Project Appraisal

# Allow users to see what drives are connected

The software allows the user to see what drives are connected, along with if the drive has a backup record. The method it does this isn’t the most efficient in terms of performance, but this is only likely to make any kind of difference on very low-end devices. Ideally it would be more efficient, because currently it just scans every 100ms for new drives. Hopefully with a little more research, an easier method could have been used.

# Allow users to add drives for backup

Users have the ability to add backup record through an easy to use interface. The interface is clear and can has tooltips to help with what the inputs do. If I were to do it again, I would write a separate wizard, so there can be more help information, and a much cleaner, and less cluttered interface.

# Allow the user to backup content from the drive to their computer

The user can back up the USB drives efficiently, at a speed that would be expected for the interface. Unfortunately this is using a separate library, and so I have very little control over what is happening with it, and so I had to create my own, rather badly implemented handlers for certain events, because the DLL didn’t have the functionality for this.

# Users can see per-file and overall progress of backup process

Thanks to the copying library, there are 2 progress bars shown on the copying window. One is per file, the other is entire progress. It is very accurate, being updated by a call back from the library. This implementation works perfectly, and I don’t think it could work any better

# Allow users to manage backups on their computer

Users can manage all backups using the management window. It allows editing, deleting, and adding backups. The window isn’t the best implemented, but it’s functional. Ideally I would like it to look much nicer, and contain more information. Also I would like to format the controls better, because the display of information doesn’t make the

# Allow users to restore backups onto another drive

During the development process, there was not enough time to implement this. I tried to concentrate on the more important sections of the backup program. The client thought that the restore feature would be useful, but not 100% necessary, especially considering it could be done manually if need be. This feature would be easy enough to implement, however may need editing of already implemented functions.

# Allow different users to have different backup rules on the same computer, for the same drive

The software had the ability to do this, however the entire database would need to be regenerated, and some of the database functions rewritten, because at the moment the database doesn’t store the username. An easier, but more complicated way to implement this would be to store the database in the user’s home directory, but again, I ran out of time to complete this.

# Allow users to delete backup records

Through the drive management window, users can delete backup records, very easily and simply. One problem with this is again the user interface for the window. It could be much simpler, and display the information clearer. But all in all the implementation works rather well.

# Use file compression to allow users to roll back through backups

Originally, when I started the project, I was going to offer 3 options: No compression, zip file compression, and LZMA compression. After some debating, I removed the zip file compression, because it was much easier to implement and store in the database, and still comply with normalisation rules.

# Allow backups to run at the same time to different drives

Thanks to how the library that handles copying works, multiple backups can run perfectly at the same time, obviously with bandwidth limitations. The implementation of this in the software is a little bodged, meaning that certain controls have to be cleared manually. Ideally I would have made the implementation smoother, and cleaner, at the moment it can be a little confusing to edit.

# Use a hash to calculate if the drive has changed since last backup

I was able to find online a function that hashed a directory, and edited it to hash an entire drive. It worked very well, and also rather quickly. However came into problems with large drives. During the testing I was doing when writing the software, and implementing the function, I was only using 1GB USB drives, on my main computer, which it handled with no problems. When it came to testing it in real world environments,, such as more standard machines, and around 15GB of data, the program crashed. After research I found that this was because the algorithm used tried to store the entire contents of the software in system memory (RAM). Most machines have 4-6GB, and the machine I was real world testing on had 3. This error would take a very long time to fix, because of how heavily implemented the function is to the system. Whilst I know how to fix it in theory, implementing this would be a large task, and to make it efficient enough to work on low-end hardware would take a lot of time.

# The system should be self-documenting, and need very little explanation

Thanks to a very large amount of tool tips, and extra documentation, as well as a simple layout and obvious buttons. The software is very easy to navigate through. The main part that threw peers was the concept of adding a backup record before a drive could be backed up, or the security implementations that automatic backup could introduce. Once I explained the reasons for an automatic backup, because otherwise the computer wouldn’t know what to do with drives, the worries they had were answered.

# The system must be able to run on windows 7, 8.1, and XP (both 32 and 64 bit)

Thanks to the nature of the C# framework and runtime in windows, the system is fully functional on all windows platforms, provided that they have the correct framework installed, which is also bundled with the installer. Fortunately, my compiler (Visual Studio) contains the option to specify whether to create code for 64-bit or 32-bit computers. In order to make release and production much easier, I selected the ‘Any CPU’ option, which meant it doesn’t matter. This could have a performance impact on 64-bit machines, but I don’t know without testing it.

# The user must be able to change the theme to match the users OS or preferences

Through testing with DotNetBar, I found that styles were managed per window using a style manager object. I also found that the parameters were accessible and editable during runtime. However there appears to be a bug with this, because when assigning the property, instead of copying it from window 1 to window 2, it did it in reverse, creating the wrong style. Therefore I stopped developing this feature, especially as it was one not in the client spec, just one that I thought would be a nice addition

# The system must use no more than 15kb system memory and 1% CPU when minimized to tray

The numbers here were just pulled from thin air, so could have been a tall order to work to. Fortunately I was making sure that my code was relatively efficient throughout. So it came way under this estimate. If I were to go through the code again, I could probably make it even more lightweight, but I think it would be a lot of work for a small outcome, especially considering how much system memory and CPU power the average computer has.

# The system must use no more than 5% CPU and 30kb system memory when in GUI (No backup running)

# System must be protected so that no more than 1 backup is stored on the system per drive

Thanks to the way that the database is implemented, it will throw errors when someone tried to add 2 backup records for the same drive. So this problem is easily sorted. If I could do the project again, I would have caught this problem before it got to the database, to present the user with a better error message than the one that comes from the database, but at least it stops them.