;
                            break;
                        }
                    }
                    // If the choice is invalid, prompt to choose the clip again
                    if (!flag)
                    {
                        LoadClip(ref username, true);
                    }
                    // If the choice is valid, create a new clip object and load the
data from the database
                    else
                    {
                        Clip newClip = new Clip();
                        newClip.LoadClipWithSingleKey(ref id);
                        clips.Add(newClip);
                        DisplayArrangement(ref username, 0);
                    }
                }
                catch
                {
                    // If the user makes an invalid input, run the function again but
with showing an error
                    LoadClip(ref username, true);
                }
            }
        }

        /// <summary>
        /// This deletes a clip within the list at the currently-selected index
        /// </summary>
        private void DeleteClip(ref string username)
```

```
        {
            // Delete clip using index
            clips.RemoveAt(selectedClipIndex);
            DisplayArrangement(ref username, 0);
        }

        /// <summary>
        /// When creating a new arrangement, this serialises the initial arrangement
data (tonic note, major/minor and octave) and sets up a new record in the Arrangements
table
        /// </summary>
        public void CreateArrangement()
        {
            // Will store encoded arrangement data
            // Encodes main arrangement data first
            string encoded = "0" + tonicNoteNumerical + "1";
            // If the key is major, then a C is added. If it is minor, then a D is
added
            if (major)
            {
                encoded += "C";
            }
            else
            {
                encoded += "D";
            }
            encoded += "2" + timeSignature;
            // Create statement to insert data
            MySqlCommand insertCommand = Program.SqlConnection.CreateCommand();
            // Create statement to get the ID of the arrangement
            MySqlCommand getID = Program.SqlConnection.CreateCommand();
            MySqlDataReader IDreader;
            insertCommand.CommandText = "insert into Arrangements (ArrangementName,
ArrangementData, FK_AccountID) values (\"" + name + "\", \"" + encoded + "\", " +
Program.CurrentUserID + ")";
            getID.CommandText = "select LAST_INSERT_ID()";
            // Insert data
            Program.OpenConnection();
            insertCommand.ExecuteNonQuery();
            IDreader = getID.ExecuteReader();
            while (IDreader.Read())
            {
                ID = IDreader.GetInt32(0);
            }
            Program.SqlConnection.Close();
        }

        /// <summary>
        /// If the user wants to save their arrangement, this method serialises the
clip and note data and inserts it into the Arrangments table using SQL.
        /// It uses very similar code to the CreateArrangement method above, but
focuses on updating the data rather than creating a new record.
        /// </summary>
        public void SaveArrangement(ref string username)
        {
            // Variabes to be inserted into database
            string arrangementData;
            string clipContentsData = "";

            arrangementData = "0" + tonicNoteNumerical + "1";
            if (major)
            {
```

```
                arrangementData += "C";
            }
            else
            {
                arrangementData += "D";
            }
            arrangementData += "2" + timeSignature;

            int clipID;

            foreach (Clip clip in clips)
            {
                // If the clip is saved, then we make a reference to it
                if (clip.IsSaved())
                {
                    clipID = clip.GetID();

                    // Compound ID formed of arrangement ID and clip ID together,
which gets inserted into the clip data, preceded by an A, to show that a saved clip
has been included
                    string compoundID = clipID.ToString() + ID.ToString();
                    arrangementData += "A" + compoundID;
                }
                // If the clip is not saved, insert it into the ClipContentsData field
                else
                {
                    // Generates a random key
                    // If the key has already been implemented within the arrangement
data, then generate a new one, and keep looping until you find a unique key
                    Random random = new Random();
                    int randomNumber;
                    string newKey;
                    bool flag = true;
                    do
                    {
                        // We randomly generate a key and keep doing it until the key
is unique
                        randomNumber = random.Next();
                        newKey = "E" + randomNumber + "F";
                        if (!arrangementData.Contains(newKey))
                        {
                            flag = false;
                        }
                    } while (flag);


                    // Append the new data onto the existing data
                    // The key for the clip is added to both the arrangementData, and
the clipContentsData, to cross-reference each other
                    arrangementData += "B" + newKey;
                    // The clip's note data is serialised and inserted into the
clipContentsData variable
                    clip.EncodeClip();
                    clipContentsData += newKey + clip.GetEncoded();

                    }
                }

            // Create new commands to update the data in the arrangement table
            MySqlCommand updateData = Program.SqlConnection.CreateCommand();
```

```
            updateData.CommandText = "update Arrangements set ArrangementData = \"" +
arrangementData + "\", ClipContentsData = \"" + clipContentsData + "\" where
PK_ArrangementID = " + ID;
            // Execute query
            Program.OpenConnection();
            updateData.ExecuteNonQuery();
            Program.SqlConnection.Close();

            // Display success message and display the arrangement again
            System.Diagnostics.Stopwatch stopwatch = new
System.Diagnostics.Stopwatch();
            Console.Clear();
            Program.WriteLineGreen("\n\n\n\n\n\n\n\n\n\n\n\n
Arrangement saved");
            stopwatch.Start();
            // Wait for 1 second before showing the arranger
            while (stopwatch.ElapsedMilliseconds < 1000)
            {
                // Do nothing
            }
            stopwatch.Stop();
            DisplayArrangement(ref username, 0);
            }
        }
  }
```

A2.2.4 Clip Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using MySql.Data.MySqlClient;

namespace ProjectGracenoteAlpha1Point1
{
    /// <summary>
    /// This class is used as a template for the clip object.
    /// Its behaviour includes the methods to create notes, load note data from the
database and display them.
    /// The note generation algorithm is contained in the CreateClip method.
    /// </summary>
    class Clip
    {
        /// <summary>
        /// These are the properties used for each clip, including the note list
        /// </summary>
        private int rangeOfOctaves;
        private int rangeOfNotes;
        private int syncopationRandomisation;
        private List<Note> notes;
        private static short[] accidentals;
        private string name;
        private bool saved;
        private string encoded;
        // This array is used for translating the note value codes into their
respective note values in decimal form. E.g. one means whole note and 0.0625 means
1/16 note
        private static double[] fractionNoteValues = { 1, 0.75, 0.5, 0.25, 0.125,
0.0625 };
        private int ID;
```

```
        private static short maxNoteValue;
        private static short minNoteValue;

        /// <summary>
        /// This the constructor which sets the initial values of all of the variables
        /// </summary>
        public Clip()
        {
            accidentals = new short[7];
            // Assign accidental values
            accidentals[0] = 4;
            accidentals[1] = 1;
            accidentals[2] = 5;
            accidentals[3] = 2;
            accidentals[4] = 6;
            accidentals[5] = 3;
            accidentals[6] = 7;
            saved = false;
            // Instantiate the notes list
            notes = new List<Note>();
            encoded = "";
            ID = 0;
            //IsIncluded = false;

            // ONLY CHANGE THESE
            minNoteValue = 0;
            maxNoteValue = 5;
        }

        /// <summary>
        /// This is the access modifier, which lets the other classes change the
properties of the clip
        /// </summary>
        public void SetDetails(ref int RangeOfOctaves, ref int RangeOfNotes, ref int
SycopationRandomisation)
        {
            rangeOfOctaves = RangeOfOctaves;
            rangeOfNotes = RangeOfNotes;
            syncopationRandomisation = SycopationRandomisation;
        }

        // This returns the flag showing if the clip is saved or unsaved
        public bool IsSaved()
        {
            return saved;
        }

        //  This returns the name fo the clip
        public string GetName()
        {
            return name;
        }

        //
        /// <summary>
        /// This method generates the notes.
        /// It generates note values randomly one at a time.
        /// If a note is outside the octave range allowed, it is rejected, and another
is created.
        /// If the note needs to be sharpened or flattened according to the key
signature, then the pitch value has a 1 or 2 added to the start.
```

```
        /// The method checks if there is enough space left within the clip. It will
keep generating notes until all of the notes have note values that fit within the
space allowed. There are not rests.
        /// This is based on the note generating pseudocode, which can be found in the
Design, Section 3.3.2, although it does not necessarily operate in the same manner
        /// </summary>
        /// <param name="sharps">The method needs to know if the key signature uses
sharps or flats</param>
        /// <param name="major">and if the key is major or minor.</param>
        public void CreateClip(ref bool sharps, ref bool major)
        {
            // This creates an array storing two values - the lowest octave available
is 4 minus the range of octaves input, and the highest octave available is 4 plus the
range of octaves
            // So if the range of octaves is 1, the lowest octave is 3 and the highest
is 5
            int[] octaveRangeAvailable = { 4 - rangeOfOctaves, 4 + rangeOfOctaves };
            // This variable is used for the note value of the highest note available
            int highestNoteAvailable;
            // Copy the tonic note (the main note without any accidentals) from the
Arrangement class
            string mainTonicNote = Arrangement.tonicNoteNumerical[1].ToString();
            // This varible takes the main tonic note (if the tonic note is a C#, the
main tonic note is a C) as an integer so it can be compared
            int TonicNoteNumerical = Convert.ToInt32(mainTonicNote);
            // If the tonic note is a C, make the highest note a B
            if (TonicNoteNumerical == 1)
            {
                highestNoteAvailable = 7;
            }
            // If it's not a C, make the highest note the note below the tonic note
            // So if the tonic note is an E (3), the highest note available is a D (2)
            else
            {
                highestNoteAvailable = TonicNoteNumerical - 1;
            }
            // This variable gets the number of accidentals of the key signature by
checking how many there are in the arrangement's key signature
            int numberAccidentals = Arrangement.GetNumberAccidentals(ref sharps, ref
major);
            Random randomGenerator = new Random();
            // The note value to use (note length) is stored and is changed if the
syncopation randomiser function (which is further below) decides if it needs changing
            int previousNoteValue = randomGenerator.Next(minNoteValue, maxNoteValue);
            // This array will hold the pitch values of the notes that need sharpening
or flattening
            int[] accidentalsToApply = new int[numberAccidentals];
            // If the key signature contains sharps, apply sharp accidentals
            // This is done by adding the accidental pitch values to the
accidentalsToApply array in forwards order (as that is how sharps are applied in key
signatures)
            if (sharps)
            {
                for (int i = 0; i < numberAccidentals; i++)
                {
                    accidentalsToApply[i] = accidentals[i];
                }
            }
            // Otherwise, if the key contains flats, apply flat accidentals
            // This is done by adding the accidental pitch values to the
accidentalsToApply array in reverse order
            else
```

```
            {
                short counter = 0;
                for (int i = 6; i >= 7 - numberAccidentals; i--)
                {
                    accidentalsToApply[counter] = accidentals[i];
                    counter++;
                }
            }
            // This variable stores the amount of space left for the notes
            // The intial value is the number of the key signature (if the key
signature is 4, then the space left is 4)
            double spaceLeft = Arrangement.GetTimeSignature();
            // The spaceLeft is taken away using the decimal note values each time a
note is added
            double spaceToTakeAway;
            // These variables are used as properties for a new note
            int pitch;
            int octave;
            int noteValue;
            string pitchString;
            // This is the note object variable that we will be using to create a new
note and add it to the list
            Note newNote;
            // This loop will decide the properties for a new note
            // It will then check if the note is valid according to the arrangement's
key and time signature
            // If it is invalid, the process starts again
            // If it is valid, the new note is added to the clip list
            do
            {
                // A pitch is randomly selected, from the note C (1) up to B (7)
                pitch = randomGenerator.Next(1, 7);
                // The octave is randomly selected from 1 to 10
                octave = randomGenerator.Next(1, 10);
                // If the syncopation randomiser decides that the note value needs
changing
                if (syncopationRandomiser())
                {
                    // Change the note value
                    noteValue = randomGenerator.Next(minNoteValue, maxNoteValue);
                }
                // If it decides not
                else
                {
                    // Use the note value of the previous note
                    noteValue = previousNoteValue;
                }
                // Adding of the note will only be processed further if there is
enough space left, using the spaceLeft variable to check
                if (spaceLeft >= fractionNoteValues[noteValue])
                {
                    // It will only go further if the octave is greater or equal to
the lowest octave available, and less than or equal to the highest octave available
                    if ((octave >= octaveRangeAvailable[0]) && (octave <=
octaveRangeAvailable[1]))
                    {
                            // If the note generated needs sharpening or flattening
according to the key signature
                            if (accidentalsToApply.Contains(pitch))
                            {
                                // If the key signature uses sharps, a 1 is added to
the start of the pitch to show that it is sharp
```

```
                                if (sharps)
                                {
                                    pitch = pitch + 10;
                                }
                                // If the key signature uses flats, a 2 is added to
show the pitch is flat
                                else
                                {
                                    pitch = pitch + 20;
                                }

                            }
                            // Once note's properties are decided, add it to the notes
list
                            // If the note is natural, then a 0 needs to be added to
the end, as the integer value doesn't start with 0
                            if (pitch - 10 < 0)
                            {
                                pitchString = "0" + pitch.ToString();
                            }
                            // Otherwise, the value is just converted
                            else
                            {
                                pitchString = pitch.ToString();
                            }
                            // We store the note value as the previous note value so
that we can use it for the next note
                            previousNoteValue = noteValue;
                            // A new note object is created and the properties are set
                            newNote = new Note();
                            newNote.SetDetails(pitchString, noteValue, octave);
                            // We add this new note object to the note list
                            notes.Add(newNote);
                            // Reduce amount of space left so that new notes can be
made
                            spaceToTakeAway = fractionNoteValues[noteValue];
                            spaceLeft = spaceLeft - spaceToTakeAway;
                        }
                    }
                } while (spaceLeft > 0);
            }

            // This function decides if the note value should be changed
            private bool syncopationRandomiser()
            {
                Random random = new Random();
                // A random number is generated between 1 and the syncopation
randomisation value that the user chose
                int randomNumber = random.Next(1, syncopationRandomisation);
                // If the number is one, then return true
                if (randomNumber == 1)
                {
                    return true;
                }
                // If not, return false
                else
                {
                    return false;
                }
            }

            /// <summary>
```

```
        /// Leads to EncodeClip and SendToDatabase
        /// </summary>
        public void SaveClip()
        {
            // User prompted to enter clip name
            bool flag = false;
            string text = "                        Enter a name for this clip:";
            do
            {
                Console.Clear();
                Program.WriteLineCyan("                                    Save
Clip\n\n\n\n\n\n\n\n\n\n");
                Program.WriteLineCyan(text+"\n");
                Console.Write("           ");
                string nameInput = Console.ReadLine();
                if (nameInput == "")
                {
                    text = "                     Please don't enter a blank name:";
                }
                else
                {
                    flag = true;
                    name = nameInput;
                    saved = true;
                }
            } while (!flag);

            // Once name is entered properly, encode the clip ready for the database
            EncodeClip();

            // Mark the clip as saved
            saved = true;

            // Send to database
            SendToDatabase();
        }

        /// <summary>
        /// This method serialises the note data for each note in the clip into the
correct characters, so that it is ready for inserting using the SQL.
        /// </summary>
        public void EncodeClip()
        {
            // Loops through each note in the list in order to encode it
            foreach (Note note in notes)
            {
                // Mark start of note with X
                encoded += "X";
                // Mark note value with A
                encoded += "A";
                // Insert note's note value code
                encoded += note.GetNoteValue();
                // Mark note pitch with B
                encoded += "B";
                // Insert note's pitch code
                encoded += note.GetPitchNumerical();
                // Mark note octave with C
                encoded += "C";
                // Insert note's octave
                encoded += note.GetOctave();
                // Mark end of note with Y
                encoded += "Y";
```

```
            }
        }

        // This returns the serialised string that we created above
        public string GetEncoded()
        {
            return encoded;
        }

        /// <summary>
        /// This displays the clip to the user.
        /// It serialises the notes into their respective octaves so that they can be
processed for printing to the console.
        /// This method makes use of the Octave class to do this.
        /// </summary>
        /// <param name="timeSignature">The time signature is needed to determine how
many beats are displayed at the top of the screen</param>
        public void DisplayClip(ref int timeSignature)
        {
            // Check which is the highest and lowest octaves within the notes list
            // Initially set these variables to the values of the first note. They
will be changed later
            int highestOctave = notes[0].GetOctave();
            int lowestOctave = notes[0].GetOctave();
            int currentOctave;

            // Loop through all of the other clips
            for (int i = 1; i < notes.Count; i++)
            {
                // Select the octave of the current note
                currentOctave = notes[i].GetOctave();
                // If the octave of the current note is higher than the highest octave
value, set that as the highest octave
                if (currentOctave > highestOctave)
                {
                    highestOctave = currentOctave;
                }
                // If the current octave is lower than the lowest octave value, set
that as the lowest octave
                else if (currentOctave < lowestOctave)
                {
                    lowestOctave = currentOctave;
                }
                // This will loop so that we have the lower and upper octave limits
            }

            Octave octave = new Octave();
            // We get the serialised note position data using the function in the
Octave class
            string toDisplay = octave.GetNoteDisplay(ref highestOctave, ref
lowestOctave, ref notes);

            // We clear the console
            Console.Clear();
            // If the clip is unsaved, we just say UNSAVED CLIP at the top, otherwise
we show the clip's name
            if (this.name == null)
            {
                Program.WriteLineCyan("UNSAVED CLIP");
            }
            else
            {
```

```
                Program.WriteLineCyan(this.name);
            }
            Console.WriteLine();
            Console.Write("       ");
            // We print the bar indicators at the top of the viewer
            for (int i = 1; i <= timeSignature; i++)
            {
                // We print this spanning divider for the number of the time signature
                Program.WriteGrey("<------" + i + "------->");
            }
            Console.WriteLine();
            // We loop through each character in the serialised note position data
            foreach (char i in toDisplay)
            {
                Console.BackgroundColor = ConsoleColor.Black;
                switch (i)
                {
                    // If the note is natural, print a blue space
                    case '#':
                        Console.BackgroundColor = ConsoleColor.Blue;
                        Console.Write(" ");
                        break;
                    // If the note is sharp, orint a green space
                    case '/':
                        Console.BackgroundColor = ConsoleColor.Green;
                        Console.Write(" ");
                        break;
                    // If the note is flat, print a red space
                    case '*':
                        Console.BackgroundColor = ConsoleColor.Red;
                        Console.Write(" ");
                        break;
                    // Otherwise, print whatever is there
                    default:
                        Console.Write(i);
                        break;
                }
            }
            // Show the instructions below the notes
            Program.WriteCyan("\nNatural notes are ");
            Program.WriteBlue("blue");
            Program.WriteCyan(", sharp notes are ");
            Program.WriteGreen("green");
            Program.WriteCyan(" and flat notes are ");
            Program.WriteRed("red");
            Program.WriteCyan(".\n");
            Program.WriteMagenta("\nPress any key to return to the arranger ");
            Console.ReadKey();
        }

        /// <summary>
        /// This method usese given note data in order to create the note objects and
populate the notes list.
        /// </summary>
        /// <param name="data">This is the serialised note data that we want to
split.</param>
        public void LoadClipWithData(ref string data, bool mode)
        {
            // If the clip is an unsaved one, we mark the clip as unsaved
            if (mode)
            {
                saved = false;
```

```
            }
            // Then, we continue with extracting the data
            // Flags to indicate if each part of the note is found
            bool startFound = false;
            bool noteValueFound = false;
            bool pitchFound = false;
            bool octaveFound = false;
            // Properties for the new note
            int noteValue = new int();
            string pitch = "";
            string octaveString = "";
            int octave = new int();
            // Searches for
            for (int i = 0; i <= data.Length; i++)
            {
                // We should consider each possible combination

                /// NEW NOTE ENCODING - ignore those in the Design section
                /// X - start of note
                /// A - note value
                /// B - pitch
                /// C - octave
                /// Y - end of note

                // If the start has not been found
                if (!startFound)
                {
                    // If the character is an X, then the start has been found
                    if (data[i] == 'X')
                    {
                        startFound = true;
                    }
                }
                // If the start has been found but not the note value
                else if (startFound && !noteValueFound)
                {
                    // If we have found an A, then we have found the note value
                    if (data[i] == 'A')
                    {
                        // We move to the next index, which should be a single integer
representing the note value code
                        i++;
                        // Once again, we need to convert out of the ASCII code into
the actual note value code
                        noteValue = data[i] - 48;
                        // We now mark the note value as found
                        noteValueFound = true;
                    }
                }
                // If the start and note value have been found but not the pitch
                else if (startFound && noteValueFound && !pitchFound)
                {
                    // If we have found an B, then we have the two-digit pitch
                    if (data[i] == 'B')
                    {
                        // We move forward to find the two digits of the pitch code
                        i++;
                        pitch += data[i];
                        i++;
                        pitch += data[i];
                        // We now mark the pitch as found
                        pitchFound = true;
```

```
                    }
                }
                // If the start, note value and pitch have been found but not the
octave
                else if (startFound && noteValueFound && pitchFound && !octaveFound)
                {
                    // If we have found a C, then we have found the octave
                    if (data[i] == 'C')
                    {
                        // Unlike the other properties, the octave can be any length,
so we need to find F
                        do
                        {
                            i++;
                            if (data[i] == 'Y')
                            {
                                // If we have found the Y, then stop working
                                break;
                            }
                            else
                            {
                                // Otherwise, start to store the octave value, first
as a string
                                octaveString += data[i];
                            }
                        } while (true);
                        // Then convert value to an integer
                        octave = Convert.ToInt32(octaveString);
                        // Flag that we have found the octave
                        octaveFound = true;
                    }
                }
                // If we have reached the end of the note, then set all flags to false
                else
                {
                    startFound = false;
                    noteValueFound = false;
                    pitchFound = false;
                    octaveFound = false;
                    // Create new note and add to the note list
                    Note newNote = new Note();
                    newNote.SetDetails(pitch, noteValue, octave);
                    notes.Add(newNote);
                    // Mark properties in this function as empty
                    noteValue = new int();
                    pitch = "";
                    octaveString = "";
                    octave = new int();
                    if (i < data.Length)
                    {
                        i--;
                    }
                }
            }
        }

        /// <summary>
        /// This method is used if we want to load a clip from the database.
        /// It loads the name and note data from the Clips table and passes the note
data to the LoadClipWithData method.
        /// </summary>
```

```
        /// <param name="selectedID">This is the PK_ClipID that the user has
chosen.</param>
        public void LoadClip(ref int selectedID)
        {
            do
            {
                // Retrieve clip data from
                MySqlCommand select = Program.SqlConnection.CreateCommand();
                ID = selectedID;
                select.CommandText = "select ClipName, NoteData from Clips where
PK_ClipID = " + ID;
                MySqlDataReader reader;
                string noteData = "";
                Program.OpenConnection();
                reader = select.ExecuteReader();
                while (reader.Read())
                {
                    name = reader.GetString(0);
                    noteData = reader.GetString(1);
                }
                Program.SqlConnection.Close();
                LoadClipWithData(ref noteData, false);
            } while (true);
        }

        /// <summary>
        /// This method is used where we want to load a saved clip, so it checks the
ClipUses table to link the clip to its arrangement.
        /// </summary>
        /// <param name="key">This the key of the chosen clip.</param>
        public void LoadClipWithKey(ref string key)
        {
            saved = true;
            // Check which field to load data from by checking primary key
            MySqlCommand command = Program.SqlConnection.CreateCommand();
            command.CommandText = "select FK_ClipID from ClipUses where
concat(FK_ClipID, FK_ArrangementID) = " + key;
            MySqlDataReader clipIDReader;
            int clipID = new int();
            Program.OpenConnection();
            clipIDReader = command.ExecuteReader();
            while (clipIDReader.Read())
            {
                clipID = clipIDReader.GetInt32(0);
            }
            Program.SqlConnection.Close();

            // Now load the clip name and data from the clips table, using the key
            string clipName = "";
            string noteData = "";
            MySqlCommand getData = Program.SqlConnection.CreateCommand();
            getData.CommandText = "select ClipName, NoteData from Clips where
PK_ClipID = " + clipID;
            MySqlDataReader dataReader;
            Program.OpenConnection();
            dataReader = getData.ExecuteReader();
            while (dataReader.Read())
            {
                clipName = dataReader.GetString(0);
                noteData = dataReader.GetString(1);
            }
```

```
            // Set the name of the clip
            name = clipName;

            // Send data to be decoded in other function
            LoadClipWithData(ref noteData, false);
        }

        /// <summary>
        /// This method loads note data from the Clips table but uses the PK_ClipID
value for the clip rather than the compound primary key in the ClipUses table
        /// </summary>
        public void LoadClipWithSingleKey(ref int clipID)
        {
            // Now load the clip name and data from the clips table
            string clipName = "";
            string noteData = "";
            MySqlCommand getData = Program.SqlConnection.CreateCommand();
            getData.CommandText = "select ClipName, NoteData from Clips where
PK_ClipID = " + clipID;
            MySqlDataReader dataReader;
            Program.OpenConnection();
            dataReader = getData.ExecuteReader();
            while (dataReader.Read())
            {
                clipName = dataReader.GetString(0);
                noteData = dataReader.GetString(1);
            }

            // Set the name of the clip
            name = clipName;

            // Send data to be decoded in other function
            LoadClipWithData(ref noteData, false);

            saved = true;
        }

        /// <summary>
        /// If we are saving a clip, this method inserts the serialised data into the
Clips table using SQL
        /// </summary>
        private void SendToDatabase()
        {
            // When clip is not already saved, insert into new record
            // Otherwise, update record
            MySqlCommand insertClips = Program.SqlConnection.CreateCommand();
            insertClips.CommandText = "insert into Clips (ClipName, NoteData,
FK_AccountID) values (\"" + name + "\", \"" + encoded + "\", " + Program.CurrentUserID
+ ")";
            if (ID == 0)
            {
                // Get the PK_ClipID of the clip once it's been inserted
                MySqlCommand getID = Program.SqlConnection.CreateCommand();
                MySqlDataReader IDreader;
                getID.CommandText = "select LAST_INSERT_ID()";
                Program.OpenConnection();
                insertClips.ExecuteNonQuery();
                IDreader = getID.ExecuteReader();
                while (IDreader.Read())
                {
                    ID = IDreader.GetInt32(0);
                }
```

```
                Program.SqlConnection.Close();
            }
            else
            {
                insertClips.CommandText = "update Clips set NoteData = \"" + encoded +
"\" where PK_ClipID = " + ID;
                Program.OpenConnection();
                insertClips.ExecuteNonQuery();
                Program.SqlConnection.Close();
            }
        }

        // This returns the PK_ClipID of the clip to another class
        public int GetID()
        {
            return ID;
        }
    }
}
```

A2.2.5 Note Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ProjectGracenoteAlpha1Point1
{
    /// <summary>
    /// This class is used as a template for the note object.
    /// It holds the note's properties.
    /// </summary>
    class Note
    {
        // Properties
        private string pitchNumerical;
        private int noteValue;
        private int octave;

        // Constructor
        public Note() { }

        // Access modifier, used for setting the values of the properties
        public void SetDetails(string Pitch, int NoteValue, int Octave)
        {
            pitchNumerical = Pitch;
            noteValue = NoteValue;
            octave = Octave;
        }

        /// <summary>
        /// These functions are all used to return the properties
        /// </summary>

        public string GetPitchNumerical()
        {
            return pitchNumerical;
        }

        public int GetNoteValue()
```

```
        {
            return noteValue;
        }

        public int GetOctave()
        {
            return octave;
        }
    }
}
```

A2.2.6 Octave Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ProjectGracenoteAlpha1Point1
{
    /// <summary>
    /// This class is used for processing each note so that it is displayed in the
octaves correctly, for when the clip is displayed to the user.
    /// </summary>
    class Octave
    {
        // This struct contains holders of notes of one octave
        private struct OctaveOfNotes
        {
            public string[] Notes;
        }

        // Constructor
        public Octave() { }

        // This struct contains a number of an octave and its index in the list of
octaves later
        private struct Pointer
        {
            public int Index;
            public int Octave;
        }

        /// <summary>
        /// This method uses the list of notes from the Clips class and serialises
them into their positions in the various octaves.
        /// Each octave is stored as one string. Each string is in an array. The array
starts with the tonic note and octave number, so if it was G, octave 4, it would start
with G4.
        /// Then, the position of each note is marked by either a #, / or *,
representing whether the note is natural, sharp or flat, respectively.
        /// Each string is concatenated into one string, which is returned to the Clip
class.
        /// When the DisplayClip method in the clip class detects a #, it shows a blue
space. When it sees a / it prints green. When it sees a *, it prints red.
        /// </summary>
        public string GetNoteDisplay(ref int highestOctave, ref int lowestOctave, ref
List<Note> notes)
        {
            string toDisplay = "";
```

```
            // List to hold all of the octaves required
            List<OctaveOfNotes> listOfOctaves = new List<OctaveOfNotes>();

            // We keep a value of the number of octaves. We keep is a 1 less than what
the actual number of octaves is so that we can use it as an index pointer for the
arrays later
            int numberOctaves = (highestOctave - lowestOctave) + 1;

            // Generate correct number of octaves
            for (int i = 0; i < numberOctaves; i++)
            {
                // We instantiate a new octave with 7 rows for each of the 7 notes
                OctaveOfNotes newOctave = new OctaveOfNotes();
                newOctave.Notes = new string[7];
                // We add this octave to the list of octaves
                listOfOctaves.Add(newOctave);
            }

            // Assign the correct octave numbers to each octave created
            int count = 0;
            Pointer[] pointers = new Pointer[numberOctaves];
            // Octaves will be listed in descending order
            // We start labelling octaves with the highest octave first going down to
the lowest one
            for (int i = highestOctave; i >= lowestOctave; i--)
            {
                // Give each note row its corresponding note label at the beginning
                listOfOctaves[count].Notes[0] = "G" + i + "     ";
                listOfOctaves[count].Notes[1] = "F" + i + "     ";
                listOfOctaves[count].Notes[2] = "E" + i + "     ";
                listOfOctaves[count].Notes[3] = "D" + i + "     ";
                listOfOctaves[count].Notes[4] = "C" + i + "     ";
                listOfOctaves[count].Notes[5] = "B" + i + "     ";
                listOfOctaves[count].Notes[6] = "A" + i + "     ";

                // Give a pointer to this octave
                pointers[count].Index = count;
                pointers[count].Octave = i;

                // Increment counter for next octave
                count++;
            }
            // Loop through each of the notes and assign them to the correct octave
            foreach (Note currentNote in notes)
            {
                // Variables used for assigning the notes
                int octaveListIndex = new int();
                int noteInOctave = new int();
                int numberSpacesToUse = new int();
                int octave = currentNote.GetOctave();
                // Find which octave array it needs to be assigned to in the list
                for (int i = 0; i < pointers.Length; i++)
                {
                    if (pointers[i].Octave == octave)
                    {
                        octaveListIndex = pointers[i].Index;
                    }
                }
                // Find which pitch within the note needs to be assigned to
                string pitch = currentNote.GetPitchNumerical();
                // Extract accidental and main pitch
```

```
                string accidental = pitch[0].ToString();
                // Change accidental code to #, / and * to avoid confusion with octave
numbers
                switch (accidental)
                {
                    case "0":
                        accidental = "#";
                        break;
                    case "1":
                        accidental = "/";
                        break;
                    case "2":
                        accidental = "*";
                        break;
                }
                string mainNote = pitch[1].ToString();
                // Choosing which index in one of the octaves needs to be assigned to
                switch (mainNote)
                {
                    // If the note is a C
                    case "1":
                        // Cs are stored in index 4 in each of the octaves
                        noteInOctave = 4;
                        break;
                    // If the note is a D
                    case "2":
                        // Ds are stored in index 3 in each of the octaves
                        noteInOctave = 3;
                        break;
                    // If the note is an E
                    case "3":
                        noteInOctave = 2;
                        break;
                    // If F
                    case "4":
                        noteInOctave = 1;
                        break;
                    // If G
                    case "5":
                        noteInOctave = 0;
                        break;
                    // If A
                    case "6":
                        noteInOctave = 6;
                        break;
                    // If B
                    case "7":
                        noteInOctave = 5;
                        break;
                }
                // Decide how many characters to print based on how long the note
value is
                int noteValue = currentNote.GetNoteValue();
                // Extract note value based on its corresponding code
                switch (noteValue)
                {
                    // If the note is 1 beat
                    case 0:
                        // A note that is 1 beat will take up 16 spaces
                        numberSpacesToUse = 16;
                        break;
                    // If the note is 3/4 beat, it will take up 12 spaces
```

```
                    case 1:
                        numberSpacesToUse = 12;
                        break;
                    // If the note is 1/2 beat, it will take up 8 spaces
                    case 2:
                        numberSpacesToUse = 8;
                        break;
                    // 1/4 beat
                    case 3:
                        numberSpacesToUse = 4;
                        break;
                    // 1/8 beat
                    case 4:
                        numberSpacesToUse = 2;
                        break;
                    // 1/16 beat
                    case 5:
                        numberSpacesToUse = 1;
                        break;
                }
                // Print out accidental code for number of spaces needed
                for (int i = 0; i < numberSpacesToUse; i++)
                {
                    // For each row where our note is contained, its accidental code
(#, / or *) will be assigned
                    // For each row where our note is not contained, spaces will be
added
                    for (int j = 0; j < listOfOctaves.Count; j++)
                    {
                        // Find index where the note is located
                        if (j == octaveListIndex)
                        {
                            for (int k = 0; k < 7; k++)
                            {
                                if (k == noteInOctave)
                                {
                                    // If the current position in the row is correct
for the note that we're adding, we add the accidental code to show the note's position
                                    listOfOctaves[j].Notes[k] += accidental;
                                }
                                else
                                {
                                    // Otherwise, we just add a space
                                    listOfOctaves[j].Notes[k] += " ";
                                }
                            }
                        }
                        else
                        {
                            // For every row for the notes other than the one we're
working on, we add a space
                            for (int k = 0; k < 7; k++)
                            {
                                listOfOctaves[j].Notes[k] += " ";
                            }
                        }
                    }
                }
            }
            // Loop through each of the rows and add to toDisplay
            foreach (OctaveOfNotes currentOctave in listOfOctaves)
            {
```

```
                for (int i = 0; i < 7; i++)
                {
                    // Inserts each row separated by line
                    toDisplay += currentOctave.Notes[i] + "\n";
                }
            }
            return toDisplay;
        }
    }
}
```

A2.2 SQL Code

The tables in the database were built using the following data definition language. They were written as a text file and then pasted into the phpMyAdmin server page:

```
/* This creates the Accounts table, where we need an account ID (auto-incrementing primary key), username and password */
create table Accounts
(
        PK_AccountID int not null auto_increment,
        Username varchar(30) not null,
        Password varchar(50) not null,
        primary key (PK_AccountID)
);

/* This creates the Clips table, where we need fields for clip ID (auto primary key), clip name, note data and account ID (which links as a foreign key to the Accounts table) */
create table Clips
(
        PK_ClipID int not null auto_increment,
        ClipName varchar(50) not null,
        NoteData text not null,
        FK_AccountID int not null,
        primary key (PK_ClipID),
        foreign key (FK_AccountID) references Accounts(PK_AccountID)
);

/* This creates the Arrangements table, where we need fields for the arrangement ID (auto primary key), arrangement name, arrangement data, clip contents data and account ID (which links as a foreign key to the Accounts table) */
create table Arrangements
(
        PK_ArrangementID int not null auto_increment,
        ArrangementName varchar(50) not null,
        ArrangementData text not null,
        ClipContentsData text,
        FK_AccountID int not null,
        primary key (PK_ArrangementID),
        foreign key (FK_AccountID) references Accounts(PK_AccountID)
);

/* This creates the clip uses table, which has two foreign keys, clip ID and arrangement ID, which link to the Clips and Arrangements tables respectively. They form a compound primary key.
```

```
This table is used to link a user's clip with its use in an arrangement, when a clip is saved */
create table ClipUses
(
        FK_ClipID int not null,
        FK_ArrangementID int not null,
        primary key (FK_ClipID, FK_ArrangementID),
        foreign key (FK_ClipID) references Clips(PK_ClipID),
        foreign key (FK_ArrangementID) references Arrangements(PK_ArrangementID)
);
```

A3 References

Researching the existing software (Sibelius and MuseScore):

- http://www.avid.com/US/products/SibeliusFirst
- http://www.sibelius.com/products/sibelius_first/features.html
- (Screenshots of Sibelius) https://www.google.co.uk/search?q=sibelius&source=lnms&tbm=isch&sa=X&ved=0CAkQ_AUoA2oVChMIpIS12dyJxgIVpLPbCh25nADu&biw=630&bih=902#tbm=isch&q=sibelius+software&imgrc=Mx_f_o2W1G20GM%253A%3BcXCj4NF-q6mpRM%3Bhttp%253A%252F%252Fregmedia.co.uk%252F2011%252F09%252F17%252Fsibelius_first_software_1.png%3Bhttp%253A%252F%252Fwww.theregister.co.uk%252F2011%252F09%252F19%252Fgeek_treat_of_the_week_m_audio_keystation_mini_32%252F%3B1440%3B900
- http://createdigitalmusic.com/files/2011/07/sibelius7_ui.jpg
- http://regmedia.co.uk/2011/09/17/sibelius_first_software_1.png
- https://www.google.co.uk/search?q=sibelius&source=lnms&tbm=isch&sa=X&ved=0CAkQ_AUoA2oVChMIpIS12dyJxgIVpLPbCh25nADu&biw=630&bih=902#tbm=isch&q=sibelius+ideas+hub
- http://www.sibelius.com/products/sibelius/5/images/ideas_header.jpg
- http://www.sibelius.com/products/sibelius/7/images/Play_Perform_and_Share.png
- http://www.sibelius.com/products/sibelius/5/hub.html
- http://en.wikipedia.org/wiki/MuseScore
- https://musescore.org/en/whats-new-musescore-2
- https://musescore.org/en/node/36161#start-center
- http://www.youtube.com/watch?v=yoMOAIzBSpY
- https://dev.mysql.com/doc/refman/5.0/en/blob.html
- https://en.wikipedia.org/wiki/Binary_large_object
- http://stackoverflow.com/questions/1203710/which-datatype-is-better-to-use-text-or-varchar
- https://dev.mysql.com/doc/refman/5.0/en/storage-requirements.html
- http://i1-win.softpedia-static.com/screenshots/Sibelius_2.png
- https://www.youtube.com/watch?v=0oGqZ201h0U
- https://www.youtube.com/watch?v=x4HtK7QsCEw
- http://musescoredownload.com/wp-content/uploads/2013/08/Musescore-Software.jpg
- http://lrrpublic.cli.det.nsw.edu.au/lrrSecure/Sites/Web/T4U_MuseScore/graphics/MS_New_wizard2_tp.jpg

How to create abstraction models:

- http://www.codeproject.com/Articles/38655/Prevent-Circular-References-in-Database-Design
- http://dba.stackexchange.com/questions/3134/in-sql-is-it-composite-or-compound-keys

How to make an appraisal of alternative solutions:

- https://msdn.microsoft.com/en-us/library/5t6z562c(v=vs.90).aspx

How to use SQL:

- http://stackoverflow.com/questions/5835978/how-to-properly-create-composite-primary-keys-mysql
- http://stackoverflow.com/questions/11476855/invalid-attempt-to-access-a-field-before-calling-read
- http://www.w3schools.com/sql/sql_primarykey.asp
- http://stackoverflow.com/questions/3458046/how-to-include-quotes-in-a-string
- http://stackoverflow.com/questions/8321514/input-string-was-not-in-a-correct-format
- http://stackoverflow.com/questions/3867299/reading-multiple-rows-from-database-where-am-i-going-wrong
- http://stackoverflow.com/questions/6427764/concatenate-two-database-columns-into-one-resultset-column
- http://stackoverflow.com/questions/17112852/get-the-new-record-primary-key-id-from-mysql-insert-query

General advice on how to write the project:

- A2 AQA Computing, by Kevin Bond and Sylvia Langfield, Nelson Thornes, ISBN: 978-0-7487-8298-7

There were more websites used, though unfortunately, many of the links were lost.