Joon Ki Hong

CS 2340

Individual Case Study

**Audacity Analysis**

Dominic Mazzoni and Roger Dannenberg at Carnegie Mellon University developed Audacity to use as a platform for developing and debugging audio processing algorithms. When the software became open-source, it garnered a large amount of attention by other developers. The developers and growing community of Audacity was set to solve various high level as well as low-level problems. Audacity was developed to be an audio editing and developing program that is easy to use, yet very functional and free. One of the ways they approached this goal was to make it “discoverable”. Discoverability meant that users should be able to understand and use the software right away without the need to struggle to use simple functionalities. The discover part of it comes in when the user gradually learns the deeper and the richer functions of the program as they spend more time with it. They strived to uphold this principle to give the software greater interface consistency.

This goal did come with its consequences however. To maintain its open-source standards, the architects of Audacity had to be cautious of the constraints of developing open source software. These constraints included licensing restrictions, architectural inconsistencies in code over time, limited resources, constraints regarding how much abstraction is wanted in reused code, and time.

Its intended user base or demographics has been pointed towards those needing to edit or develop audio at any level. Their desire to maintain the concept of discoverability makes clear that they want people from all levels of experiences to use the software without lacking functionality.

They support a modular design that allows more experienced and demanding users to take advantage of the software. It also allows them to keep the core components of Audacity very stable and also allows them to experiment with additional functionalities without breaking its core. To achieve this modular design, the developers used various libraries layered on top of each other to create a level abstraction that other developers can take advantage of. This level of abstraction however, is not complete and perfect. Due to the constraints of limited resources and open source standards, Audacity had to somewhat deviate from the most efficient of approaches. For example, because Audacity relies on other open source third party APIs, they do not have the flexibility to choose which parts of their application they want to have abstracted. This leads to a lot of ugly code because it requires them to stay within the constraints of the API and develop uglier solutions to bypass such constraints. An example of this comes from their implementation of playback and recording, which is done by three threads that are each handled in very different ways. This implementation was due to the fact that the API’s they have implemented force them to use pre abstracted layers of code, which ultimately does not allow them to process these threads at a lower and more efficient level.

Given these constraints (open source, resources, time, licensing), it would be hard to suggest a better architectural design for Audacity. However, they seemed to have some of their low level code base intruding into their higher level GUI code base. This can be observed in their Track Panel implementation. Their Track Panel elements are essentially GUI elements that are customized for Audacity applications and from this, we can observe the fact that some of their GUI -specific code also controls their application, lower-level, code that drives their program. In an ideal software design, the GUI-specific code should not have to know or handle the application specific code. Taking this into consideration, I would use the Model-View-Control architecture to clean up any holes of their abstraction layers. This would keep the GUI- specific code only within the GUI-specific and the control code within itself. Having holes within abstraction layers would allow for extremely ugly code and code that is not applicable to other applications other than those for Audacity.

The developers of Audacity depend on third party libraries that are each specific to the function they desire. This follows the packing paradigm of grouping classes together. The software is divided into layers of libraries such as the Port Audio library, the wxWidgets library, and other supporting libraries that allow for plugins. The Port Audio library, for example, is responsible for dealing with playback and recording functionality of the application. The wxWidgets library is responsible for dealing with all GUI related controls.

With that known, we can safely say that Audacity implements a reuse paradigm because it largely depends on these third party libraries for its core functionalities. At its core, Audacity is a collection of components woven together by thin layers of abstraction. However, we may see some spots of maintenance here and there in the code. As mentioned before, Audacity is at the same time restricted by these components. The thin layers of abstraction that weaves these components together is subject to maintenance. One such example is the Track Panel implementation that goes through abstraction layers and prevents it from being reused.

The software, from a SOLID design point of view, is well built, considering the above-mentioned constraints. It withholds the single responsibility principle through the fact that, as mentioned previously, that the software is built as a collection of components, all with a specific role to the application. For example, the wxWidgets library is responsible only for GUI related applications, whereas the Port Audio library is responsible only for playback and recording.

The software also withholds the open/closed principle by having a modular design, open for plugins. Although this implementation has some disadvantages such as restrictions regarding the Fourier transform code, having slower code, or having larger builds, the software allows for external and additional functionalities to be implemented without altering any of its core functionalities.

The software does not withhold the Liskov substation principle. Specifically, the Track Panel implementation is to a certain extent a derivation of wxWidgets. As far as wxWidgets is concerned, there is only one widget for the entire Track Panel. The Track Panel has been extremely customized to the applications of Audacity. For the UI functionalities that the developers wanted, this was the only way to avoid performance and graphical issues. Because of this, their Track Panel in no way, could replace the wxWidgets model without redefining functionalities.

The software withholds interface segregation appropriately in that the modular design of the software breaks up the responsibilities of the application rather than having one interface that defines the entire program, all completely interweaved together with dependencies.

The software also maintains abstraction layers to prevent higher level implementations from reaching lower level code. There are exceptions to this as explained above. Audacity’s Track Panel implementation does violate and cross over an abstraction layer meaning that the Track Panel knows about the functional aspects of the program when it should only know the graphical aspects. In that regard, it breaks the dependency inversion principle.

As mentioned before, and by James Crook, the author of the Audacity portion of *The Architecture of Open Source Applications*, the main design issue that exists with Audacity is with the Track Panel. As suggested by James Crook, I would also suggest the flyweight design pattern to overcome the disadvantages of using wxWidgets to power the Track Panel. The problems with fully implementing wxWidgets is that populating the screen with too many widgets would decrease performance, lead to graphical issues (flickering), and would take up too much unneeded memory. Using the flyweight design pattern, they could create similar structures to wxWidgets without the extra bloat that would hinder Audacity’s performance.

Regarding quality factors (FURPS), Audacity addresses functionality in that it takes into consideration potential security loop holes. The developers preemptively disabled all TCP/IP connections in and out of the application to prevent security risks. Audacity also addresses usability in that it applies the “discoverability” concept to keep the UI consistent, easy to use, and aesthetically pleasing. It also addresses reliability and supportability in that its modular design allows for a stable core, which prevents any additional code from altering its core functionalities. It also allows for safe experimentation of additional plugins without breaking any other important lines of code.