Final Exam

Tate Larsen

2013-04-15 Mon

Problem A

For a year or so now I have been using Miro[†] to organize my music and video files and have noticed a distinct lack of a feature I would very much like to use, that feature being a visualizer. Additionally, I think it would be kind of neat to have the ability to alter the playing sound file by doing things such as changing the playing speed and moving elements of the song from one frequency band to another, although not necessarily in real time. Now, Miro has been around since early 2006 (although it was called Democracy Player then), was written primarily in Python, and incorporates video playing, music playing, a BitTorrent client, an RSS feed reader, and more.

Due to the way that Miro handles sound and video output, each feature I want to add is both possible and ought to be relatively easy to implement. Additionally, there will be significant overlap in their implementations without causing conflict. As to how they will be implemented, Miro essentially takes a given file and outputs it on a `Renderer` object that can be either an `AudioRenderer`, `VideoRenderer`, or `FakeRenderer`. Both features can be implemented by essentially listening to the `Renderer` being used for a given file and either analyzing or modifying what they hear.

Now, from the user side of things, everything is fairly simple. Lets say they wish to use either feature on a particular audio file–they start by specifying both the desired `Effect` and which `Analyzer` to use (in the case of the modifying feature they must also specify an output file to use), then they hit play and watch the visualizer or wait and then listen to the modified version of the audio file.

On the backend, there is quite a bit more going on. There are four classes necessary to the features, two of which are abstract and therefore need additional implementations. These four classes are `EffectApplier`, `Effect`,

[†] http://getmiro.com/

`Analyzer`, and `AnalyzedBuffer` with `Effect` and `Analyzer` being the abstract two. Starting from the top, there is the `EffectApplier` class. This class is essentially just a listener and controller and doesn't actually modify anything itself. It holds a reference to an `Effect` and basically enters `listen_loop()` that, when started, continuously calls `check()` to see if the active `Renderer` is currently playing. If it is playing it fills a byte array buffer (henceforth `byte[]`) with the sound data it is rendering and passes it on to the `Effect`. An `Effect` is an abstract class and is the real workhorse of both features. It holds an `Analyzer` and has a `process(byte[])` function that is implemented by any child classes. In most cases, the `process(byte[])` function passes the `byte[]` buffer to the `Analyzer` and gets a bunch of information back in the form of an `AnalyzedBuffer` that it then uses to perform whatever its primary task is. In the case of the visualization, this class is the `BasicVisualEffect`, which essentially renders colors, lines, and other visual effects to the screen using the PyGLet library based on the received `AnalyzedBuffer`. With the modifier feature, the `AnalyzedBuffer` is modified and then converted back to a `byte[]` buffer and written to the output file by the `BasicAudioEffect` class. An `Analyzer` is an abstract class that has two primary functions: `analyze(byte[])` and `rebuild(AnalyzedBuffer)` which do exactly what their names suggest and transform a `byte[]` buffer to an `AnalyzedBuffer` and transform an `AnalyzedBuffer` back into a `byte[]` buffer. How the `Analyzer` does this can vary, but the most common method in similar applications is to perform a Fourier Transformation on the buffered data and the inverse of that same transformation. We will create the `BasicAnalyzer`, which performs Fourier Transformations and their inverse. An `AnalyzedBuffer` is a simple container that stores the data from each frequency band in a two dimensional array, the width of each band in another array, and the duration of each cell of the data array. Figure 1 consists of a class diagram describing the classes and their interactions, while Figure 2 is a sequence diagram for the visualization feature.

By implementing the features this way, it allows for relatively easy modification of the existing classes as they are all loosely coupled and fully encapsulated. Additionally, the implementation of other new features is easy and requires no modification of the existing code. For example if someone wanted to add the ability to have Miro control a lighting controller (for example, to have lights flash to the beat of a song or something similar), all they would need to do is write a new class that inherits from `Effect` and communicates the necessary control data to the lighting controller instead of outputting to the screen or an audio file.

In addition, due to the way that both `Effect` and `Analyzer` are imple-

mented, the program as a whole obeys the Open-Closed Principle. The Open-Closed Principle states that the behavior of a program should be changeable without requiring modification of the source code. The most common way to ensure a program obeys this principle is to make good use of abstract base classes or interfaces as the language allows. By doing this, one can swap one child class for another and thereby change the functionality of the program by implementing a new child class rather than rewriting the code. Because they are both abstract classes that have their functionality implemented by their children, neither `Effect` nor `Analyzer` need ever be changed to change functionality. You would merely create another class that inherits from them.[1]

On a related note, the Liskov Substitution Principle states that if class X is a child class of class Y, than any instance of class Y can be replaced with an instance of class X. As a result of the way that the `process(byte[])`, `analyze(byte[])`, and `rebuild(AnalyzedBuffer)` are implemented, the program obeys the Liskov Substitution Principle as any `Effect` can be replaced with any other `Effect` without breaking functionality and likewise with `Analyzers`. This would allow for virtually any functionality based on listening to the `Renderer` to be implemented without requiring modifications to the existing code.[2]

Expanding from both of those principles, to add a new feature, none of the existing code ought to require change. You would just need to implement a new child class of `Effect` or `Analyzer` and all you would be required to add are those child classes (they could additionally require new classes for their functionality, but those would not be added as a dependency for the existing features). In the unlikely event that change **is** required in either the `Effect` or `Analyzer` classes, the changes can be implemented in those two base classes and thus no child classes need any modification because the changes would not require re-implementation as they would be inherited by the child classes.[3]

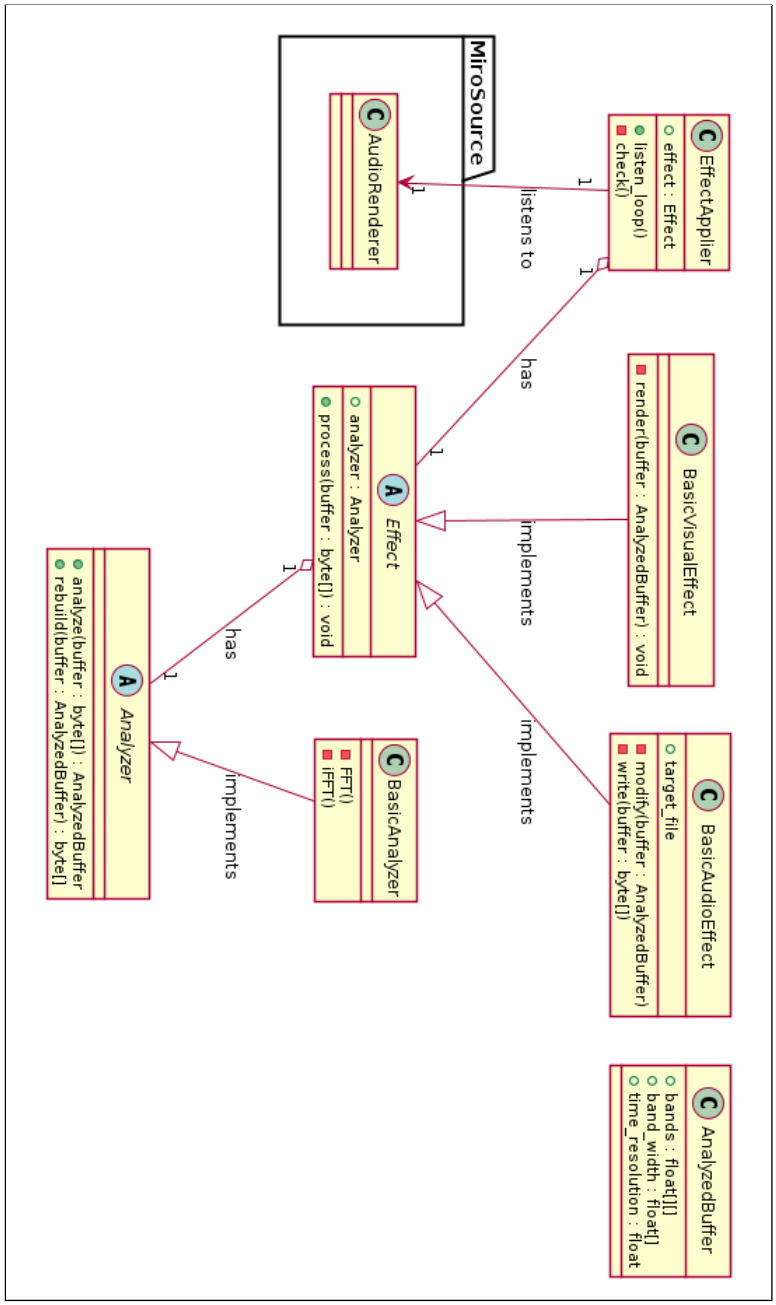

Figure 1: Class Diagram

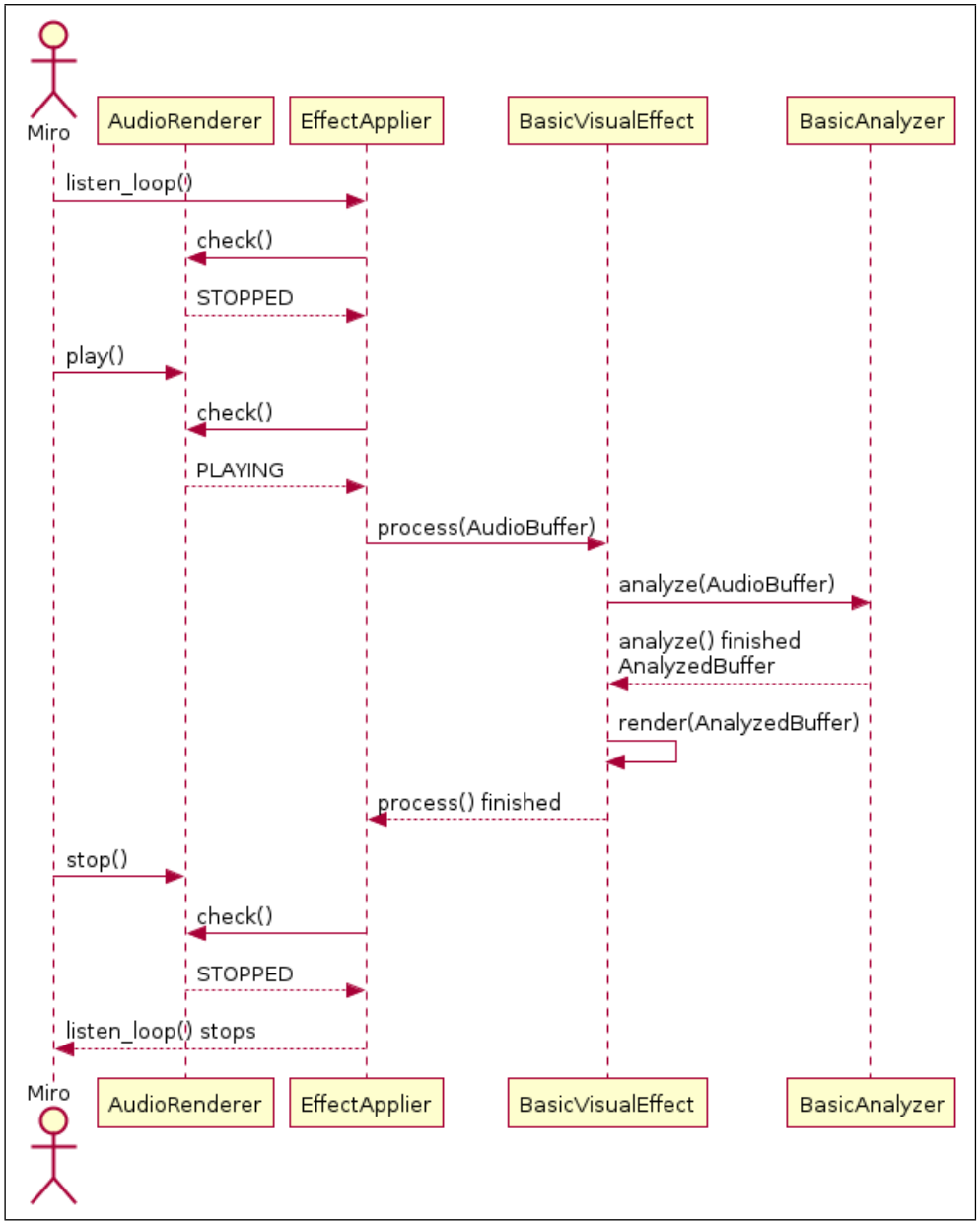

Figure 2: Sequence Diagram

Problem B

My preferred Agile methodology for implementing the features outlined above is Scrum. Scrum is a development framework consisting of a number of parts: the Sprint Planning Meeting, the Sprint, the Daily Scrum, the Sprint Review, the Sprint Retrospective, the Product Backlog, the Sprint Backlog, and the Increment. The Sprint Planning meeting is where the Scrum Team decides which items from the Product Backlog will be worked on in the next Sprint and adds them to the Sprint Backlog as well as how they will make sure that those items get completed. The items selected are done so with the intention of wholly completing them during the upcoming Sprint. The second piece of Scrum is the Sprint which is a consistent length of a month or less in which one development iteration or Increment occurs. The Sprint is a period in which the team works on items from the Sprint Backlog. If they fail to complete the items during the Sprint, the incomplete portion of the item is put back in the Product Backlog for completion during another Sprint by either the same team or another. The Daily Scrum is a daily meeting wherein each team member outlines three things: What they've done since the last Daily Scrum, What they will do before the next Daily Scrum, and What obstacles might they encounter in doing so. The Daily Scrum allows the team to gauge the progress of the project and also to get an idea of what other team members are up to and how they are doing. In the Sprint Review the Scrum Team inspects the Increment and discuss what was completed and what has yet to be completed as well as any changes to the Product Backlog. The Sprint Retrospective is a meeting during which the Scrum Team works to identify any issues with what happened during the Sprint and generally try to improve the way the Team functions as a whole. The Product Backlog is a list of all of the known parts and tasks necessary for the project and is ever changing while the Sprint Backlog consists of the items selected from the Product Backlog for a given Sprint and the Team's plan for the Sprint. The Increment is the collection of all of the work done so far on the product.

Additionally a Scrum Team consists of the Product Owner, the Development Team, and the Scrum Master. The Product Owner communicates with any stakeholders and customers and ensures that the product is what they want or need. They also are the ones who maintain the Product Backlog and order the items on it. The Development Team is a small group (generally 3-9 people) that works to actually create the product during the Sprints. The Development Team is fairly autonomous and members hold no rank. The job of the Scrum Master is to make sure that all Team members both

understand and follow the Scrum framework and basically helps everyone to work in the most efficient and easy way possible.[4]

The following is a Product Backlog and rough schedule of Sprints for implementing the features discussed in the previous section.

Product Backlog

- Create the `Effect` abstract class
- Create the `AnalyzedBuffer` class
- Create the `Analyzer` abstract class
- Create the `EffectApplier` class
- Connect the `EffectApplier` class to a `Renderer` from Miro
- Create the `BasicAnalyzer` class which extends `Analyzer` and performs Fourier Transformations on a byte array buffer and inverse Fourier Transformations on an `AnalyzedBuffer`
- Create the `BasicVisualEffect` class which extends `Effect` and uses PyGLet to render effects to the screen
- Create the `BasicAudioEffect` class which extends `Effect` and modifies an `AnalyzedBuffer` and writes it out to a specified file
- Integrate functionality into Miro's GUI

Schedule

Sprint 1

- Create the `AnalyzedBuffer` class
- Create the `Effect` abstract class
- Create the `Analyzer` abstract class

Sprint 2

- Create the `BasicAnalyzer` class which extends `Analyzer` and performs Fourier Transformations on a byte array buffer and inverse Fourier Transformations on an `AnalyzedBuffer`
- Create the `BasicVisualEffect` class which extends `Effect` and uses PyGLet to render effects to the screen

Sprint 3

- Create the `EffectApplier` class
- Create the `BasicAudioEffect` class which extends `Effect` and modifies an `AnalyzedBuffer` and writes it out to a specified file

Sprint 4

- Connect the `EffectApplier` class to a `Renderer` from Miro
- Integrate functionality into Miro's GUI

Problem C

The second chapter of *The Mythical Man-Month* by Frederick Brooks outlines five misconceptions that cause issues in software development, but all five of those can be easily mitigated by the Scrum development framework described in the previous section.

The first misconception is that programmers are optimists and will therefore underestimate how long a task will take because almost inevitably something will take longer than expected. Scrum combats this in the way that features are organized in the Product Backlog and in the Sprint Backlog. Tasks are designed to be completed within a single Sprint, but if they aren't, the part that was not completed is added back into the Product Backlog and just goes back into the cycle without causing significant delay in production. Additionally, the Daily Scrum and Sprint Planning Meeting serve to allow other Team members to provide input on issues causing delay and also to assess how a Team member is proceeding with their tasks.

The second misconception is that throwing more people at a problem will solve it. This does not work due to the non-parallel way in which most software must be developed. Scrum dismisses this by having the Product Backlog and Sprint Backlog. Due to their mostly autonomous structure, when members of the Development Team finish one assignment, they self-assign their next, allowing for much greater parallelization of work. Additionally, because the Scrum Team cannot be changed mid-Sprint and because management is forbidden from interfering with the Development Team, there is really no way that the misguided can ever throw more people at the problem in any way that would delay production.

The third misconception is that managers will not defend the estimates that the team makes and will, if placed under enough pressure, crack and

force the team to rush and release a shoddy product. Scrum combats this misconception by the presence of the Scrum Master. The Scrum Master's job is to make absolute certain that everyone on the Scrum Team is working as efficiently as they can by removing any obstacles and helping them organize. As such, they are required to maintain the credibility of the Team's estimates and therefore have a vested interest in not giving in to demands to rush. Likewise the semi-autonomous nature of the Development Team makes it very difficult to force them to rush.

The fourth misconception is that the schedule is not maintained and if the team falls behind there is no way of recovering save by adding more members, extending the schedule, or cutting features. Scrum addresses this in two ways: the Product Owner and the Sprint. The Product Owner has a vested interest in the product and is also the one who maintains the Product Backlog. They are essentially the ones who maintain the items to be scheduled so that the schedule is maintained. The Sprint allows tasks to be broken up into parts that can be completed within the scope of the Sprint. If something fails to get completed, it just gets pushed back onto the Product Backlog without causing a production delay.

The fifth and final misconception listed in *The Mystical Man-Month* is that when the schedule slips, the only solution is to throw people at the problem, which just makes the problem worse. Scrum is great at this, even if the schedule were to slip, which it could not easily significantly do, it does not allow the possibility of throwing people at the problem because of the way Scrum Teams are set up. Additionally, any slippage is just thrown back into the Product Backlog and can be picked up in another Sprint by either the same or a different Team and a smaller goal can be set for the next Sprint without causing any issues.[5]

Sources

1. *The Open-Closed Principle* http://www.objectmentor.com/resources/articles/ocp.pdf
2. *The Liskov Substitution Principle* http://www.objectmentor.com/resources/articles/lsp.pdf
3. *Head First OOA&D Slides from Chapter 5a* http://headfirstlabs.com/books/hfooad
4. *The Scrum Guide* http://www.scrum.org/Scrum-Guides
5. *The Mythical Man-Month - Chapter 2* http://black.goucher.edu/~kelliher/f2010/cs245/theMythicalManMonth.pdf

Certification

I, Tate Larsen, certify that the above is my own original work

---