Thought Process: Phoenix

The purpose of this document is to show potential employers (or anyone) my thought process as I develop applications. It serves as a way to show off what I could bring to the table in terms of organization, coherence, and brainstorming. A few legal things before I get started: (1) the application I will create is for educational and personal use only. It is not meant to be sold to anyone and will always be open source. (2) I did everything completely and 100% on my own. I wrote the code, I wrote this document, and I will be the one to distribute this on my server.

Introduction

Phoenix is a space invaders esc game that runs on TI-83 and TI-84 calculators through MirageOS. There have been many versions of this game like this: http://www.ticalc.org/archives/files/fileinfo/148/14876.html and this https://www.youtube.com/watch?v=LTRujnUvgHU. I played many versions of this game throughout middle and high school, each very similar and all very fun and rewarding to play. I have actually created this game before in python as part of a collaboration exercise, which can be found here https://github.com/george-miller/Millercodes/tree/master/Phoenix. That version is pretty messy, and although it was a blast to make, I need to remake it.

Motivation

I started learning to program when I was in high school, reading an old python textbook my brother left from a college class he took. Throughout the rest of high school I casually kept learning to program with different structures and languages, and I felt as though I was gaining lots of knowledge. As I started my major as NYU, learning more structures and languages, I began to feel like there was something missing in all of this knowledge. I could write programs well and I felt I had the power to make a computer do a lot of things, but debugging was extremely hard and looking at even week old code became hard to understand. However, as any good student would I kept learning and doing my assignments as best I could.

It wasn't until the summer of 2015, when I got my hands on my first Computer Science theory book, that I realized what I was missing. It was The Pragmatic Programmer, by Andrew Hunt and David Thomas. After the first chapter I was hooked, I remember being glued to this book like it was the best fantasy I've ever read. Of course we have to keep software readable and concise! Of course we have to minimize classes /methods with too much responsibility! Realizations struck me and as I looked back at my old code, I was embarrassed to see I had done exactly the wrong thing. My massive takeaway from this book was that it is not good enough to make code work, it must be understandable and readable so that people can collaborate and debugging makes sense.

Finally I got my first taste of what real software development entailed, and I was hooked. I ripped through Clean Code, which helped give me concrete ways to write code better. After that I loved reading The Clean Coder, which gave me some insight on what it means to be a professional software developer. Sadly the school year started too soon and I kept my three Bibles by my side as I worked hard at my classes. With this new insight, my code was getting better with every line, I kept central ideas in my head and it felt amazing to apply some real idioms to my programming.

After completing one of my favorite classes of my fall 2015 semester (Object Oriented Programming) I told my professor about the love I had gained for CS theory and asked him for a few recommendations on more books to read (I didn't want to simply trust people on forums and I knew my professor had just gotten out of a great career as a developer). He recommended many books, so I bought all the ones that weren't freely available and I am currently reading Domain Driven Design, by Eric Evans.

Ever since summer of 2015, I have been working hard making (and learning how to make) real websites that had complex back end systems on my server (http://gmmotto.ddns.net). I love this and feel like I am leaning so much, so that brings me to winter of 2015, from which I am currently writing. I love making websites and managing servers and I couldn't be practicing a more useful skill. However, I don't feel like I have shown all the knowledge I have gained about understandable and logical systems as I would like.

To better show my knowledge and skills, I decided to remake the Phoenix game in a java applet. However, it wasn't enough for me to simply make the game, I wanted to be able to show anyone the process I go through to create applications. So there is the motivation for this document: **I want to show you how I think, and you can decide if I am a good developer**. Every time I work on Phoenix, I will write an entry in this document explaining what I am doing, why I am doing it, and everything that came into my head. It will essentially be like if I was in a room with you, brainstorming and working the structure of the game out on a whiteboard. There will be diagrams, there will be code, and there will be lots of learning. Lets get started.

Development

As I work through creating Phoenix, I will be separating the work into three parts: first will be analysis, then design, then implementation. By separating the work in this way, I will make sure that I have all the information I need at the correct time. I cannot design classes without analyzing what needs to be done, and I cannot implement classes without knowing the design of my program.

This is more commonly known as Waterfall development. Although this is less commonly used, by using Waterfall I can show my thinking in discrete parts (which is the whole point of this project). Also since I don't have time constraints and I know the end goals of the project from the start, Waterfall is a better option for this project. A small note, I will be using Test Driven Development when I get to the Implementation phase, as opposed to doing testing at the end like traditional Waterfall.

1/9/2015 – Analysis

As it is the very start of my project, I will need to ask myself a few questions (for simplicity, I will work out the answers in a bullet point form):

- What work needs to be done by the program?
  - The most obvious work that needs to be done is the gameplay itself. This means collision checking, input event handling, drawing the sprites on the screen, creating enemies and making them move/shoot, and moving from wave to wave.
  - The program also must save the players progress as he plays through the game. We have to save upgrades the player was rewarded with, the current level he is on, and his personal settings like controls.
  - Lastly the program must have a way of navigation and instruction. A menu system on startup, a pause menu in game, and some kind of tutorial.
- What inherent constraints does the domain (the Phoenix game) have?
  - Since the domain is a game, there really isn't too many constraints. The beauty of Phoenix (and most games) is that you can make really anything happen within the context of the game. You can have lasers, homing missiles, bombs, upgrades, multiple players, multiple ships under your control, etc. The limit is only imagination.
  - However, the context of the game does enforce some constraints so that the game can be called Phoenix. There must be:
    - A ship (or group of ships) that the player controls
    - Enemies for the player to destroy (ex. simple enemies and complex bosses)

        - Rewards for the player to attain (ex. new weapons, ship upgrades, high scores)
- What are the discrete parts of the program that can be separated?
    - The program can be separated into pieces correlating to my answer to the first question:
        - A piece that controls the menu system (title screen, instructions, pause menu)
        - A piece that controls gameplay (collision handling, etc as mentioned above)
        - A piece that controls saving and reloading saves
    - I would like to note that these all can be three distinct pieces of the system that are modular and communicate with each other in a discrete way.

1/10/15 – Analysis

Now that we have separated the program into subsystems (gameplay, save/load, menus), we can start separating those subsystems into subsystems. From now on, I will be essentially determining responsibilities of systems and how they will work together, and then when design comes I will use these definitions of responsibilities to make models and class diagrams. Let's start with the **save and load system analysis**:

- What do we need to save?
    - Levels beaten and scores for each level – so we can lock future levels and also users can compare high scores. Also so the player can maintain progress between sessions.
    - Current health/money – so that a user can build up a bank to buy expensive weapons and they won't lose their bank when they load their save.
    - Weapons unlocked – so the player can maintain his rewards between different sessions.
    - Controls settings – so we can access control settings throughout the program and also so that the player can maintain control settings between sessions.
- What responsibilities does this system have?
    - We must be able to read the save data elsewhere in the program, so it must have a way of accessing the save data.
    - We also must be able to write to the save data when, for example, the user completes levels or changes his settings.
    - There must be a way to maintain the save data throughout sessions. We can't have the save data deleted whenever the user exits the program!

Next is the **menu system**:

- There needs to be a menu system where the player can navigate and interact with the game on startup so he can change settings and launch the game.
- What are the different states of the menu system I want to make?
    - A title screen to launch at startup and give access to all other menus
    - A level select screen to allow the player to replay and view all the levels
    - A settings screen where the player can change his controls and manage saves
    - An instructions/tutorial screen for new players
    - A pause screen for in game help and utilities (links to quit and get to settings)
- What are the responsibilities of this system?
    - This system must take control of the program at startup.
    - It must allow the user to access all elements of the program in an intuitive way.
    - It must initiate the gameplay system, and be able to modify the save data.

Last on the list is the **gameplay system:**

- What are the responsibilities of this system?
  -
- What characteristics does this system need to have as a whole?
  - Extensibility: I want to be able to create and add completely new and different things that I won't be able to come up with right now. For example, when I make a new weapon, I don't want to have to mess around with the collision system or the drawing system, I just want to add the functionality of the new weapon and have a plug and play experience. Same idea for new enemies, new ships, multiple players, everything needs to be plug and play.
  - Phoenix's gameplay (in the most general of terms) consists of a player shooting enemies with a weapon while dodging enemy weapons. Thus the responsibility of the gameplay system is to allow this to happen.
- With the requirements from the above bullet in mind, can we separate the system into subsystems with different responsibilities? Yes:
  - Positioning: Everything needs to be positioned correctly: arriving on the screen in a certain way and moving in a certain way. The enemies need to move based on a predefined pattern, and the player needs to move based on the players input.
  - Drawing: Everything needs to be drawn on the screen in the correct position and with a specific sprite.
  - Collisions: Collisions need to be handled between projectiles and the players and projectiles and the enemies.

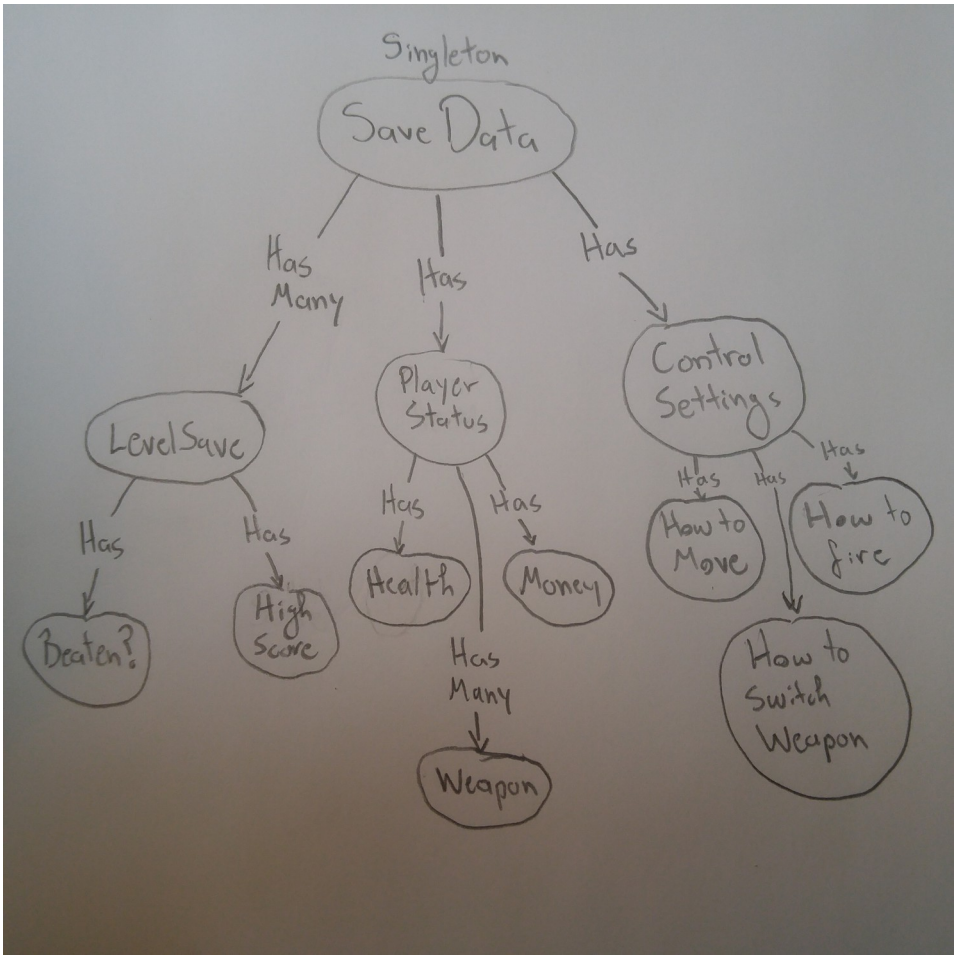
1/11/2015 – Design

Now that we have analyzed each subsystem and determined what each subsystem's responsibilities are, we can start making design decisions about how these ideas can be implemented in an Object Oriented language (Java) running as an applet on a server. Let's start with the **save system design**:

- How should we save it?
  - There are many options for this:
    - Cookies on the browser – Downfalls: you have to use one browser on one computer, and there's no way to save and reload if you uninstall your browser or clear your cookies.
    - Logins on startup – Downfalls: Puts all the weight on me as the server and creates a lot of extra work, forces me to integrate a database and forces people to remember their login and password, also I might have to implement a whole user system with login recovery.
    - Allow anyone to unlock anything at any time – Downfalls: Ruins the spirit of the game.
    - Import/Export of save files – Downfalls: Users must always remember to import and export their saves any time they quit or restart the application.
  - The best option seems to be import/export save files. Although this puts some work on the user, it is more robust because you can easily take your save file onto any computer and also be able to record points in your game and easily go back and replay. Also there can be warnings and reminders in the menu system so that it will be hard for people to forget to import/export their save files.
- How should the rest of the program access the saved data?
  - This should be pretty obviously implemented as a singleton, where the singleton is initialized/exported on every import/export of a save file.
  - The menu system will need a reference so that it can show which levels/weapons are unlocked, and so that it can write to the save singleton when weapons are bought.

- The gameplay will need a reference so that it can read which weapons are unlocked and so it can write to the singleton when levels are beaten and when health/money is gained/lost
- Only certain parts of these subsystems will need a reference to the singleton, so when I map out those subsystems I will have make sure I don't give references of the singleton to parts that won't use it.

Ok, great. So I am pretty happy with this analysis of the save/load system with import/export of save files, so let's draw a simple diagram to work off of and evolve.



1/12/2015 – Design

Next on this list is **menu system design**:

- The first decision is whether I want to use Applet or JApplet: The main difference according to Oracle is that JApplet allows use of swing and awt components and Applet only allows use of awt components. After reading this stackoverflow answer (http://stackoverflow.com/questions/408820/what-is-the-difference-between-swing-and-awt) I want to use JApplet so that I can use swing components. The main reason for this is compatibility. I don't want anyone's OS messing with the GUI making it not work, and since swing is “more-or-less pure-Java”, this will give me more compatibility. (Of course users will need to have a JVM installed for it to be compatible but that's required either way)
- I have used these features before, and I can simply have a bunch of classes that inherit from JPanel, each representing a state of the menu system.
- Since I want to have a pause menu in game, as well as an animating background when

traversing through menus, I can use a JLayeredPane as the root pane:
    - In game, this allows me to have a JPanel with just a pause button on it as the top layer and a JPanel that draws all the gameplay as the bottom layer.
    - When traversing menus, this allows me to have a JPanel with any GUI elements I want on top (for the menus) and a JPanel that draws a pleasing animation rendered below the menus.
- So essentially I will have a JPanel for the title screen, the control settings screen, the pause menu, the instructions/tutorial screen, the level select screen, the animation below the title screen, and the gameplay graphics.
- How should the panels interact with each other? How should switching be done?
    - We are going to have multiple panels each with links to each other, they must manage each other somehow.
    - One option is whenever the user clicks a link for a new panel, we instantiate the panel and switch to it, deleting the old panel we came from.
        - The benefit of this approach is less memory usage, since we always delete the old panel.
        - The problem with this approach is that if a user exits a panel, it is fairly likely that he will return to said panel after doing things in the new panel. For example, if a user goes from the title screen to the settings screen, he probably will want to return to the title screen so that he can play the game. Same goes for the tutorial screen.
        - Another problem is the instantiation and deletion of objects requires compute time. This means less responsive menus and a less pleasant time for the user.
    - Another option is to instantiate all panels at startup, and keep them all for the lifetime of the program.
        - The benefit of this approach is super responsive menus since we don't need to instantiate any panels after startup.
        - The problem with this approach is more memory usage. We will have 7 JPanel subclass instances sitting in memory at all times.
            - NOTE: 7 object instances is not a huge memory overhead, if it were 50 instances, this problem would be way more substantial.
    - Last option is to make a caching system where we try to predict where the user will go based on where he has been or expected places
        - This approach is the best of both worlds, keeping low memory usage yet also maintaining responsive menus. (It also is an awesome example of the universal idea of caching! Man, my CS teachers were right, binary trees and caches are the right answers to every CS problem).
    - Comparing design choices generally, caching is the best option. For the purposes of this thought process (which is to show my thought process generally as a develop applications), I should choose caching since it is accordance with everything I've learned about making good design choices. However, in this particular situation it is not the right choice. Let me explain:
        - I mentioned in the introduction of the development section that I do not have time constraints for this project and I don't. However, I am a practical developer who tries to think practically about the problems he is given. If I was working on a project for a company and I decided to design and implement a complex caching system to deal with 7 object instances, I wouldn't be surprised if they fired me. Assuming these object instances aren't super large*, it is pointless to spend all that time making a caching system when 7 object instances really isn't that much memory overhead.
        - * = Knowing that Java has very smart developers, it is very unlikely that they will create classes who's instances are so large that 7 of them will cause too much trouble for the

user's computer to handle.
    - Based on the reasons given above, I will pursue option 2, and instantiate all JPanels on startup.
- Now that we know that there will be 7 JPanel instances used to navigate the menu system, there will have to be somethings that hold each instance and allows switching and access to each panel. That can simply be the JApplet itself.
- However, the JApplet doesn't need to have all the panels, based on the dependencies each state of the game has, we can organize ownership.
    - The animation panel is only going to be under the title panel and the level select panel. All other panels will never have access the animation panel.
    - The gameplay panel is only going to be accessed from the level select panel.
    - The pause panel is only needed by the gameplay panel, all other panels never need to access the pause panel.
    - The instructions and settings panels will be accessible from anywhere, so they should be owned by the JApplet.
    - The level select panel will be owned by the JApplet, since we want to be able to exit to level select for easy level switching.
    - The title screen will be owned by the JApplet, since it is required to start and we want to be able to exit to the title screen.
- With these dependencies in mind, we can make a diagram: