Designing and Coding a Note-Taking Program to Aid Sixth-Form Students in Efficient Note-Taking

An EPQ Project
by George Bryant
gb@gbryant.co.uk

February 26, 2014

Contents

Chapter 1

Introduction

A vitally important aspect of success in the sixth form is taking good notes. Without them, revision becomes much more difficult, and success in A-level exams is less certain. Note-taking is essential to provide students with the necessary information to develop their knowledge of the subject further than that which is taught in textbooks.

Most students still take notes on paper, despite the fact that most with a reasonable degree of typing skill can type faster than they can write. I wanted to create a way for students to take notes that would be as efficient as possible, so that those who choose to use a computer to take notes can take the best notes possible. To do this, I had to research what constitutes good notes, how to make good notes easy to take, and how those compare with how students currently take notes.

Chapter 2

Research

2.1 Note-taking Research

Firstly, to make a program to aid efficient note-taking I needed to know what constitutes "efficient note-taking". Due to many differing opinions on optimal note-taking, I tried to use peer-reviewed scholarly sources which detailed the results of a study, as this allowed me to try to obtain more objective information than I could get from most internet sources or books. For example, several of my sources came from the Journal of Educational Psychology, a journal published by the American Psychological Association, which has a masked peer review policy to screen all submissions.

Purdie and Hattie[1] assessed students' conceptions of learning – their ideas of what learning is "for", and compared it to the students' results in simple tests. They identified six conceptions of learning: "Gaining information", "Reading, using and understanding information", "Duty", "Personal change", "Process" and "Social competence". This is similar to Shale and Trigwell's ideas[2] about them, which say that there are also six different conceptions of learning. However, they differ from Purdie and Hattie's identified conceptions of learning in that they split "Reading, using and understanding information" into two - "The acquisition of facts, methods, etc., which can be retained and used when necessary" and "The abstraction of meaning". Shale and Trigwell also include "An interpretative process aimed at understanding reality", instead of Purdie and Hattie's unexplained "Process".

Shale and Trigwell go further, saying that the later conceptions such as "Learning as changing as a person" contain and require the inner conceptions, but not vice versa. Purdie and Hattie do not point out a heirarchy, instead pointing out that students hold multiple conceptions of learning. Their findings showed that "the higher achieving students were those who held multiple conceptions of learning rather than one single or predominant

1. Purdie and Hattie, "Assessing students' conceptions of learning."
2. Shale and Trigwell, "Paper 3: Students' conceptions of learning."

one."

Purdie and Hattie's findings indicate that if possible, my note-taking program should encourage students to hold multiple conceptions of learning, for example by encouraging users to read, use and understand information, rather than simply collect it.

However, trying to make users develop a deep understanding of the material as they take notes can be detrimental. The University of Reading[3] suggested that "note-taking can distract you from listening to lectures". Kiewra et al.[4] pointed out that "during lecture learning, students must continuously and simultaneously listen, select important ideas, hold and manipulate lecture ideas, interpret the information, decide what to transcribe, and record notes." However, while reviewing their notes later, they do not have these pressures on them. In the same study, Kiewra et al. found that those who took notes during a lecture and then reviewed them later (encoding plus storage) outperformed those who only took notes during the lecture (storage only).

Another important aspect of learning is joining ideas together. In another paper[5], Kiewra et al. noted that "when ideas are interconnected, the recall of one idea may prompt the recall of logically associated ideas as well." Shale and Trigwell noted that connecting ideas is a major part of the abstraction of meaning from notes.

Kiewra noted that this improvement could be attributed to the repetition effect (the idea that looking at notes multiple times will improve recall), but even so, reviewing notes did have a significant effect. Furthermore, Kiewra's study also found that in all their tests, the results of encoding only group (ie those who were given another's notes) did not significantly differ from those of the group who were not permitted to take notes. This shows that reviewing notes is a vital part of efficient note-taking only when the notes were made by the person reviewing them.

A study by Vesta et al.[6] showed that when a break between taking notes and being tested on them was provided, it was more effective to review the notes than to take a break doing other activities, which again shows that reviewing notes improves recall. They also found that "subjects who were permitted to take notes recalled significantly more ideas than those who were permitted only to listen", disagreeing with Kiewra et al., who concluded that only taking notes offered no significant improvement over not taking notes.

Most of the studies I have mentioned were only taken by people writing notes. I was concerned that they might be less valid when applied to taking notes using a computer. In simple terms of speed of writing, "research has compared typing speed to writing speed and found evidence that proficient

3. "Effective note-taking."
4. Kiewra et al., "Note-taking functions and techniques."
5. Kiewra et al., "Providing study notes."
6. Di Vesta and Gray, "Listening and note taking."

typists can type faster than they can handwrite" - Brown, 1988.[7] Bui et al.[8] found that "Taking notes using a computer led to better overall test performance compared to taking notes by hand", and that when ordered to take notes with a computer, immediate test scores were better when the lecture was transcribed than when organised notes were taken. Those who transcribed the lecture did better in delayed tests than those who summarised it as long as studying notes was allowed.

This does not, however, mean that I should neglect note organisation in my program. The repetition effect states that looking at notes multiple times will improve recall, but if simply scanning over the notes to reread them, the repetition effect will not be as pronounced as if they are reworded or summarised. For this reason my program should allow rewriting a note while keeping the original, and have some way of linking them together.

The conclusions I drew from my note-taking research were that if possible, my program should aid users in abstracting meaning from their notes, as well as encouraging them to read and use the information. This could be accomplished by allowing users to link their notes together, allowing them to follow a connection for related information. The program should not distract or obstruct users while taking notes, as they already need to do a lot at once. Finally, it should make it easy to review and/or summarise notes later, to aid in the "storage" process.

2.2 Market Research

Once I had an idea of what "efficient note-taking" consisted of, I needed to do some market research to find out how people would use a note-taking app, and what they wanted or expected in one. As my program will be aimed at sixth-form students, I was ideally placed to survey them. To avoid issues with permission, I gave out my questionnaire (See Appendix A on page 24) only in my own school.

Before settling on questions for my main questionnaire, I did a pilot study of roughly 30 students. At that point my project's title included "sixth form and university students", but it became clear when analysing responses to the pilot study that I would not be able to contact enough university students to have a good representation of their views, so I removed them from my title and focused on sixth form students.

The comments on the pilot study had some useful advice. One suggestion was to add profiles for different subjects, which would change settings to be optimal for that subject. I considered it to be a good suggestion, so I added it to the main questionnaire. Another was that keyboard shortcuts would be useful, which prompted me to add a question about them

7. Brown, "Comparison of Typing and Handwriting in "two-finger typists"."
8. Bui, Myerson, and Hale, "Note-taking with computers."

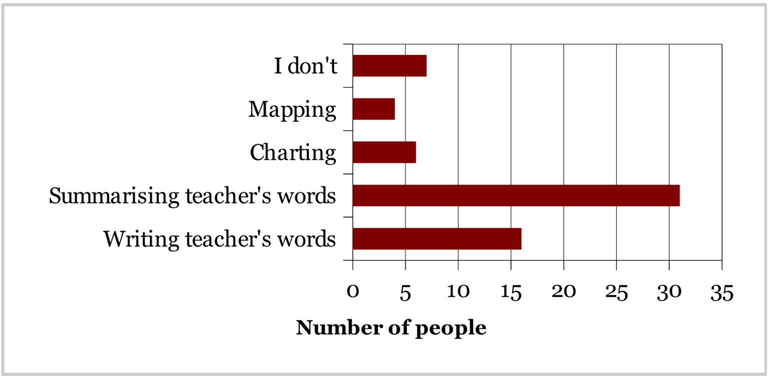


Figure 2.1: Note-taking during lessons

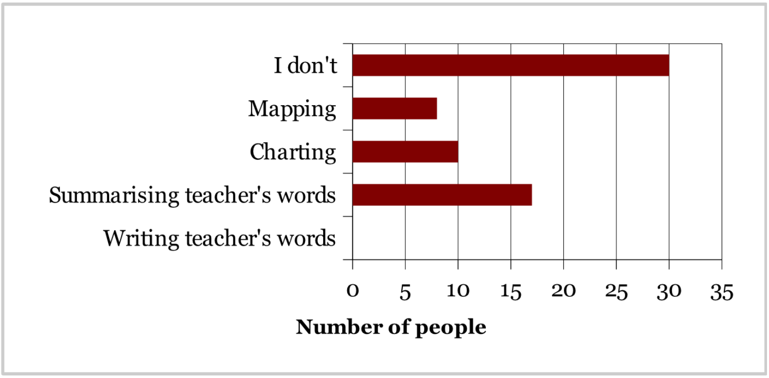


Figure 2.2: Revision after lessons

to the questionnaire. One comment which particularly influenced me was a university student's comment that emphasised the sheer importance of the speed of the program, "as neither teachers nor lecturers stop for you to write something down."

After the pilot study, I gave out the main questionnaire in the sixth form common room in my school, a place the majority of the sixth form frequent. This yielded 56 responses, which I consider adequate as it is over 30% of the sixth form, and the sample was close to a representative sample of both genders, with 14% females, compared to 17% in my school's sixth form.

Figures 2.1, 2.2 and 2.3 show that summarising the teacher's words is the most popular method of note-taking both during and when reviewing after lessons. During lessons, transcribing the teacher's words also ranks highly, although nobody said they used that method after lessons. Mapping and charting were used by few people, so my program needed to be oriented towards text-based approaches to note-taking. This suits me well, as text is significantly easier to work with when programming than tables and maps with complex positioning.

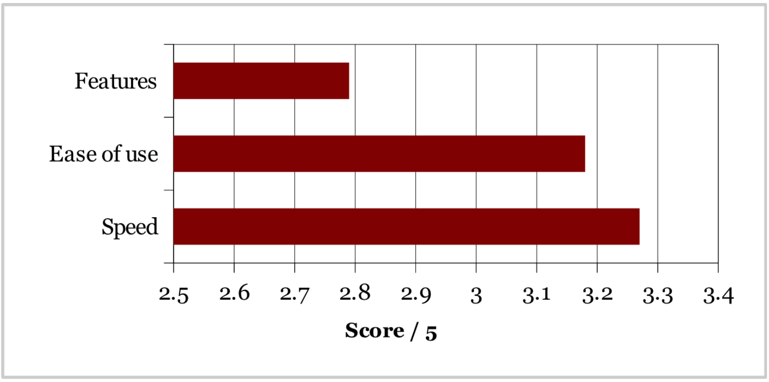


Figure 2.3: Importance of features, ease of use and speed

The fact that the majority of responses said they did not refine their notes after lessons suggests that the people I questioned do not know about the benefits of reviewing their notes, or consider the benefit to be smaller than the effort expended by reviewing them. I decided from this that my program should make it easy to review and summarise notes so that users are more likely to review them.

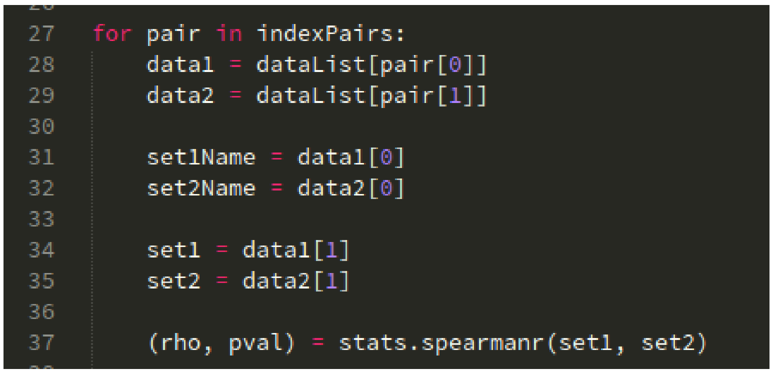
Figure 2.3 compares how important people felt the features, ease of use and speed of a note-taking program are. Speed and ease of use are close together (at 3.27 and 3.18 out of 4 respectively) and people preferred simplicity over features. This showed that I needed to prioritise making the program easy to use and fast over adding complex features.

To see the significance of this data, I wrote a short program in Python to compare the significance of the correlation between people's preference for features, ease of use and speed. What this would show me is whether there is any correlation between, for example, people wanting the program to be fast, and people wanting it to be easy to use. I hypothesised that someone who values the speed of the program will be likely to value the ease of use of the program. My null hypothesis was that there is no correlation between valuing the speed of the program and valuing the ease of use of the program.

For each pair of named sets of values put into it, my Python program will calculate the spearman's rank correlation constant of the pairs. This can be between -1 and 1. -1 would show a perfect negative correlation – If one value was higher, the other was always lower. +1 shows a perfect positive correlation – if one value was higher, the other was also higher. I accomplished this with the use of the `scipy`[9] module (modules explained in the "Planning" section).

A sample of my code is shown in figure 2.4 – this part is after the sets of data have been paired up. The final line of the code shown is what gets the spearman's rank correlation constant(`rho`) of the pairs of data sets. It

9. Jones, Oliphant, and Peterson, "SciPy, Open source scientific tools for Python."



```python
for pair in indexPairs:
    data1 = dataList[pair[0]]
    data2 = dataList[pair[1]]

    set1Name = data1[0]
    set2Name = data2[0]

    set1 = data1[1]
    set2 = data2[1]

    (rho, pval) = stats.spearmanr(set1, set2)
```

Figure 2.4: A sample of my data analysis code

Comparing	Correlation	Probability
“Speed” and “Ease of use”	-0.020696	0.879662
“Speed” and “Features”	0.083429	0.540993
“Ease of use” and “Features”	0.064416	0.637167

Table 2.1: Correlation between aspects of survey results

also returns the probability (`pval`) that this level of correlation was only obtained by chance. The correlation constants and probabilities are shown in table 2.1.

As you can see, all three correlation constants are very close to zero, which means there is unlikely to be any correlation between any of the three factors. The probability that the correlations happened by chance is also very high, so I have to accept my null hypothesis that there is no correlation between valuing the speed of the program and valuing the ease of use of the program.

A more useful way to obtain the data on which aspect of the program people preferred may have been to ask them to put “Speed”, “Ease of use” and “Features” in order of importance. I could also have included some other factors. This would have allowed me to give each factor a score from 1 to [number of factors]. With some more factors included, this would give me a higher resolution scale, and more potential for analysis.

Despite the preference of simplicity over features, 75% of those who responded said they would appreciate text expansion in the program, and 82% said that they would like to have different profiles for different subjects.

Finally, Google Drive was the most popular cloud storage option, and fortunately it also is fairly easy to integrate into a program. I prioritised it lower than features, though, as under 40% of people responding wanted any cloud storage at all.

What I understood from the results of my survey was that I could focus on text-based forms of note-taking such as transcribing the teacher's words, as they are what the majority of my target market uses. I needed to encourage users to review their notes, as most do not, and my research showed that it had benefits. My focus had to be speed and ease of use, as simplicity was preferred over advanced features. When adding advanced features, I had two options that I knew my target market would appreciate: text expansion and subject profiles.

2.3 Interface Research

I needed a small amount of research into how to design a user interface for a program. The user interface (referred to as the UI or the Graphical User Interface (GUI) from now on) is what the user sees when they interact with the a computer program. An example of a UI would be the UI of Windows, consisting of the familiar taskbar, start button and windows with similar layouts - consisting of a bar across the top (for information about the program, like its name) and a box for the program's content.

Unfortunately, interface design is an area which is very subjective and has little useful objective research. For these reasons I chose not to include my interface research in this report, and to merely have a basic but functional UI, which would allow the program to be used as intended, but may not look aesthetically pleasing enough for users to actually use the program in their everyday lives.

Chapter 3

Production

3.1 Planning

Now that I had my research, the next step was to make a specification, to have all the requirements for the program in one place. I combined the two aspects of my research – note-taking and market research – and made the specification which is included in this report as appendix B on page 26.

I needed to ensure that I referred back to my specification often as I made the program, so that I could be sure that the program was meeting its specification and I was not adding unnecessary features. As one of my most important aspects of the program was its speed, I chose to call it NoteVelocity.

Before beginning programming NoteVelocity, I had to choose what format it would come in and what programming language I would use. I had three options for the type of program it could be:

- A traditional desktop program
- A mobile application
- A web application (webapp)

To create a desktop or mobile application, I could use programming languages I already knew something about, and extend my knowledge of them. If I were to make a webapp, I knew how to control the appearance using HTML and CSS, but I would need to learn a new programming language (JavaScript) to make it function. Another advantage of a webapp is that it could be created to work on both desktops and mobile devices.

Eventually I decided on creating a desktop program, because it would allow me to work on an aspect of programming I hadn't used before: Graphical user interface programming (as opposed to text-based), and because creating mobile applications would limit my market. If you create an iOS application for Apple devices, only 54%[1] of the smartphone market – those with iPhone and iPads - can use it, whereas if you create a program for Windows, 91%[2] of the computer market can use it.

The next factor to choose was my programming language – I already knew the basics of some programming languages, but whatever language I chose I would have to learn more. The main languages I could have chosen were:

- C (basics known)
- Python (intermediate level)
- C++
- C#
- Java

Of course, my first choices were C and Python due to knowing the basics of how the languages worked. C is a "lower-level" language than Python, meaning it requires skills such as memory management, whereas Python does these things for you, allowing you to concentrate on what the program needs to do. The trade-off is that Python is slower than C, although it is not noticeable unless doing things like analysing large sets of numbers.

C++ and C# are modified versions of C, and Java is similar to C#. C++, like C, is a "low-level" language, so I chose to avoid it for this reason. My reasons for not choosing C++, or Java are merely personal, as I do not like the style of their code – it is very rigid and requires precision to work properly. One example of this is the syntax of the code – the style the code is written in.

For example, in C#, a function would be defined like this:

```
public static void function(string []arguments) {
  Console.Write('Hello, World!');
  // Do things
}
// rest of the code
```

As you can see, the code of the function is surrounded by curly brackets. Each line within a function has to end in a semicolon, or it will feed into the next line as if there was no space between them. C# and Java are also what is known as "statically typed" languages, which means that variables and functions only contain a specific type of data (be it an integer, a character or something else) and this cannot be changed. Looking at the example, the "public static void function" part of the code says that the function called

1. Epstein, "iOS markets share."
2. Protalinski, "Windows 8 rockets to 7.41% market share."

"function" will return no data to the rest of the program ("void"), can be accessed by other parts of the program ("public") and is not related to a specific instance of the function ("static"). On the other hand, in Python the syntax is a little simpler:

```
def function(arguments):
    print('Hello, World!')
    # Do things
# rest of the code
```

Notice that there is less punctuation cluttering the code – Python is designed to be readable. At first glance, it looks as if there is nothing showing where the functions's code ends and the "# rest of the code" begins. However, an interesting thing to note about Python is that, unlike most programming languages, whitespace matters. Each line of the function is indented by the same amount. When the indentation ends, it shows that the function has ended. This means that it is easy to see which parts of a program contain which other parts by looking at the indentation rather than looking for curly brackets.

Python is also dynamically typed. What this means is that a new variable can be created (for example `variable = 0`) and then a different type of data can be put into it – to continue the example, I could change the variable to be the letter "e" instead of a 0 using `variable = \e"`. This is a difference in style, as in C# or Java you would simply use a differently named variable to store the "e", so it is really only a matter of personal preference.

Because of this simplicity in programming – Python has been described as "executable pseudocode"- I chose to use Python instead of C for my programming. An interesting thing to note is that Python is an interpreted language, whereas C is a compiled language. What this means is that when you run a Python program, there is another program jumping around your code working out which parts to execute. When you run a C program, all this "jumping around" has been done for you by the compiler, which has turned the code into a binary format that the computer can use.

Once I had chosen Python as my programming language, I had to choose what tools to use it with. One important feature of Python is modules, which allow you to use someone else's code in a neat way. For example, if you wanted to display a window on the user's screen, instead of writing your own code to manually create the window you could use a module which already includes the code for drawing the window. This means you do not have to re-use complex code which has already been written by someone else.

One module I needed to use was something to create a graphical user interface (GUI) – the average user certainly does not want to be typing their notes into a black and white terminal window. A user-friendly interface is the best way to get people to use NoteVelocity. With this in mind, I looked for the best GUI module for the task. The popular options I found were:

- Tkinter
- PySide or PyQt
- Kivy
- PyGTK

Tkinter is included in the standard installation of Python on Windows and Mac, but is limited in the styling which can be applied to it. Pyside/PyQt (2 very similar modules) had not been updated for the latest versions of Python, so I could not use them. Kivy looked promising as it worked on Android phones as well as desktop computers, and PyGTK had also not been updated.

I decided the best way to choose between Tkinter and Kivy was to try using them. I began working through the tutorials on the Kivy website. To keep the code simple, Kivy allows you to have code affecting the appearance of your program separate from the code which controls what it does. Unfortunately, to do this Kivy uses a second "miniature programming language", which is fairly complex. This complexity, while perfect for making a game, is too much for a program with a simple appearance, such as a note-taking program. I turned to Tkinter.

As Tkinter is one of Python's default modules, it would make installation simpler for users, as they would not need to find and install Kivy (which can be fairly complex). Another good thing about it is that it is very "Pythonic" – simple and has a certain way of doing things. It does not require learning another language, as it works the same way as many Python modules. Python actually has a style guide called PEP-8[3], which dictates how the code should be formatted. There are extensions for most text editors which format already written code to comply with PEP-8.

There are a lot of online references (for example, Effbot[4] and the offical Python Tkinter reference[5]) to look at examples of code and the functions each part of Tkinter does, as opposed to Kivy's lone reference. I began learning Tkinter by using these references to create a very basic program – simply a text box and some buttons – to get used to the style of code the package used. Figure 3.1 is a screenshot of the basic program I made.

After this, I needed to set up the programs I would be using to write the code for NoteVelocity. As the project began, I was using a text editor called Sublime Text 3, but as it progressed I realised I needed more keyboard

3. Rossum, Warsaw, and Coghlan, "Style Guide for Python Code."
4. Lundh, "An Introduction to Tkinter."
5. "Tkinter - Python interface to Tcl/Tk."

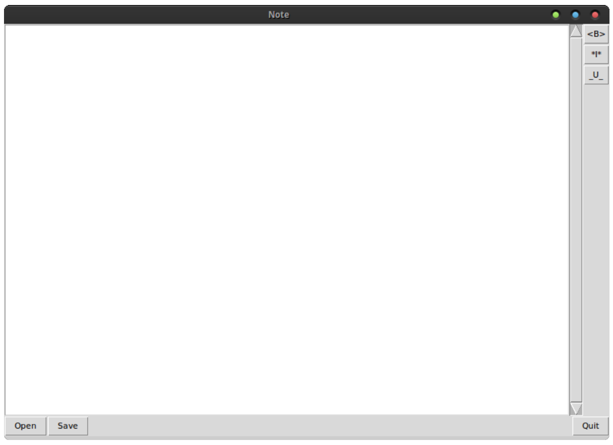


Figure 3.1: Screenshot of basic program made to learn Tkinter

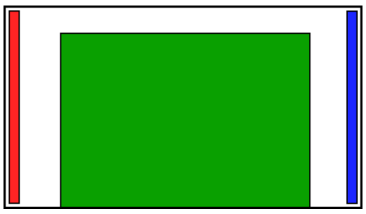


Figure 3.2: Initial program design

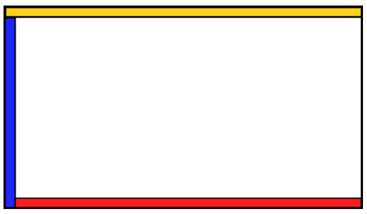


Figure 3.3: Final program design

control to work as quickly as possible, and switched to Vim. Vim was created in 1991 and is a command-line text editor. What this means is that it uses only the characters available in a terminal window to act as its user interface, so it consists only of text. This does not, however, mean that it is limited. It is extremely user-configurable, as scripts can be used to add extra functionality, and with 22 years of people writing scripts for it there are scripts for almost every function imaginable. I used several to highlight the syntax of python code (different types of object are different colours), to test my code, and for many other smaller functions.

My initial design consisted of a virtual page of notes in the centre, with a "tab bar" containing tabs which could be clicked to switch to different notes on the left, and a "formatting bar" which contained options for formatting the text on the right. What the formatting bar contained was functionality such as adding emphasis to a word, or adding an equation – the type of functionality that is usually found in the top bar of programs such as Microsoft Word. Because the virtual page was shown as a page-shaped object – ie not taking up the whole width of the program – the two bars would be outside the boundaries of the page, making it easy to accomplish my goal of unobtrusiveness.

To illustrate this I included a drawing (Figure 3.2), in which the red area shows the tab bar, the blue the formatting bar, and the green the text area. Eventually I decided to go for a more conventional approach with toolbars mounted around the edges (Figure 3.3), although I did retain the vertical formatting bar from the original design. The tab bar moved to the bottom of the program, and I added a new toolbar (yellow in the illustration) to contain controls such as "open" and "save" as well as the title of the note.

Main Window			
Title bar	Main container		
	Formatting bar	Document container	
		Tab bar	Text box

Table 3.1: Diagram of program structure

Once I decided on my layout, I needed to work out the structure of the program (Table 3.1). The main window would contain all the other elements, and the main container would contain all except the title bar. In the actual program the main container was unnecessary, as Tkinter included elements which allowed me to align the formatting bar to the left. Most of the code for formatting the text would be inside the text box's code, and most of the rest would be in the main window, meaning that my code would be collected together rather than spread out, making it easier to read.

Before getting to the actual programming I needed to choose a VCS. What a version control system does is take snapshots of all the code whenever they are told to, and the programmer can backtrack to any snapshot at any point in the programming. These snapshots are called commits. They also allow for different "branches" of the code, so that if a feature is being worked on, it can be coded separately from the main code, ensuring that if it breaks the functionality of something else, it is not an irreparable change. The version control system I used was one called `git`, because it is the most popular VCS in use today, and thus help with its features is extremely easy to find. I used it in conjunction with a website called GitHub[6], a social coding site. The main advantage of using git and GitHub is that they allow me to have a backup of my code online, which makes it easy to switch between computers and means I have an up-to-date backup in case my computer breaks. The use of GitHub also allows other people to view my code and submit "pull requests", which ask me to download improvements and fixes they have made to my code, although I did not receive any while programming.

6. Preston-Werner, Wanstrath, and Hyett, "About GitHub."

3.2 Programming

I will only provide a brief summary of the actual coding, because most readers would be uninterested in the intricate specifics of the programming aspect of this project. If you would like more detail, my code, along with my commit messages, is available on GitHub[7]. I started with a basic layout of all the parts of the UI, except for the tabs. To explain why I did not work with the tabs from the start, I will need to explain how I intended for tabs to function.

When changing tabs, the content of the main text box would need to change, as of course the tabs would have different contents. There were two ways I could accomplish this – switching tabs could either swap the main text box for another one, or it could simply swap the contents of the main text box. Swapping the text box would be simpler in terms of the code required, but swapping the contents would make the program handle large numbers of tabs better. This is because swapping the text boxes would require storing the text box and its contents in the computer's RAM, and swapping the contents would require storing only the contents, meaning the program would use less RAM.

The method I chose was to swap the contents of the text box, as the program had to be as fast as possible, and if a program uses too much RAM on a machine without a lot of it, it will slow down. Using this method for the tab switching allowed me to make the program function without the tab switching before adding it in, although I did have to be careful not to make any decisions which would make adding the tab functionality difficult.

Making the GUI was as simple as combining Tkinter's `Frame`, `Button` and `Text` classes in the layout I wanted. A `Frame` is a container which can be coloured and can hold other objects. A `Button` is a clickable button, which can be assigned to run a function in the Python code. A `Text` is the most complex element I used – it has various functions within it for changing the fonts used inside it, changing the behaviour of the enter key, and many other functions.

The most difficult functionality to implement was the note linking functionality. I used a Tkinter function called `filedialog.askopenfilename()`, which opened a popup window where the user could select a note. I then made another window in which a user could select which line of the note to link to. The link was then inserted into the text box as the name of the linked file, or if some text was selected, the link would be applied to that. When clicked, the link opened the linked note in a new tab.

7. Bryant, "Luxtylo/NoteVelocity."

Chapter 4

Conclusions and Evaluation

Towards the end of the project, I sent NoteVelocity to several sixth-form students for some final testing and feedback. Some comments I received include:

- “It’s actually very fast.“
- “There’s no right-click menu.”
- “The text formatting updates quite slowly as you type.”
- “The buttons on the left make no sense.”
- “Keyboard shortcuts should use `Command` instead of `Ctrl` on Mac.”
- “The top level is not selectable with the mouse.”

I considered how I could fix the problems pointed out. The lack of a right-click menu is not easily fixed – there are solutions but they would require a lot of code to insert. A right-click menu usually includes options such as “copy” and “paste”. These can fortunately still be used with the keyboard shortcuts `Ctrl-C` and `Ctrl-V`, so the functionality still exists, but may not be accessible to the average user.

The slowness in updating the formatting is, unfortunately, due to a limitation of Tkinter. It could be worked around using Tkinter `mark`s in the text box, but this would break during tab switching. Tab switching functionality is more important than the speed of updating the formatting, so this problem will remain.

The buttons in the formatting bar are limited to a single character. I could replace them with an icon, but once the functionality is known, a picture illustrating making something a title (for example) is less clear than a “T” denoting “Title”.

Keyboard shortcuts can currently be modified by changing the code of the program. One way I would have fixed this would be to supply the program with a text file containing the keyboard shortcuts for the commands. If the keyboard shorcuts in the text file differed from the program's defaults, the different ones would be applied. This would allow the user to customise their keyboard shortcuts without having to download NoteVelocity's code.

The final problem was large enough that I chose to fix it. It was due to the order of the `tags` – the currently selected text was given a tag which marked it as selected. This was below the tag which made the title look like a title, meaning the `selection` tag was not shown. I fixed this by changing the order of the tags, so the `selection` tag was above the `title` tag.

Overall, NoteVelocity accomplishes most of the goals I set for it:

- On my personal laptop, the slowest computer available to me, the longest time NoteVelocity took to start was 0.47 seconds, just under my target of 0.5 seconds. On most computers it would be even faster than this. I included keyboard shortcuts for almost every feature, although some are unconventional (for example, Tkinter would not allow me to use the conventional `Ctrl-Tab` to switch tabs and I had to opt for `Ctrl-D`)
- Notes can be rewritten by pressing the rewrite button or using the keyboard shortcut `Ctrl-Shift-R`. This is only one click of the mouse, so it is as easy to access as I intended
- The linking functionality works but has a major problem, which I will explain below
- Most of the features are fairly simple, and all except the linking functionality have a button in the user interface. When the `Shift` key is held as the mouse moves over the `Save` button, it changes to `Save As`. Other similar things happen with the control key, and with the `Open` button, and this may be too complex or hidden for the average user. It would probably have been a better design decision to have a separate button for each function visible at all times or in a basic menu.
- As the user interface is around the edge of the program and has neutral tones, it is very non-distracting.

The program's main bug is in the linking functionality. It arises when switching tabs. Each link is stored as a `tag` applied to the text, which makes it appear blue, underlined, and clickable. Switching tabs makes all links disappear, as upon switching tabs all the `tags` in the text box are updated. This meant that I needed to find a way to fix it. I stored each link, the location it linked to and the characters in the text box which contained the link in a `list` as it was created, which should have allowed me to restore them after switching tabs. However, something prevented this from functioning, meaning that the links do not reappear upon switching back to the tab. After over a week of trying to fix this, I had to give up as I was coming to the end of my project.

The best way to fix this problem would be to change how tab-switching works to the other option I discussed – storing the whole tab instead of just its contents. With the high speed of NoteVelocity, storing all the tabs would not slow the program down noticeably. This would take a long time to implement and test to ensure it was working, but it would result in a more robust program. If I had known how fast the program would be, I would have chosen to use the easier option of storing the whole tab when I first started programming.

Over the duration of my project, my commit messages in git improved – from vague single-line comments to multi-line comments with a good summary of what was updated in the latest commit. This made it easier to see what needed to be worked on when I next resumed programming, and means that other people can much more easily see what was changed in each commit and how the development of the program progressed.

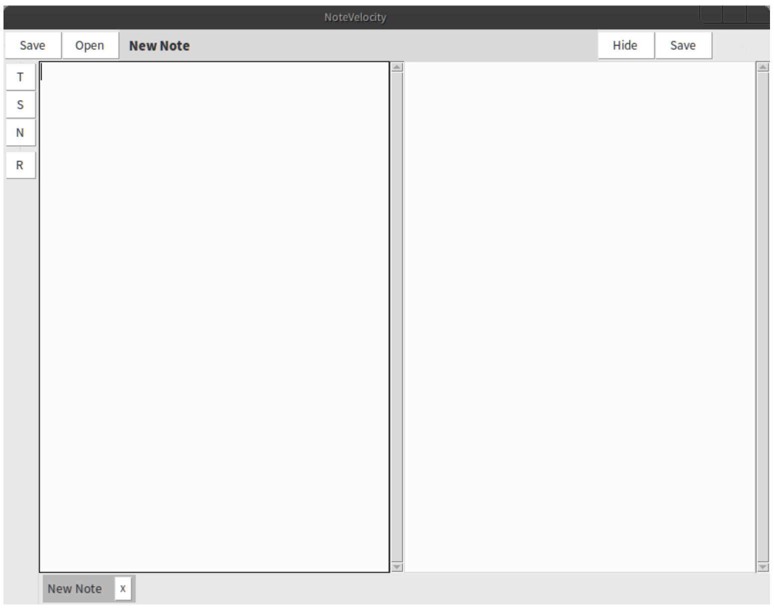


Figure 4.1: The finished version of NoteVelocity

A screenshot of my finished program is shown in figure 4.1. It will look different on different operating systems, as Tkinter changes to match the look of the operating system. The program is available for download from its page on GitHub[1].

1. Bryant, "Download NoteVelocity."

As a whole, NoteVelocity fulfils its goal of being a program "to aid sixth-form students in efficient note-taking" to a reasonable extent. The ability to rewrite notes allows summarising of the notes, which according to my research is an effective revision technique. It could be made more effective by adding functionality to allow the notes to easily be summarised multiple times – perhaps by allowing any two notes to be opened side-by-side at any time. The lack of linking functionality means that it is more difficult to link ideas – another important element of revision. Overall, I think I have accomplished the task set out in my title.

Bibliography

"25.1. tkinter — Python interface to Tcl/Tk — Python 3.3.3 documentation." Accessed November 3, 2013. `http://docs.python.org/3/library/tkinter.html?highlight=tkinter#module-tkinter`.

Brown, C. Marlin. "Comparison of Typing and Handwriting in "two-finger typists"," 32:381–385. 5. October 1, 1988. Accessed September 11, 2013. doi:`10.1177/154193128803200533`. `http://pro.sagepub.com/content/32/5/381`.

Bryant, George. "Download NoteVelocity." Accessed February 10, 2014. `https://github.com/Luxtylo/NoteVelocity/releases`.

———. "Luxtylo/NoteVelocity." Accessed February 8, 2014. `https://github.com/Luxtylo/NoteVelocity/`.

Bui, Dung, Joel Myerson, and Sandra Hale. "Note-taking with computers: Exploring alternative strategies for improved recall." *Journal of Educational Psychology* 105, no. 2 (2013): 299–309. ISSN: 1939-2176(Electronic);0022-0663(Print). doi:`10.1037/a0030367`.

Di Vesta, Francis, and Susan Gray. "Listening and note taking." *Journal of Educational Psychology* 63, no. 1 (1972): 8–14. ISSN: 1939-2176(Electronic);0022-0663(Print). doi:`10.1037/h0032243`.

"Effective note-taking." Accessed September 2, 2013. `http://www.reading.ac.uk/internal/studyadvice/Studyresources/Reading/sta-effective.aspx`.

Epstein, Zach. "iOS market share lost ground to Android in September despite iPhone 5S, 5C launches." Accessed January 9, 2014. `http://bgr.com/2013/10/01/ios-market-share-september-2013/`.

Jones, Eric, Travis Oliphant, and Pearu Peterson. "SciPy, Open source scientific tools for Python." 2001. `http://www.scipy.org/`.

Kiewra, Kenneth, Nelson DuBois, David Christian, and Anne McShane. "Providing study notes: Comparison of three types of notes for review." *Journal of Educational Psychology* 80, no. 4 (1988): 595–597. ISSN: 1939-2176(Electronic);0022-0663(Print). doi:`10.1037/0022-0663.80.4.595`.

Kiewra, Kenneth, Nelson DuBois, David Christian, Anne McShane, Michelle Meyerhoffer, and David Roskelley. "Note-taking functions and techniques." *Journal of Educational Psychology* 83, no. 2 (1991): 240–245. ISSN: 1939-2176(Electronic);0022-0663(Print). doi:`10.1037/0022-0663.83.2.240`.

Lundh, Fredrik. "An Introduction to Tkinter." Accessed November 3, 2013. `http://effbot.org/tkinterbook/tkinter-index.htm`.

Preston-Werner, Tom, Chris Wanstrath, and PJ Hyett. "About GitHub." Accessed February 8, 2014. `https://github.com/about`.

Protalinski, Emil. "Windows 8 rockets to 7.41% market share as Windows XP falls below 35% mark." Accessed January 9, 2014. `http://thenextweb.com/insider/2013/09/01/windows-8-rockets-to-7-41-market-share-as-windows-xp-falls-below-35-mark/#!u0cfB`.

Purdie, Nola, and John Hattie. "Assessing students' conceptions of learning." The University of Newcastle, Australia, *Australian Journal of Educational & Developmental Psychology* 2, no. 1 (2002): 17–32. Accessed August 19, 2013. `http://www.newcastle.edu.au/journal/ajedp/previous-issues/volume-2.html`.

Rossum, Guido van, Barry Warsaw, and Nick Coghlan. "PEP-8 – Style Guide for Python Code." Accessed October 13, 2013. `http://www.python.org/dev/peps/pep-0008/`.

Shale, Suzanne, and Keith Trigwell. "Paper 3: Students' conceptions of learning" (2002).

Appendices

Appendix A

Questionnaire

Answer boxes have been removed from questionnaire

1. How do you take notes during lessons?
    - Writing down what the teacher says
    - Summarising the teacher's words
    - Charting (using tables)
    - Mapping (eg mind maps)
    - I don't
2. How do you take/refine notes after lessons?
    - Writing down what the teacher said
    - Summarising the teacher's words
    - Charting (using tables)
    - Mapping (eg mind maps)
    - I don't
3. How important do you think the speed of a note-taking program is? On a scale of 1-4, with 4 being the most important
4. How important do you think the ease of use of a note-taking program is? On a scale of 1-4, with 4 being the most important
5. Simplicity vs features On a scale of 1-4, with 1 being the simplest, and 4 being the most advanced features (eg adding equations, etc.)
6. Would you value the option to have text expansion? (Eg turning "myaddress" into your full address when you type it) Yes/No
7. Would you want to be able to store notes on any of the cloud services below? Select multiple or none if necessary

- Evernote
- Google Drive
- Dropbox
- Skydrive

8. Would you value custom "profiles" for different subjects? Each profile would change settings to be optimal for that subject – eg maths would make equations easier to type Yes/No

9. Do you use keyboard shortcuts when taking notes? Yes/No

10. Do you have any comments or suggestions?

Appendix B

Specification

Specifying features and design decisions which I need to use in my program. Important aspects of the program:

- Speed
- Ease of use
- Easy rewriting and summary of notes
- Linking notes
- Unobtrusive user interface

Quantitative goals

- Program must start, and be able to create notes, within 0.5 seconds on a computer with processing power roughly equivalent to my school's computers. If the program cannot meet these targets it should have cues that make the user percieve it as being fast. There should be keyboard shortcuts for most actions to allow knowledgeable users to work faster
- Notes must be able to be rewritten while looking at the original note. The rewriting functionality should take no more than 2 clicks of a mouse to access
- Notes must be able to be linked together, whether by hyperlinks in the text or buttons in the UI

Qualitative goals

- The program must be easy to use for the average sixth form student, including the more complex features
- The user interface should be unobtrusive and not distract the user from their work

Appendix C

Glossary

bug

In a program, a bug is a point where the program is not functioning as intended.

class

In computer programming, a class is a template for a type of object plural.

commit

A snapshot of the contents of a project, made by a version control system.

commit message

A message accompanying a commit in a version control system. Should explain what changes have been made since the previous commit.

formatting bar

In NoteVelocity, the formatting bar is the part of the program which holds the buttons for formatting the text.

function

A unit in a computer program which contains a set of instructions to do a specific task.

GUI

A Graphical User Interface (GUI) is a way for humans to interact with computers, which uses visual windows, icons and menus to display information.

HTML and CSS

Markup languages used on websites to define the content and how the content is shown respectively.

instance

An object made from a class template.

keyboard shortcuts

Combinations of keypresses which can be used to complete actions in a program as an alternative to using a mouse.

module

In Python, a module is a file containing Python definitions and statements - a section of a program.

operating system

The low-level software which supports a computer's basic functions, such as scheduling tasks and controlling peripherals, for example Microsoft Windows or Apple OS X.

popup window

A window which appears in front of the window currently being viewed.

pseudocode

An informal description of the operating principle of a computer program.

RAM

Random Access Memory (RAM) is volatile storage which computers use to temporarily store data related to programs while they are running.

statically typed

A programming language which is statically typed has variables which, once declared, are limited to hold a specific type of data.

syntax

The structure of statements in a computer programming language.

tab

A tab is a visual element of a computer program, which indicates that a further document or page can be opened.

tab bar

In NoteVelocity, the tab bar is the part of the program which holds the tabs for switching between different notes.

terminal

A text-based interface to a computer, in which commands can be input and output received.

text formatting

The style (font, size, etc) of text.

title bar

In NoteVelocity, the title bar is the part of the program which holds the file operations buttons as well as the title of the currently opened note.

VCS

A Version Control System (VCS) is a program which can take "snapshots" of an area of the user's hard drive, and view and move between these snapshots, effectively travelling in time as a project was worked upon first.

whitespace

Any character or series of characters which represent horizontal or vertical space in typography. When rendered, a whitespace character does not correspond to a visible mark, but typically does occupy an area on a page.

window

A framed area on a computer screen for displaying information.