Scotty Shaw (sks6)
VOOGASalad – Los Torres
Computer Science 308

Project Journal

Much of my time and effort involving this project came during the beginning, starting with Day 1 on November 3, and the end. Due to my commitments with HackDuke, I had to scale back just before Thanksgiving break, but I was able to attend most team meetings even if I was unable to be heavily involved in some decisions and coding during those two weeks. In all, I spent at least 100 hours alone. With the team meetings and other non-individual responsibilities, that number could be above 200 hours.

Most of my time in the beginning focused on helping discuss and determine our design. For example, most of the team originally favored a simple path-based progression for the baddies, but I pushed strongly for the ability to support an open arena, which would allow AI. This led to our game authoring environment's design and the inclusion of A* search so that our baddies can pursue simple paths or "smarter" strategies. I also helped push for composition so we could flatten out our hierarchy and create a more flexible implantation of behaviors.

I was originally assigned to the game engine, but my commitments as a HackDuke organizer forced me to reassign to building utilities for our team to use. My first project was a multithreading chatroom that would allow team communication during a co-op game, but that did not become a feature because we opted for a simpler, single-thread chatroom built on Heroku. Afterwards, I built the drag-and-drop panes for audio and video files, allowing the user to use various files, such as .mp3, .mp4, .m4a, .gif, and more. Then I built the AudioPlayer. This would allow the user to immediately hear the audio file being dropped into the pane, and I felt inspired to build a VideoPlayer. This easily took me at least 40 hours because I had never learned anything on the frontend. It took me a lot of time and help from others more experienced than me to understand the JavaFX documentation and create the VideoPlayer. This was certainly my hardest task all semester, but now having my own VideoPlayer that largely follows the UI/UX design of YouTube is a wonderful achievement I feel very proudly about.

This team was also very enjoyable to work with. Although we are all strong talkers and have big egos, we were all able to set aside our personal agendas to work together. Duke, Timmy, and Chase formed a powerful engine team, with Duke focusing more on the backend and the other two building the hierarchies with composition. Greg, Brian, and Allan put together an amazing game player, and the various features and games we could demo are testaments to their hard work. A single-player AI game and a co-op path-based game are very different, and they both worked perfectly. Brian also implemented the earthquake attack using Leap Motion, and that added an element to our gameplay experience. Austin and David worked to put together our authoring environment, and the fact that we could build a game live during our demo and play it immediately is proof that they put in lots of work. David's work with game data is also great, and it allowed us to load and save games

as needed. His photoshop skills also put together a large library of images for us to use. In total, I estimate that we easily surpassed 1000 total hours of work.

The communication in our team was mostly very good. Although we had some issues regarding some team members being less active or more stubborn when discussing something, we stayed on the same page about everything and could achieve a great deal. Although we mostly held to our original assigned roles, we were not hesitant to shift around and help with other teams when needed. Communication issues never arose, and we all truly enjoyed working and being with each other. Many of us feel that this team was our most enjoyable team.

Because of our team chemistry, we were able to complete the project as planned without deviating much. Other than my shift from engine to utilities, everything happened as expected, and even my change in role did not affect our team adversely. In fact, the single-thread chatroom became a possibility because I had built the multithread chatroom, and my expansion of the drag-and-drop panes and creation of the VideoPlayer helped push our boundaries of possibilities. We just simply ran out of time to implement all of our features because of our commitment to excellent design principles.

In total, I committed code over 120 times. 80+ of them ended up on the master branch, and I averaged around 50 lines for each commit. Of course, some of them were very small, such as deleting an unnecessary comment, while others were larger, such as building entire drag-and-drop panes in one commit. I feel the commits accurately represent my contribution to the project because I worked mostly on my own to create utilities for my team. Some of my commits also went to the utilities repo accessible to the entire class.

My first significant commit "Added per-thread class and read-write loop on the server side" comes from December 4. This was part of my first utilities project and marked my return from my HackDuke commitments to our VOOGASalad project. Despite my confusion trying to re-insert into the team, I was able to avoid causing any merge conflicts or bugs for my teammates. This commit introduced multithreading to the chatroom and led to us discussing whether we would adopt this model or the simpler single-thread chatroom built on Heroku. In the end, my chatroom fell out of favor, but we realized we could incorporate chat into our networking capabilities, and this allowed Timmy and Chase to demo our co-op game more impressively.

"Creating DragAndDropPanes for audio files; commenting and slight refactoring" was my second significant commit. It comes from December 6 and was my first step towards creating my VideoPlayer, although that was not an idea yet. More directly, this commit added a new feature that would allow us to add audio files for sound effects and background music. The significance of this commit is that it marked the start of a 3-day, 70+ commit coding tear, during which I also built a drag-and-drop pane for video files, then went on to build a simple AudioPlayer, which then inspired me to build a VideoPlayer. No conflicts came from this commit, and during my sprint, I was careful to avoid any potential conflicts that would hinder our team progress.

My last significant commit could not have happened in the wee hours of Sunday morning without Allan's help. I "Refactored and added lambda expression"

for the first time and finally understood how they work. No merge conflicts came from such a simple commit, but my masterpiece code is proof of how much Allan helped me. Without his help on this small commit, my following 25 commits, all of which involved refactoring frontend for the first time, could not have produced a masterpiece code with such a high quality that I can comfortably defend. The use of lambda expressions and a much more thorough understanding of JavaFX and frontend programming allowed me to thoroughly refactor VideoPlayer.java, and I am extremely pleased with the overall product, even though we were unable to incorporate it into our main VOOGASalad project.

In the end, I think I took on enough responsibility on the team and communicated quite well. Also, being the first project in which I did not pair program, I feel that I learned quite a bit and was able to learn a great deal about the frontend. Because Allan helped me learn lambda expressions while understanding how JavaFX and frontend programming works, I feel much more comfortable now tackling either end of a project. To be a better teammate, I hope to continue learning more so I can help others learn when they struggle. This is the same philosophy I bring to my time as a TA, and I believe that this is my greatest asset as a teammate to those who are struggling.

Design Review

Upon review of our entire project, the code is generally consistent in its layout, naming conventions and descriptiveness, and style. It is also quite readable and does not require comments to understand. Our team policy was to try to write our code in a way that allows any reader to know very clearly what is happening. As the result, the comments we did include mostly explain the why behind our project. Adhering to the DRY and SHY guidelines stayed a high priority throughout our work, so we were also largely able to maintain high-quality dependencies that are clear and easy to find. In addition, our implemented features are easy to extend due to effective composition in our hierarchies, and testing for correctness is quite simple with our JUnit tests. In the end, despite our best efforts, a few bugs remain in our project. They are discussed later in this analysis.

One of the three classes in our program that I did not write or refactor in detail is DragAndDropFilePane.java in our utilities.JavaFXutilities package, authored by Austin. This code was extremely well written and allows for simple extension. One important feature is that it allows the user to specify allowable file extensions. I was able to create DragAndDropAudioPane.java and DragAndDropImagePane.java and specify a wide range of allowable files. Since Austin's class also extends Observable, the panes can listen for when the user has dropped a file. This provides a simple UI/UX in any project, as long as DragAndDropFilePane.java (as well as any class that extends it) is integrated. Some features I would suggest adding would be the use of colored shades to indicate that the files are in the correct locations for dropping and the ability to drop multiple files at a time. These are only improvements to the UI/UX, but there is not much else I have in mind.

The second class that caught my attention is GSONFileWriter.java in the utilities.GSON package. David created this class to write general settings, game stats, towers, and baddies with Google APIs to a JSON file for saving. Although the code is slightly messy, due to some very long lines, this class is easy to understand because of its simplicity. Unfortunately, this class cannot be easily extended because it is written to handle specific objects. Also, many methods do not have the proper class modifiers and therefore allow inappropriate levels of access, but this class is important because it allowed us to use JSON files. Unlike XML files, JSON files cannot directly save JavaFX objects, but also does not require us to have our own parser. This is because JSON files have access to Google APIs. To reuse this code in a different project, however, this class would need to be changed to be more flexible and paired with object wrappers.

The third class of note is our AuthorController.java class. This class manages different scenes, such as path building and author creation, and holds the baddies, towers, and level objects that will be written to JSON files at the end of the authoring process. This allows our users to create their own games and even reuse this code in completely different projects. The user would only need to upload the images, sounds, and videos they desire with our drag-and-drop panes, and then they are free to mix and match to create other games. The code is also very organized and clean, but has many public methods. This needs to be resolved, but AuthorController.java is otherwise a class that I feel very supportive about.

We designed an XML-based game player. The individual components are placed in containers corresponding to panes, which are then organized based on purpose. For example, our right pane holds towers for the user to drag into the game environment, which occupies the center pane. The bottom pane holds options, such as selling and upgrading towers. The player relies on the game engine to spawn baddies and, in turn, informs the engine when the user has placed or upgraded a tower. In the end, however, our player is highly modular and has minimal dependency on the game engine.

Our game engine was designed with composition in mind to flatten out our hierarchy and achieve more flexibility. With composition, we could create necessary interfaces, and then implement the functionality we desired. Objects implementing the desired behaviors then help create system behaviors. This allowed us much more flexibility because composition is much more flexible than inheritance, which should only be used when clearly related pieces of code are reusable and fit under one common concept. Our situation, however, needed much more flexibility and adaptability, so we discarded inheritance for composition.

Our game authoring, however, does use inheritance because many of our tool panes inherit the BuildingPane.java superclass. When the user clicks on a tool, the authoring environment opens a new pane, through the use of the onEnter and onExit methods, that corresponds to a class. As a result, we have a seamless transition from pane to pane for improved UI/UX, and the user is able to quickly and easily navigate the building tools for their project. Because the codes for the panes are clearly related, reusable, and fit under a common purpose, inheritance remained an excellent choice for our game authoring environment.

Representing a specific game with our VOOGASalad project involves JSON files, the images and sprites for each actor, the game engine to operate on the backend of the program, and the game player so the user can interact with and enjoy the game. Because of our utilities, the user can enjoy the options of networking and communicating with a chatroom during a co-op game or a more challenging solo game in which the baddies use A* search, a form of AI, to continually recalculate the optimal paths to avoid towers and reach their destinations.

My code, which mostly focused on VideoPlayer.java, relies on lambda expressions to hide away details on the actions and implementations of my various components. At the same time, the variables and methods are named quite clearly to allow for readability, and the layout is carefully structured to allow for a smooth and largely top-down read. Instance variables were pushed down as far as possible during refactoring to limit their scope, so even within this one class, I was able to achieve shyness without compromising DRYness.

Our overall code is designed as is because it is closed to modification and open to extension. We were careful throughout the entire project to prevent unnecessary leakage of information, and our mixture of composition and inheritance did not cause any problems because we successfully isolated them away from each other by limiting their influence to only the game engine or the game authoring environment as needed. Our features, such as the drag-and-drop panes, media players, networking, and A* search are also implemented in our project, and

they can also be reused in other projects. This flexibility allowed us to very quickly extend features not in our earlier design without comprising its core.

One of the three more notable features involves heavy work from David. He implemented much of our game data with the files in his utilities.GSON package, which is an open source Java library that can convert Java Objects into JSON representations or convert JSON strings back to their equivalent Java Objects. This allowed us to bypass the parser that the XML files would have required in loading and saving games. Although the code in the utilities.GSON package is limited to specific objects, it works extremely well for our purposes and can quickly be modified to fit the needs of others.

Another feature is the ability to place game elements in our game authoring environment. Austin's work allows a user to create any level as desired by placing starting and ending locations, towers, obstacles, baddies, and other features. Even the path taken by the baddies can be varied, with lines, curves, and forks being available options during the construction of a level. A user can even create a campaign. The use of various BuildingPanes, images, GSON, and other classes in our game authoring environment allow a very expansive UI/UX if the user wants to explore. Of course, to support that, these features are extremely extensible.

A third feature involves a lot of work from Greg and Allan. The user needs to know how the amount of money and health after each level, and that information is stored and displayed throughout the gameplay in the right pane. Of course, this can be extended to include other information, and we even support multiple languages, some of which I translated. The user's information can be saved to JSON files if desired, but is also closed to modification so that it cannot be tampered. This allows our game player to limit information leakage and dependency on other parts of our code while providing a fun experience for the users.

Our original design handled extensions to the original specifications very well. Utilities such as the A* search, chatrooms, leap motion, and drag-and-drop panes were very easy to implement because we focused early in our progress on building towards the features and extensions we felt were most important to our project. Other features we could not include, but could have done so very easily if we had the time, include drag-and-drop panes for audio and video files, an audio player for background music and sound effects, and a video player for visual tutorials, game replays, and even video chat capabilities. At the same time, our original API did change in a few ways. For example, we had to add some DoubleProperties so the game player can bind values, such as level, health, and money, to equivalent values in the game engine. We also changed our game engine's interaction with the authoring environment so that it could operate there as well as in game player. This allowed us to more fully implement the authoring environment and support a more complete game authoring experience for the players, especially in testing their own products before saving.

One design decision that we discussed involved our chatroom. Originally, I had written a chatroom that uses multithreading, which would allow our server to respond quickly to each user by creating a thread for each connection. It would also remove any closed connections to prevent exceptions, lag, and the overuse of memory. The server also would have used enumeration and synchronization to

prevent any errors caused by multiple threads attempting to improperly call the sendToAll and removeConnection methods. Finally, a background thread would listen for incoming messages from other players. I would have preferred this chatroom instead of the version we have implemented in our final project for the above reasons. Instead, we have a single-thread chatroom that builds on Heroku. The single thread must check on the server every second for any updates as opposed to activating only when the associated user writes a message to the rest of the chatroom. I felt this was an inefficient design, but it was simpler and quicker to add to our project. In the end, although our chatroom works, I still prefer the use of multithreading purely because of the efficiency of the design.

Another design decision revolved around choosing between inheritance and composition. Inheritance was easier to understand and more familiar to many of us, but we decided to use composition instead to flatten out our hierarchy and implement the functionality we want. This is because composition starts with creating various interfaces that represent the behaviors our project needs. Objects can then implement the specific desired interfaces, thus creating system behaviors without having to use inheritance, which would require several more classes to achieve the same result. Inheritance is also more rigid than composition. This is because inheritance generally should be used only if we have clearly related pieces of code that are reusable and fit under one common concept. Because of the simplicity and adaptability provided by composition, I supported this decision and created several initial interfaces, though they were later discarded in favor of a modified hierarchy that maintained our principles of composition.

Our third design decision is related to the use of XML and JSON files. As discussed previously, we chose to use JSON files to avoid the need for a parser. Although XML files would let us directly save JavaFX objects, we felt more inclined to use JSON files because we would have access to Google APIs. My previous experience writing the CommandParser for my SLogo team led me to advocate for the easier JSON files, although I volunteered to write the XML files and parsers if we decided to do so.

Each of our design decisions turned out to work very well for us because our entire project remains closed to modification, but open to extension. We were able to uphold the principles of DRY, Shy, and Tell The Other Guy. Extensive refactoring helped us flush out any violations of good design even while we were adding new features. Even with the mix of composition in game engine and inheritance in game authoring, the designs of each sub-part was isolated from other sub-parts and served the necessary purposes of their sections extremely well. This shyness allowed each team to pursue their necessary objectives without compromising other teams.

In spite of these design decisions and careful planning, we still had some bugs in our game. One bug is the inability to run AI in our authoring environment. Although we implemented A* search so that our baddies can calculate the best paths to take through the open arenas, this ability is only found in the player. To bring this ability into our authoring environment, we would have to initialize all AI-related objects in the main game engine.

Another bug, also found in our authoring environment, is the inability to build a path for the baddies from the end location to the start location. Although this sounds minor, it is a manifestation of our limits because the user must build the path from the start location to the end location. One possible fix for this issue would be using a double linked list, as opposed to the current implementation of a simple linked list. This would allow the user to build in either direction, while saving whichever point the user started at as the proper start or end location.

One final bug involves our player. Towers occasionally cannot be upgraded if their selection shade overlaps with the upgrade button. This can be fixed in a couple different ways. One possibility is to lay the pane of the upgrade button over the pane of the game so that the selection shade does not cover the upgrade button. Another possibility would be to reduce the size of the selection shade so that it minimally overlaps with the upgrade button. Either fix would allow us to upgrade towers properly.

Code Masterpiece

For my masterpiece code, I am choosing to display VideoPlayer.java from our utilities.video package. DRYness and Shyness are important concepts I learned this semester, and I refactored heavily to maintain them. I consolidated and extracted methods from repeated pieces of code, and this further allowed me to enhance the Shyness of my code because some variables that were previously global became limited to single methods in scope. I could then convert them to local variables.

I also used YouTube as a guide on the design of my VideoPlayer to influence any UI/UX decisions I had to make. The purpose of this VideoPlayer is to allow the user to watch videos. This could be something as casual as game replays or as ambitious as video chat capabilities, combined with our chatroom, to allow much more interaction among players in a co-op game.

I also used lambda expressions throughout my code to help hide away details on the actions and implementations of my VideoPlayer, but my dedication to a high quality of layout, naming conventions and descriptiveness, and style means that the code remains quite readable. This is important to me because I did not understand lambda expressions during our time working on SLogo, but my code now clearly reflects that I was able to learn this new skill.

The only “smells” that remain are quite faint. For one, in calculating and formatting time, I feel that my code had reached a high equilibrium. I believe that there simply is no way to fully eliminate the 4 repeated lines of code without introducing a lower-level implementation, such as an array, or a much more highly sophisticated form of refactoring that I simply do not know. I also feel that my checkPlayerStatus and playOrPause methods could be improved in some way, but at the same time, one is a continuous check on the player, whereas the other one involves handling a click event on a button.

As a result, I feel that there is little I can do to remove this “smell” based on my current skill level. However, other than these two “smells” that may not even be removable (for one, JavaFX doesn’t allow the tightknit binding of the time slider and the play/pause button that I could use for the volume slider and the mute/unmute

button, which I feel negatively impacted my ability to refactor the checkPlayerStatus and playOrPause methods), I feel that the rest of my VideoPlayer.java masterpiece code displays excellent design and sufficiently shows my growth and improvement throughout the semester as a 308 student.

Considering that I had no prior experience with frontend, I am very proud of being able to shift from engine to utilities and quickly learn and understand enough about JavaFX to largely singlehandedly put together a VideoPlayer that emulates many UI/UX features found in YouTube… and most importantly, doing so with excellent design and refactoring.