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

The first question I need to answer is how will I develop? I know that I cannot design classes without analyzing what needs to be done, and I cannot implement classes or tests without knowing the design of my program. Taking this into account, I will work in a Waterfall style of development, where first I analyze, then I design, then I implement.

However, I know that I cannot simply analyze the whole thing, then design the whole thing, then implement the whole thing. When designing, there are going to be required features that I realize I need but forgot to analyze during the analysis phase. Same goes for parts that I have to implement that I didn't realize I need to design until I start implementing. For these reasons I need to be able to go back and add in information to earlier phases while I am working on later phases. I also want to be able to use it as a reference document to aid as I develop.

To keep things organized, I will separate this section into three parts: analysis, design, and implementation. I will start with analysis, then go into design, then go into implementation. However, if I notice a lack of information in the previous phases, I will go back and add onto that phase before continuing to work on the current phase. This keeps thing organized while also fully showing off my thought process.

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.

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 analysis:**

- What are the responsibilities of this system? In other words, what is the basic gameplay of Phoenix?
  - The basic gameplay of Phoenix consists of a player shooting projectiles at enemies while dodging enemy projectiles. This system must create, coordinate, and display these three elements: player, enemy, projectile
- 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.
  - For this goal to be achieved, whatever represents a ship, projectile, or enemy must be completely encapsulated. This means that the entirety of an element's behavior must be encapsulated in itself.
  - This also means that there must be functionality to manage all of the elements, telling them what to do and when to do it. (A management system) Otherwise, all the elements would just be floating around not knowing what to do.
- What needs to be encapsulated in each basic element? What is a basic element's behavior?
  - An enemy and a player do the same things, but on opposite sides of the screen, they need to:
    - Draw, shoot, define a hit-box, handle being hit by a projectile, maintain hull integrity (aka health), define damage (explained later because ships are able to hit other ships)
      - You may be thinking, wait don't enemies/players need to move? Yes they do, but after a lot of thought and rewriting I came up with a better solution: a modular positioning system
      - The player doesn't need to arrive on the screen in any special way, but for some enemies the first enemy may arrive in the first second but the last enemy might arrive a minute later. Enemies may also want to move in drastically different ways, some may want to go classic space invaders (up, down left, right) but others may want to move in random directions, or be going on and off the screen constantly. Because of this we need a way to coordinate groups of enemies that may want to arrive in a different way.
        - To solve this we should create another level of indirection, there needs to be a middleman between the enemies and the management system to position enemies in certain ways, let's call it a positioning system. It should also be modular: we should be able to use any positioning system with any enemy.
        - Responsibilities of the **positioning system** are:
          - Start enemies at initial positions
          - Move those enemies in a certain way each frame
  - A projectile does similar things as enemies, projectiles need to:
    - Move, draw, define damage, define impact points (by using impact points instead of a hit-box, the collision detection become a lot easier, and we can have multiple impact points if the projectile is big)
    - Note: A projectile can only enter the game when an enemy/player shoots it, so a projectile doesn't need to know how to enter the screen

- What are the responsibilities of the **management system**?
  - Interacting with the positioning system:
    - Moving: Tell the positioning system when to move each element
  - Drawing: It needs to tell everything when to draw itself on the screen.
  - Collisions: Collisions need to be handled in a certain way between elements, and it needs to tell each element when something has hit it.
  - Projectile creation: Each enemy/player needs to be able to shoot projectiles at each other, so there needs to be functionality that allows each enemy/player to shoot.
- All of these responsibilities of the management system have straightforward rules except for the collision system. What should be able to hit what?
  - In the real world, everything would be able to hit everything. That makes for some tough gameplay with two players, and also would result in a TON of enemies dropping bombs on other enemies. Imagine you start level one and you wait 10 seconds for all the enemies to kill each other, then you kill the last remaining enemy. That's pretty lame.
  - To solve this, I like the idea that enemy projectiles are immune to enemy projectiles and player projectiles are immune to player projectiles. No “friendly fire”.
  - However, it makes for some bad gameplay if as a player my lasers hit enemy bombs, so we can't have everything be a projectile.
  - However, I do want to make the option of kamikaze ships possible, since that would make really cool boss fights and it makes sense since in the real world ships would be able to hit ships.
  - Let's put this information together to define rules for what should be able to hit what:
    - Ships are able to hit ships (that covers the kamikaze case)
    - Projectiles (bombs/lasers) are able to hit ships
    - Projectiles are not able to hit projectiles
    - No friendly fire
  - Now with some defined rules we can define the collision systems responsibility:
    - It must check collisions between every player and every enemy/enemy projectile
    - It must check collisions between every enemy and every player/player projectile
    - It can ignore every other collision
  - It is obvious to give projectiles damage, but since ships can hit ships, how do we know how much damage a ship does when it hits another ship?
    - We will have to give ships a way to define damage just like we do with projectiles
- There are a couple of things we forgot to mention that need to be implemented for the game to work. The first is, how will we determine what is in each level?
  - First we need to determine what a level consists of so that we can encapsulate it. Here is all the information we need to make a level:
    - A Phoenix level consists of an arbitrary number of waves. When the player defeats all the enemies in a wave, he moves onto the next wave and so on until he completes the last wave, beating the level. A wave consists of:
      - An enemy type and the number of enemies in that wave
      - A positioning system to determine how enemies enter and move about the screen
  - The definition of what is in each wave of a level should not be part of the code anywhere. It is not a function that the program could do, what each wave consists of is simply information about the game. Arbitrary information should never be put into the code, it should be put in an alternate data structure (for example a database, or meta files). For these reasons, we cannot hardcode levels. However, we need some way to get the information

from the data structure so we can use it to create enemies.

Data:
We have discussed the need for saving and retrieving data (like save data and level data) but we have not decided which data structure to use when storing this data. So, how should we store and query the data?

- There's really only a couple options to store data efficiently: a database or meta files.
- A database would allow us to easily query and store information without much extra code. We could simply create a facade to access the database, and it would be quick and simple to implement. However, a database requires one big thing: an extra application. If anyone wanted to run my game on their local system, they would have to install this extra application and set it up. Another drawback to the database is that you cannot edit the level data or save data without using the database application, there's no easy way to quickly change a couple values in the data.
- Meta files, on the other hand, will require a slightly more complex facade. We will have to deal with creation of files and direct reading and writing of files. However, the payoff will be big. There will be no extra applications, and anyone can easily go into the meta files and change anything they want, since they will be simple text files.
- In the interest of making Phoenix a more portable game, requiring only the JVM to run, I am going to go with meta files. We will call this the meta file repository.

What are the responsibilities of the **Meta File Repository?**

- Contain functionality to read and write meta files
- Define rules of how meta files are organized and where they are stored

We could have the Meta File Repository create objects directly, but that would be outside the scope of the repository. We need something to work alongside the repository, connecting objects to data. We need a Meta Factory. Responsibilities of the **Meta Factory**:

- Allow construction of objects from their meta file definitions
- Allow destruction of objects to save them into meta files

Design

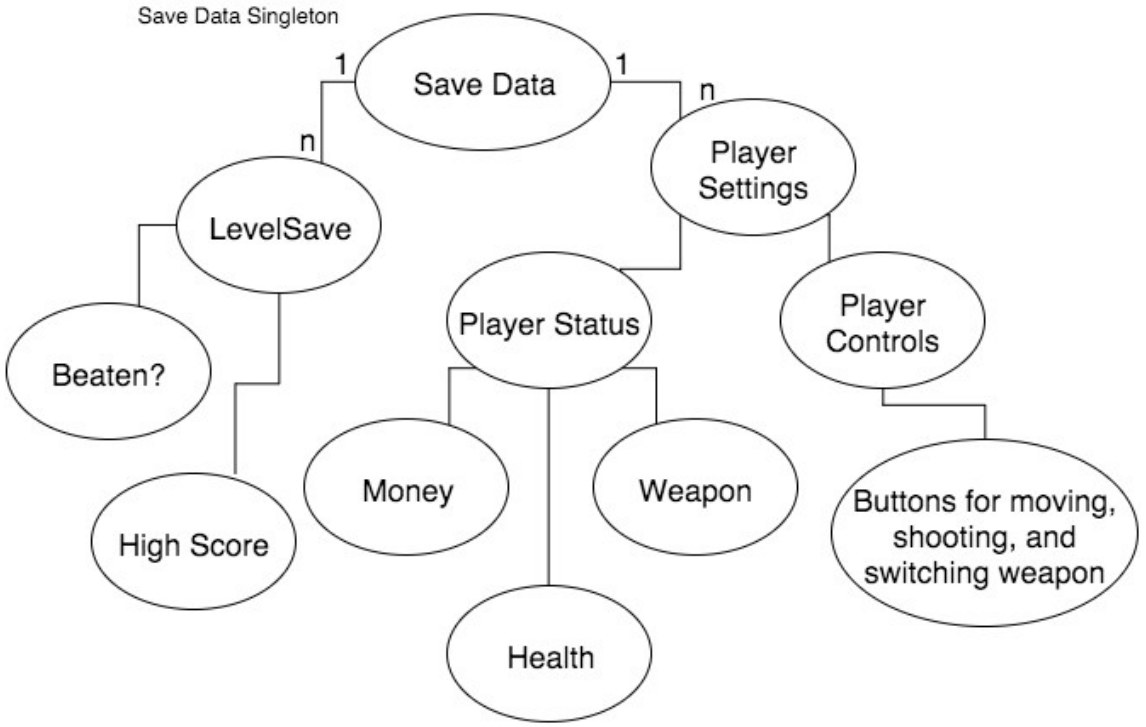
Now that we have analyzed each subsystem and determined what each subsystem's responsibilities are, we can start making design decisions about how these ideas can be implemented in an Object Oriented language (Java) running as an applet on a server. Let's start with the **save system design**:

- 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.

- If we are importing/exporting save files, we need a default save file if the user didn't import one.
  - True, so when we initialize the game, we should create a 'default' instance of save data. This could be done with serialization, hard-coding, or meta files. I want to be able to easily change the default save state, so that takes out serialization (since I would have to recreate an object and serialize it every time I wanted to change one value). Hard-coding would work, but that's just bad style. Functionality in code, arbitrary details in an outside data structure.
  - So we will put the default save values in an outside data structure.
- 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 safest way to implement a singleton in Java is to use an enum.
  - 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.

Let's draw a diagram of our Save Data:



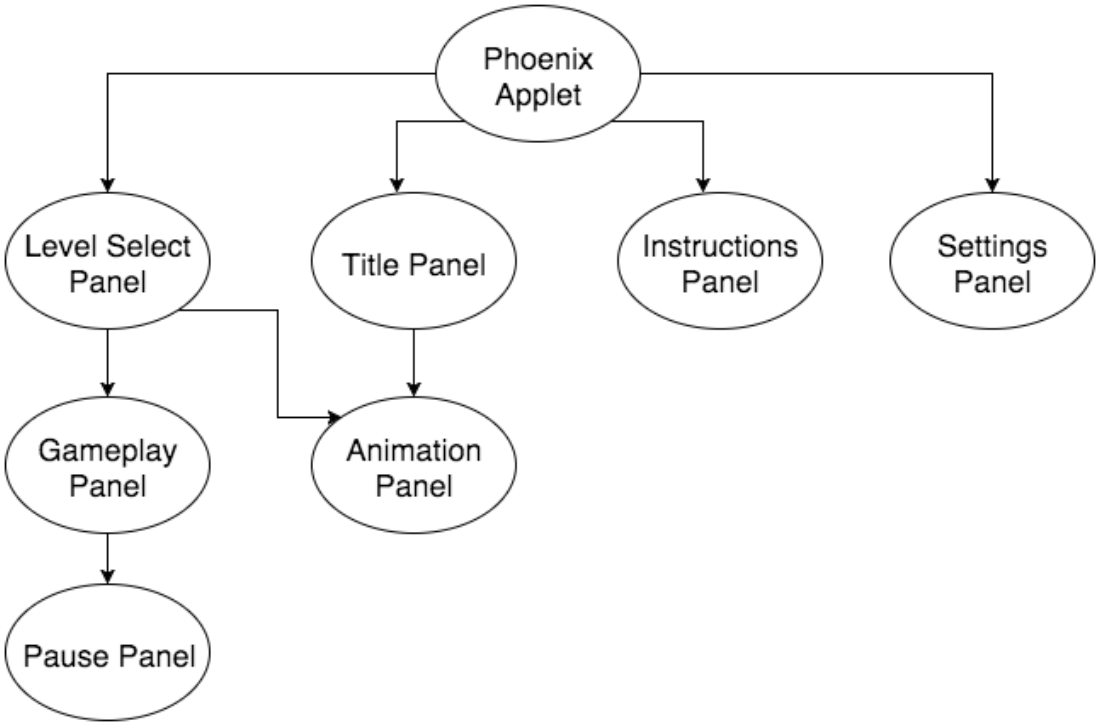
Next on this list is **menu system design**:

- 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.

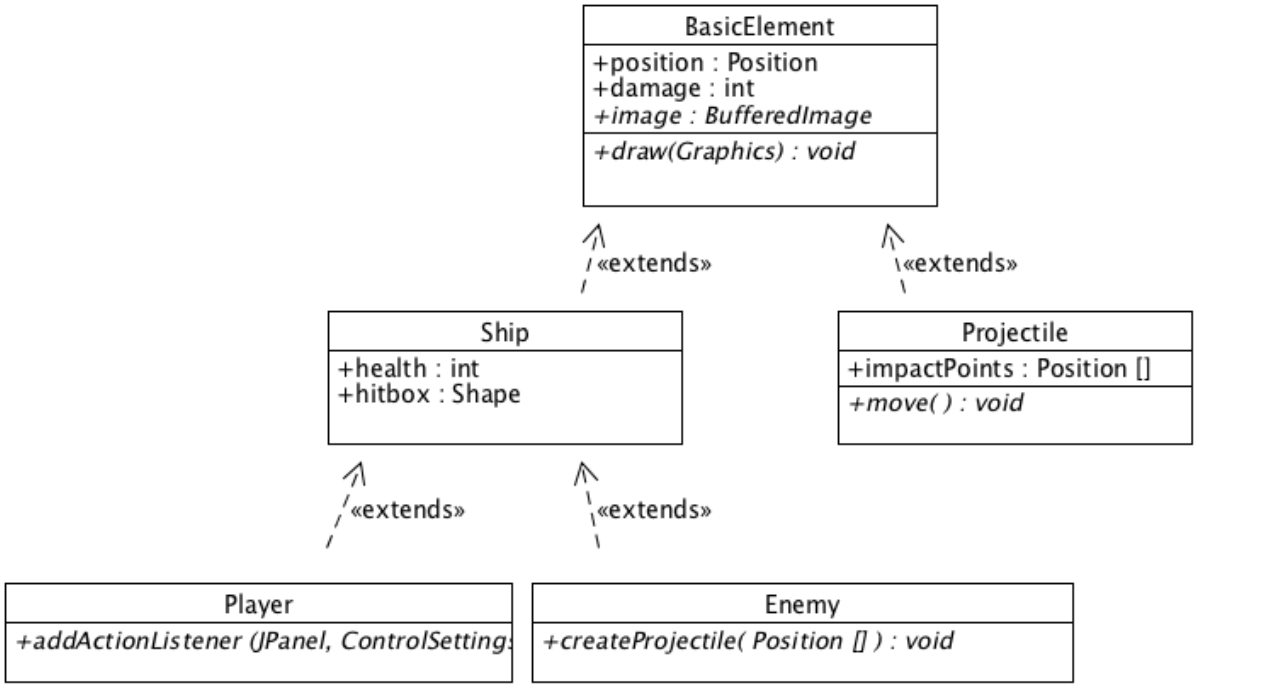
- Now that we know that there will be 7 JPanel instances used to navigate the menu system, there will have to be somethings that hold each instance and allows switching and access to each panel. That can simply be the JApplet itself.
- However, the JApplet doesn't need to have all the panels, based on 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:



Let's discuss **game system design**:

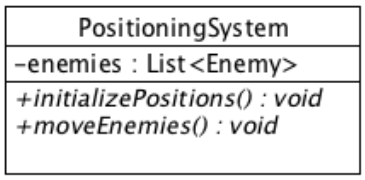
- As stated in the analysis, the three basic elements of Phoenix gameplay are enemies, players, and projectiles. From the analysis, we know that we have to design basic elements, and we need to design a management system.
- How can we represent the basic elements?
  - We can represent all of these things with a contract (in java we can use interfaces or abstract classes to enforce that contract). In this way, the other parts of the gameplay system (most notably the management system) can use all enemies, players, and projectiles in the same way, no matter what actual underlying enemy/player/projectile is doing. This achieves our goal of extensibility since we can create any class that implements the contract and all we have to do is plug it in and it will play perfectly.
  - We already know what each element needs to do (from the analysis), so we just need to design an interface or an abstract class to represent them.
  - Each basic element has a state: It needs to be positioned somewhere on the screen thus it needs x and y positions. Because it has a state, it would be better to use an abstract class to represent each element since interfaces cannot represent state, only API.
  - What abstract classes do we need to create?
    - A lot of functionality overlaps between all the basic elements, so we can create an abstract class that all other elements can inherit from. Let's call it BasicElement. This classes contract will contain:
      - Image reference (all basic elements have an image that represents them)
      - Position variable (x and y coordinates)
      - Damage reference (since all basic elements can damage things)
      - Drawing functionality: The draw function will need to take an argument since we will be using Java's AWT paint method. It will take a Graphic object to draw on.
    - Since we are using a positioning system for the enemies, enemies don't need to know

how to move, but projectiles do. Projectile's extension to BasicElement is simple:
    - Movement functionality
    - Impact points array
  - All ships (enemies/players) need to extend upon BasicElement's contract. Let's define what new functionality the ships need:
    - Health variable: to define the strength of a ship's hull and determine when a ship has been destroyed
    - Hit-box definition
  - The way players shoot will be different from the way enemies shoot, so we need to extend ship's contract for each case. First let's look at player's:
    - Players will use action listeners on the JPanel to fire and move, so there needs to be a method that adds those action listeners based on the control settings.
  - Enemies will be given a chance each frame to create projectiles. Here's the enemy extension:
    - Projectile creation functionality: This can be implemented as a function that is called each frame, giving a chance for each ship to create projectiles, spawn kamikaze ships or whatever other special functionality a ship wants to implement. This will be especially helpful for bosses, because we may want to implement a complex boss fight. Since the projectiles created may be targeted, we should implement this with an argument that is possible targets, then allow the creation algorithm to choose which of those targets to choose.
- We have successfully represented the basic elements. Let's draw a diagram for what we have designed.



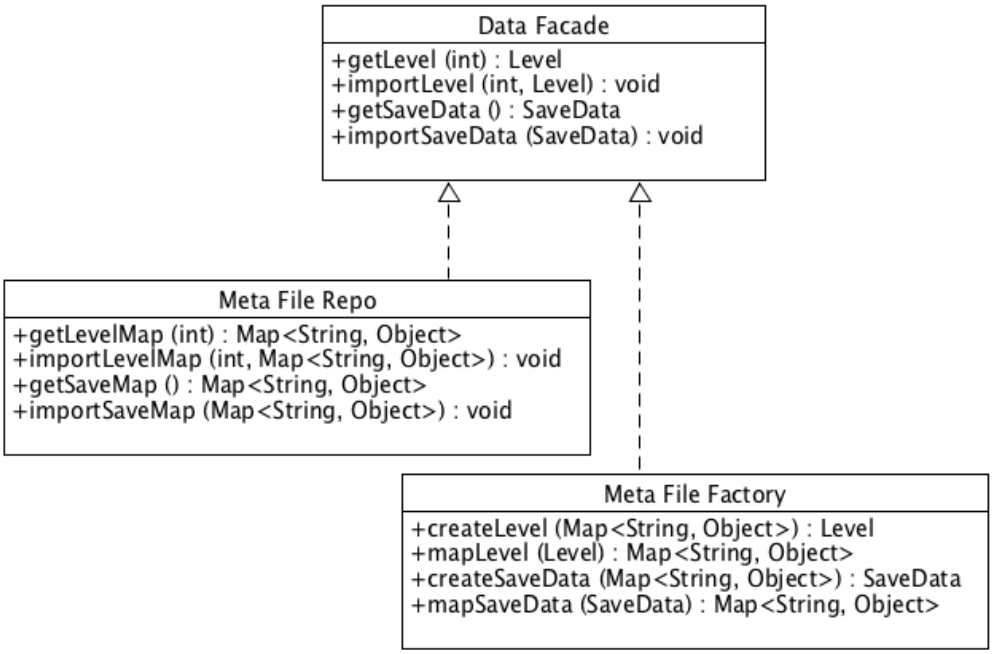
Let's move on to **Positioning System Design**:

- The positioning system has simple enough responsibilities where it can be encapsulated in a single abstract class, then we can extend that abstract class to define functionality.



Let's discuss **Meta File Repository and Factory Design**:

- These two objects are purely functionality. They do not have any state, their state is defined by the meta files that are located on the system. Because of this, both of these classes should contain only static members/functions.
- Eventhough I made the decision to go with meta files over a database, if in the future someone wants to use a database, they should be able to easily. For this reason, we need to create a facade, so that the implementation of saved data can be easily changed. Let's call it the Data Facade.
- What does the Data Facade have to do?
  - Give other parts of the program level objects on demand
  - Give other parts of the program a Save Data object on demand, and the ability to export a modified Save Data object.
- How should meta files be designed?
  - To keep meta files easily readable and easily modifiable, meta files will be a simple list of key/value pairs. This means that it will be the job of the repository to get the key/value pairs from meta files, then it will be the job of the factory to construct objects from those key/value pairs.



Still left to design: management system

Implementation

As I implement, I will be using Test Driven Development: a popular and effective way to create clean and supported code right from the start. The first thing I am going to do before I do any coding is setup my test suite. I want to be able to run all my tests at any time to make sure that all my code works as expected. I don't want to have to deal with collecting all my tests and manually compiling them all and running them, so I will write a bash script to do all that for me.

After much research and experimenting, I found a great test suite helper called ClasspathSuite, in the cpsuite extension to JUnit. ClasspathSuite runs every method with the @Test annotation in the classpath. So all I have to do is put all the folders with tests I want to run in the classpath, and they will all run. I don't want to memorize that classpath so I will just write a bash script that finds all the tests, compiles them, and runs them with the correct classpath. It could be possible in the future that running all the tests every time will be slow, and if that time comes I will have to rethink my suite, or make a few intermediary suites.

My first tests will be tests for all the designs I have written so far. I will start with the BasicElement and it's subclasses. Since these abstract classes represent contracts, I will create tests that test the general contract for all subclasses of the abstract classes. To achieve this, I can write tests using org.junit.runners.Parametrized. What this will allow me to do is write tests, then specify objects to run those tests against. So when I create subclasses, I will simply add them into the parameters collection, and I will be testing the general contract for all subclasses!