**Describe how your team integrated code from the team members. Which Integration**

**Strategy do you think your team used? Explain why. (250-350 words – 5%)**

The way we constructed our video game was each of us was assigned a specific module/component to work on. For example, one of us would be in charge for designing the over-world while another could be responsible for player/enemy class design,graphics and UI for battle scenes, level up systems, game balancing, etc.. The way we integrated our code was by linking all the sub-components together as they were written and iterate upon the design from there. The integration strategy we used would most likely be all-at-once integration. Our general process was as follows: we would write a file, and once enough code was written to where we could see the results and validate the design, we would link together the components. It was difficult to test the components by unit, because, for example, if we didn’t have an over-world that allowed us to enter a battle, we couldn’t test whether the battle was designed correctly. It was optimal to use all-at-once integration to test the components simultaneously.

We’ve found that this integration strategy is not susceptible to many of the typical faults of all-at-once integration (late bug finds, faults being difficult to isolate) because the project itself is small. Because the entire game is a collection of, say, 10 separate javascript/html files, faults are very easy to detect, because we have a small number of well-defined components with different tasks. For example, if at the point of integration we find that an enemy sprite is not loading as expected, the fault can be easily tracked to the html file responsible for incorporating/opening the images.

**Deployment plan (500-700 words - 5%)**

What are the steps required to deploy your project?

Who is the potential market?

What will it cost to deploy it?

Example costs:

• app store costs

• costs to get your game on XBoxLive

• costs to print disks for distribution

• costs to buy domains

• costs to attend conventions and conference and set up booths

Costs $10-20 per year to maintain a domain

Potential market - flash game/online video game players

Market - people who like single player games

In our implementation of project four we were tasked with putting into place the final version of our previously assigned project, in which we were required to build a prototype of our final game. In our minds it made more sense for us to host this game online through the web. In doing so that would allow the players of the game to just go onto a domain on the internet and be able to play our game using their local device. The game is a 2D turn-by-turn roleplaying game, our target audience is individuals that are interested in games like pokemon and roguelike games. This market is very large as we can have anyone around the world play our game, they just have to have access to the internet. In finding our audience we can run many tests that search through games available on the market right now that are similar to ours. Then we would want to figure out the groups of people playing these games and find a way to market the game to them. We expect the market to largely compose of single player PC gamers. From there we could venture into markets more familiar to our players and attend events to drive popularity and word of our game.

The first step to deploying this project would be to buy a domain name. This would likely cost anywhere from $2 to $20 per year, depending on the top level domain and the servicer we buy the domain from. We would likely try to choose something on the lower end of this range, since we do not anticipate a large amount of traffic and would likely only receive customers who were explicitly told the URL of our game (in other words, we do not expect any people to just happen to find our game while browsing).

Two additional factors to consider in the context of domain costs is, firstly, to simplify recurring expenses, we could pre-pay for the domain for a set period of time, say, five years. After this period, we could choose to extend the domain depending on the game’s popularity. Secondly, it is a logical investment for us to purchase domain names that are similar to the original games. Although this may seem like a redundant cost, it makes it more difficult for competitors to ‘rip off’ our game by snagging domain names similar to ours.

The second step would be to set up the server that would provide the game to the players. This could be a personal machine owned by a team member (in which case the only cost would be electricity), or it could be a server provided by a hosting service like Amazon Web Services or Linode, in which case the cost could range from $30 to $70 a month. This would likely be the better option, since it removes most of the cost/time required for configuring or maintaining the hardware, allowing us to focus on maintaining the software.

**Maintenance plan (500-700 words)**

How much it will cost to maintain your product for the next year?

Example costs:

• Costs for hiring developers

• Monthly or annual fees for servers or domain names

• Monthly or annual fees for your distribution platform (e.g. app store,

XBox live)

There are several costs to maintaining our project for the next year. These costs include, but are not limited to, personnel costs, server maintenance, and domain renewal fees. A general rule of thumb is that we can expect maintenance costs to be about sixty seven percent of total development costs. We estimate the total cost of maintaining our game for the next year is roughly 600 to 800 dollars, which we have broken up into different parts below.

Personnel costs are likely the least significant cost for maintaining our product. Because our game is relatively basic with a simple code base, we do not expect to need a dedicated full-time staff devoted to game maintenance. Rather, we, as the original developers, could edit the code base as we see fit to correct small bugs and implement minor repairs. For larger, more structural fixes, we could consider allocating an hourly wage (from ad revenue, see below) to one of us willing to commit more time to game fixes, probably fifteen dollars an hour.

Server maintenance is the most expensive of the costs for maintaining our project. If we chose to use a hosting service like Linode, we would expect a minimum maintenance price of 30 dollars a month, totalling roughly 360 to 400 dollars over a single year. This price could potentially be reduced significantly by running the server on a personal machine of one of our team members, but that would incur several additional maintenance costs, such as electrical bills, hardware maintenance, and additional personnel costs to maintain and monitor the hardware in addition to the software. Combined, this can be a difficult thing to estimate, so for now we will opt to estimate server and domain maintenance costs at 400 dollars per year.

Another cost that our group could see in the future of the games life cycle is the hiring of more software and game developers to expand our game. The hiring of more game developers would allow us to grow the game much faster and serve the community of our players with more content for them to experience and enjoy. The pay rate of these developers would likely depend on how much our game is growing and being played. It also depends on how much income we would be making from our game. The income would likely be coming from advertisements that are run next to our game, possibly to the side where the player's view of the game is not interrupted by an advertisement. The payment would have to be around the twenty to twenty-five dollar amount. This would be an hourly pay rate and would allow for the employees to be paid on a timely schedule that worked around a yearly calendar so employees knew when they were getting paid. I feel that it would also work best to allow our employees to work from home, thus allowing us to not have to pay for a central location, saving us money in the long run.

**Code Review (500-700 words – 5%)**

Undertake, as part of a team activity, an inspection of a medium-size code

segment using either a Walkthrough or Inspection as described in the lecture on

Software Verification

Document the following:

• List of faults detected

• Description of each fault

• Who was assigned to fix each fault

• How the fault was fixed

For our code review, we chose to do an informal walkthrough, rather than a rigorous inspection. We chose to walk through the “level up” section of the code, since that was the most complex and newest feature for project 4. Liam was the walkthrough leader, taking everyone through the code and how it worked, while everyone else made suggestions for improvements or potential faults in the system.

One fault/area for improvement detected during the review was the level up system in the context of move replacements. When the player levels up, they are able to replace one of their moves with a list of random moves as well as upgrade their stats. The UI for the move replacement and stats, however, didn’t match. The former was confided into a small textbox used for battle, while the latter prompted an entirely new menu that allowed the player to select and de-select certain stats. Because the stats menu was graphically superior and crowded out other unnecessary graphics remnants from the battle scene, Mickey was assigned to enhance the size of the menu screen and delete the old graphics objects, as well as make a standalone menu for replacing moves. This ensured that the UI for the entire level up process looked clean and uniform.

One other fault detected during the walkthrough was that the player was not given an option to keep their current moveset. In other words, the player had to replace a move even if they didn’t want to, otherwise the game would be stuck. To fix this, Mickey added a simple “don’t change move options” to the menu.

An additional fault detected was that the logic that implemented the player’s stat changes after player selection used stats that may have been modified during battle rather than the player’s base stats. For example, if during battle the player’s defense stat increased from a move, a level up to the defense stat after the enemy was defeated would calculate the new defense stat based on the in-battle modification. This was a fault because a) it allowed player’s in-game moves to have permanent effects (bad game design) and b) it resulted in overleveling/overpowering player stats, which undermined the game’s balancing mechanics. To fix this, Matt was tasked with creating dummy variables in an object called “oldStats” that would copy the player’s stats at the beginning of the battle, then use those stats to implement stat changes during the level up process.

A final area for improvement was code neatness/organization. The level up system relied on a large handful of functions that extended to hundreds of lines of code, all of which was contained within a single file. Additionally, the entire level up system was written in the same file as the logic that implemented battle. To ensure organization and make sure code was easily findable/fixable, Liam was tasked with moving the level up system to a new file. This involved changing the locality of some variables, but ultimately made the code significantly more legible and organized.