Analysis

Sunjeev Devulapalli

Project Journal

Time Review:

I spent a lot of time on this project. I’d say I spent around 80 hours working. We started the project on March 16th and ended on April 30th right before the demo. Most of my time was spent with the rest of the Authoring Environment team discussing overall design and implementation with the Engine team. It took us a long time to figure out how we were going to retrieve the game engine classes in a way that wasn’t hardcoded. It wasn’t until after the first demo that we settled on using annotations in the game engine classes. A lesser amount of time was spent working on implementation. When it came to implementation I spent a lot of time adding special features like our visual scripting editor and debugging small problems so that our authoring environment was compatible with the engine. After design was complete, I wanted our authoring environment to stand out as much as possible. Unfortunately I didn’t get a chance to complete all the features I wanted but I think people still enjoyed the visual scripting I implemented.

I personally managed code by committing fairly often, but not pushing as often as I should have. I tested all my code in different main methods so I wouldn’t have to wait for another person on my team to implement a feature to see if my code functioned correctly.

Interestingly enough, the tasks I found easiest was working on my editors in the authoring environment while working alone. When I was in a larger group trying to figure out design, I had a very difficult time because everyone seemed to have different opinions on how to handle the design of the project. When I was working alone, I was able to design my classes how I wished which was much more enjoyable. As I stated above, working in a larger group discussing design was the most challenging part about this project but I think I learned many valuable lessons about working in code with others. For example, I think we spent way too much time talking theoretically and not developing real code. One of my big problems with working with a team is people seemed to just want to talk about design but I tried hard to push drawing design on a whiteboard. For me, it’s much easier to understand these concepts when we draw them out and see how all the pieces fit together. I will talk more about teamwork in the next section

Teamwork

Our team spent maybe 30-40 hours on design alone. Most of these design meetings occurred in the first 2/3’s of our project. In the last 1/3 we focused more on implementing features.

In the authoring environment, Peter was responsible for the backend, Negatu was responsible for setting up the annotation system and editor component superclasses, and Reyina and I were in charge of making sure all the editors and editor components worked and looked nice.

There were some warning signs that I saw early on that I wasn’t exactly sure how to address. Firstly, I didn’t actually enjoy working with the other members of my group all that much. That’s not to say they aren’t good, nice people, but rather the atmosphere was incredibly serious and perhaps slightly argumentative. If you had an idea that you wanted to discuss, there was a good chance that someone else in the group didn’t like that idea and would attempt to shut it down. Personally this discouraged me a lot from speaking up and taking a larger role in the group. On top of this, it seemed Negatu and Peter would constantly be arguing about what good design was. Negatu would complain about how poor our design was on nearly a consistent basis and Peter would argue with every point that he didn’t agree with. Reyina was on the opposite end of this spectrum where she just never seemed to have much input if any at all during our team meetings although my attempts to try and engage her thoughts. It was actually an incredibly frustrating team dynamic, which goes to show why I ended up enjoying working more by myself later on in the project. Of course, I do assume responsibility for this poor team dynamic, I’m sure there was something I could have done to bring our team together more, but I couldn’t quite figure it out. The second warning sign was that people often wouldn’t show up to meetings. Our communication was quite good in that we worked through groupme, word docs, and slack.com. However, when people weren’t showing up to meetings it was difficult to discuss more complicated topics that can’t be discussed online. Peter consistently couldn’t meet after 9PM and Negatu consistently couldn’t meet before 9PM. Reyina also seemed to often have some commitment when we’d want to meet. It was so difficult getting everyone on the same page. Nevertheless, our authoring environment team was able to get a ‘finished’ product by the end so I am at least happy with that.

I sent to our TA some praise for my teammates but I’ll include this here as well. Although Peter was quite difficult to work with, he was spending a consistent amount of time on the project and completed the backend which held up fairly well. If I had to give credit to one other person on the authoring environment team I would give it to Peter.

The plan for completing the project was constantly changing from the beginning. One of our biggest problems was we simply just didn’t start working. We focused a lot in the beginning on our engine team because we felt that was the most important part to design correctly in the beginning. Of course, as we waited for their code, we fell behind massively from other teams. Instead, I should have focused on building frontend components for our Authoring Environment while we worked with the engine to build our backend. Only by the last week did we see things begin to come together.

Issues

Commit messages are something I have attempted to work on since Cell Society when my TA told me to make them more descriptive. I believe I have done a fairly accurate job representing the work I’ve done with my commit messages. Sometimes I do forget to commit certain things, so features get forgotten in commit messages, but that’s something I’ll continue to work on.

I believe I closed 5 issues on Github. I’ll admit that issues are one of the features on Github I did not take very seriously throughout the project. I always kept what needed to be done in my head, but I should have turned them into issues on Github. It is also interesting to note that some of the issues on Github that I was assigned by others were features that we ended up not implementing.

I will talk about the Issue: [Create Multiple Selection Edito](https://github.com/duke-compsci308-spring2015/voogasalad_TuffWizard/issues/124)r. The multiple selection editor allows a game designer to select multiple things. For example, if you build an enemy, you want to be able to specify what tiles he can walk on. This issue showcases all the design issues we had as a team. One of the things we decided early on was that our frontend should not have access to the entire game engine objects we’re creating. So everything is displayed as a string in the frontend by receiving information from our backend via our Receiver class. Of course editors like the multiple selection editor need to be able to send a list of objects to the engine classes, not just a list of strings. Another related problem was that I needed to know what type of things to display in the multiple selection editor. Am I displaying a list of towers or tiles? The first issue we resolved by adding another method to our backend that got a list of a certain type of object, and the second issue we resolved by introducing a new annotation in the engine classes for editors like the multiple selection editor that gave it the correct type. Similarly we had to design the engine classes so that when we set this information, it is compatible with what our Receiver can handle. For example, our editors cannot make complex data structures so these are things that need to be handled in the engine classes, not the authoring environment.

Working with teammates was incredibly important for this issue. Primarily I worked with Peter to figure out how to send the correct information through our Receiver which then communicates with the game engine. I also had to work with many members of the engine team to figure out how I can modify their classes to be more compatible with this type of editor component.

Once this issue was closed, I don’t believe I worked on it again.

Conclusion:

We definitely underestimated the project. To estimate better I don’t think it’s a matter of working more, I think it’s more of a matter of developing real code early on rather than thinking about theoretical design problems.

I personally believe I assumed a lot of responsibility within the team. I felt that when it was coming towards the last week, people weren’t taking the project very seriously and just gave up on it. I spent the last week working nonstop to ensure that we had at least something to show off. I just wished others had taken the project as seriously as I did at the end so we could’ve had a working player as well.

The code that required the most editing was the Game Editor. The engine team wanted a strange data structure of a map with enums and game nodes which proved quite difficult for me to alter the design of the authoring environment to fit that. So instead I changed some code in the Game Editor and also some code in the Game Engine. By the end it worked very well. I am also working on the Game Editor for my masterpiece so I assume that code will change again.

I think to be a better designer I need to be able to better manage my time. If I see my team is taking too long on design, I need to push towards working in real code.

To be a better teammate I need to be more aggressive making sure people are coming to meetings and engaging in conversation. I think I was too passive with this and should have stomped my feet a little more.

If I could work on anything I would switch to the player team and finish the player. It was the one part of the project that just didn’t exist and didn’t get the attention it needed.

Design Review

Status

I believe our code for the authoring environment is consistent in layout. We use similar style in instance variable names, although this isn’t completely consistent, and constants at the top are consistent. I would also argue the code is readable as well. It’s fairly simple to follow since it’s split between frontend and backend. The frontend consists of editor components which are specified in the editor components class. There are a few classes that aren’t being used which we should have deleted earlier, but these might reduce readability slightly. As for dependencies, we used no public variables in our authoring environment. Everything can be traced back to its origin.

There are some features that are difficult to extend. Mainly these are the specific editor components. It’s easy to add a new editor component because of our editor component hierarchy, however changing an existing editor component would be slightly difficult. Especially for editors like Title Screen Editor where you would have to add new lines of code within existing classes to fix it.

The first class I will look at is the Inventory class. I chose this class because I believe it is good code. You’ll notice that all the methods are incredibly short. This code is used to talk to the engine. It’s held separate from the rest of the model even though being mostly similar except for one or two methods. I think the shortness of the methods is something I can learn from because most of the classes I wrote had some fairly long methods. This class as a whole looks fairly elegant. One slight problem that might exist is that it seems a lot of the methods have the same call at the top, which is to get a certain map. If this could have been extracted to a separate method on top which then calls the correct method for the next step, it could have removed some slight duplicated code.

The second class I will look at is EditorComponent. This is an abstract class that is also short, elegant, and sets up a lot of code that I wrote. Its one method is “setUpEditor”. As someone that didn’t write this class but was extending it, this made my job much easier because I knew all the code I needed to write had to come from this one method. The name also helped me identify its use. The two problems I see with this class is that there is an instance variable “fieldLabel” which is inconsistent with the other instance variable names and the class extends VBox. Instead, I believe this class should have a component VBox titled myRoot. This was the one confusing part about extending this super class.

The third class I will look at is Grid Editor. I am choosing this class because I think it has several examples of poor code that I am also guilty of in my classes. This editor was one of the editors that came together last minute and I’ll pick apart a few of the problems I see. I’m not going to complain too much about the length of the methods because it is frontend code but there are spaces where I think lambda functions could have been used to make the code more readable. For example in all the setOnAction methods, a simple lambda expression could be used. There are also several instances of magic strings and magic numbers that should have been extracted to constants at the top or put in a properties file. Finally, there is one method which I believe was copy and pasted from my code: getMethod(String name). If we had designed the special editors better, methods like this could have been extracted to a superclass so we wouldn’t have duplicated code. I mentioned this earlier, but this editor, title scene editor, and game editor aren’t very extensible in their features.

Alternate Designs

I don’t think our original design held up very well especially because we ended up changing so much over the course of the project’s timeline. The original API also changed extensively over the course of the project. Originally we were going to talk to the engine using interfaces but we ended up not doing that and instead we just use annotations.

Design considerations: One design consideration, as just mentioned, was originally we were talking to our engine using interfaces. These interfaces described the objects and were things like Collidable, Moveable, etc. I’m still not entirely sure how we planned on using these interfaces, but in the end we turned to simply calling objects in the Game Engine and using annotations to get the methods we needed. The annotation system is extremely extensible because if you wanted to add to the game engine, to add to the authoring environment you just need to add the appropriate annotation and an editor component will appear. The downside, however, is that these editors are very simple, containing simple textfields or sliders but no cool user interface.

A second design consideration was how we were handling special editors. There were some editors that simply couldn’t be limited to a javafx textfield. Originally we were going to try and make everything a simple editor component but in the end we decided that having specialized UI would be very nice. What we decided was to introduce a new annotation called SpecialMethodAnnotation which would create a special editor component: grid editor, game editor, or title screen editor. These editors wouldn’t be extensible because it would be one annotation for one editor but the trade off is that they would have a great UI.

A final design consideration was splitting the authoring environment into frontend/backend. Originally we had the backend as simply exporting to xml and all editors would make the engine objects and store them in some centralized location. We figured giving the frontend so much access to the engine was a poor design choice and so we came up with the Inventory system, which made it easy to store objects, and export to xml in a way that player and the authoring environment can read.

I would say the two best features of our design is the inventory system and the annotation system we created. The two most important issues in the design are removing some duplicated code within the special editors.

Code Masterpiece

For my code masterpiece I have chosen the class GameNode and the class SceneNode as an example of its implementation. Both of these classes belong to the Game Editor which created visual scripting. I think these classes show off much of what I’ve learned throughout the semester. GameNode is an abstract class that creates the framework for adding a new node into the scene. By extending this class, you must implement the following methods. In this way, adding a new node to the game editor is completely extensible. On top of this, I was able to remove all duplicate code from this superclass by putting any common node interactions in this superclass. There are also no magic numbers in these classes.

I will now go through the code design checklist and explain why I believe it fits all the requirements of good design.

Firstly, this code is extremely readable. I have commented all parts of the code so that anyone that wasn’t involved with it could understand what’s happening here. I’ve made sure all methods and variables are scoped correctly. The constructor is protected because it is an abstract class so outside classes should not instantiate it. All other methods are public, protected, or private depending on their use. Furthermore, this code contains no violations from java.

Secondly, modularity was taken into account. As stated above, there are no public or static global variables. There are a few setters and getters, but I feel these are justified. I have a getter for the Group. This is used by the game editor to visually represent the node. I also have getters for the In and Out which is also used by the game editor to determine whether the node was selected. Finally I have a setter for the receiver. This is used because nodes are made using reflection in the Game Editor and I then pass it the receiver. This was used to avoid any if structures.

Finally I believe the code is flexible. There is no duplicated code because of the use of this hierarchy. Also there are several abstract methods which are incredibly short. This is polymorphism in effect. For example, the Boolean method canDraw only returns true or false depending on the behavior of a specific node. Again, this is used to circumvent large if structures.