Jason Goldfine-Middleton
932675265

CS 162 Final Project Reflection

Design Plan:

The name of the game is "The Chronicles of Sharnia". It is a space adventure game in which the player has a fighter ship, some items, fuel, and some money. The goal is to find the Chronicles of Sharnia, which hold the ancient wisdom of the human civilization of planet Earth. There are 25 star systems that the player can travel between and many of them contain planets to land on. The user has 75 turns (each jump between systems counts as one turn). If the player is in a system with a planet, he will have the option of looking at the map, buying new items, selling items, buying a new ship, refueling, repairing his ship, asking for information about the location of key items, and sticking around for another turn. In order to complete the game, the user will have to locate four key items: the Electron MipMap, the Eye of the Hack, the Decoder of Doom, and finally the Chronicles of Sharnia. These items must be found in that order. In fact, the game engine will not hide a key item until all prior key items have been discovered.

In addition to the options that a player has while visiting a planet, the player may jump from any system to one of its adjacent neighbors. In this case, that means left, right, upper, and lower neighbors, if they exist. The 25 systems are arranged in a square, meaning that edge and corner systems will not have four neighbors. In the code this is handled by making those neighbor pointers null. The player will not be able to jump from his current system to a neighboring one unless he has some fuel.

To handle to assignment's requirement to have multiple types of derived "spaces", there are four types of star systems: empty, enemy, friendly, and neutral. When the player enters each of these types of systems, different things happen. The logic for this is controlled by the special() virtual function in the System class. The essential flow is the following: player potentially finds money (enemy, friendly, neutral), player potentially gets taxed (enemy, friendly, neutral), player potentially has fuel restored by another traveler (empty), player is ambushed by enemies (enemy, neutral), player accesses system menu and may choose what to do. The four derived System classes are EmptySystem, EnemySystem, FriendlySystem, and NeutralSystem.

There are some other things that may "happen" to the player when he enters or is about to leave a system. As a player enters a system, he may find the next key item he needs. When he is about to leave a system, he may get drunk and waste a few turns. This can only happen after the user has discovered the Eye of the Hack.

Although there is a map feature which displays the outline of the whole universe, the player will only see names and positions for systems he has visited. The map feature is not unlocked until the player finds the Electron MipMap, so during the early turns traveling is a bit like shooting in the dark.

Jason Goldfine-Middleton
932675265

The main.cpp file is responsible for displaying a menu to the user, allowing him to play the game, read the introductory story, get some hints about effective gameplay, see who has beaten the game quickest, view credits, or exit the program.

The Game class controls the gameplay and determines when the user has won or lost. A game is lost when the user runs out of turns or is destroyed in an ambush. There is technically no way to start battles at will, but as an incentive for the user to accept always being ambushed, each time the player survives a battle he will earn some money. It is crucial that the player buy good ships and good items as early as possible in the game, because if even one battle is lost the game is immediately over.

The Player class aids the Game class by encapsulating current information about the player's in-game status. It is the glue that links together many of the classes and prevents the Game class from getting too messy and doing too many things at once. It has the largest public interface of any class.

To decrease the likelihood of the player's ship being destroyed in battle, the player has access to various items, which are represented by the Item class. An Item has a name and a description. If it is one of the four key items, it is marked as such. Otherwise, it also has a mod that improves an aspect of the player's ship. Although the code is designed so that an Item could affect more than one aspect of a ship, none of the items do. Every non-key (regular) item has a base price, which is modified depending on the type of system it is being sold in. The Item class is not sub-classed, and gets its variety via its constructor.

On the other hand, the Fighter class, which represents a ship that the player may either own or encounter in a fight, has eight derived classes: Bolq, Lamborqini, Raw, SleazeLiner, Suxel, TrollE, Voltsweeper, and Windweeper. The player always begins a game with the Voltsweeper, which is a very fast but weak ship. It is also one of the cheapest ships. Unfortunately, all ships lose 100% of their value once taken out of the shop, so there is no trade-in value when switching to a new ship. Each Fighter has several aspects which determine how well it tends to do in fights: speed, health points, armor, and a weapon. The speed determines the order ships attack in during a fight. All fights are group fights, so the ships take turns attacking. More on this later. Health points determine how much life is left in a ship. Health points may be fully restored by repairing the ship. Armor weakens all successful attacks from opponents by a set amount.

The Weapon class represents Fighter weapons. A ship's weapon is how it damages other ships. Each ship type has only one weapon, which may either be area-wide or targeted. An area-wide weapon attacks all opponents, while a targeted weapon only attacks a single opponent. Each weapon has a base attack value and an accuracy value.

As there are many occurrences in The Chronicles of Sharnia, there is an Event class. This class has multiple derived classes, namely: AidEvent, DrunkEvent, FightEvent, KeyItemEvent, MonetaryEvent, and TaxEvent.

An AidEvent may only occur on an empty system, and provides an opportunity for the player to have his ship refueled if he is stranded. It is best to avoid having these situations occur by refueling, because many turns may pass before a helpful ship comes along.

A DrunkEvent occurs randomly once the player has found the Eye of the Hack. It wastes a random number of turns and only occurs on systems containing a planet.

A FightEvent is the most complex type of event. It can occur when the player's current location is an EnemySystem or a NeutralSystem. In an EnemySystem, 4 enemy ships will ambush the player and 3 ships will come to his aid. In a Neutral System, 3 enemy ships will ambush the user and 5 will come to his aid. All fights are group battles and, as stated above, faster ships attack first. The fight consists of multiple rounds in which each ship has a chance to attack the opposition. A fight ends when either the player's ship is destroyed, ending the game, or all enemy ships are destroyed. Information about the happenings of each round are displayed on-screen.

A KeyItemEvent occurs when the player enters the system containing the next key item. The user is only allowed to hold 7 items at any one time, so the user may not have room to pick up the key item. The user must land on the system's planet and sell a non-key item in order to make room, and then stick around for another turn.

A MonetaryEvent may occur on any non-empty system and consists of the player finding a random sum of money no greater than 3000 ducats.

A TaxEvent may occur on any non-empty system as well, and consists of the player being fined a random sum of money no greater than 1000 ducats. If the tax is greater than the amount of money the player has, the player's money drops to 0 ducats. A tax event occurs with greater likelihood in enemy systems.

Please see the source code for more details.

Test Plan:

The strategy used for testing this program will be to implement the design incrementally, beginning with the simplest, most independent functions and testing them as they are completed. It will also use pre-conditions and invariants as my guides. This will mean establishing the file structure from the beginning and working on the smallest functions first, regardless of which class they belong to.

Once the basic functions have been tested, I will build the functions that depend strictly on those already tested. These will be unit tested as well. Lastly I will build functions that rely on many other functions and classes. These will be more complicated to test because I will have to actually play the game. In order to handle this, I will leave out features that are not essential to the overall goal of the game until I have tested it. Once I am satisfied with these functions, I will

implement the rest of the functionality bit by bit. Copious print statements will be used. Only some print statements will be left for the user, but since methods are called repeatedly, it may seem like a fair amount of text for the player.

As there are many menus involved in this program, I will test them all with all options in many different places. For example, I will thoroughly test the menus for each type of star system, the store menus, the sell menus, the ship store menus, and the main menu.

The last thing I will test will be game balance. I want to ensure that the game is adequately challenging and interesting for the player. This will require a lot of actual gameplay and even asking my friends to try playing my game. If they win or lose too easily, or they feel the game is otherwise imbalanced, I will resolve these issues. This is absolutely the last part that should be dealt with because I may find that I need to implement other features and having a sturdy foundation is crucial for further enhancing the game.

I have decided not to include exceptions in this program because the only possible issues that should arise will be from file reads, and the functions related to these issues already throw exceptions and there will be no way for the game to function without these files in place. Therefore, the user should ensure that all text files are included in a subdirectory called “text” inside the working directory. The zip file produced by the makefile will contain the correct directory structure, so simply unzipping the file should be sufficient.

Test Results and Adjustments to Design:

For the most part I was able to follow my testing plan. Overall, I have to say that there were no issues with unit testing the simple functions and even the functions strictly dependent on them.

However, once I started working on the FightEvent class, things got a bit hairy. Even with many print statements it took me awhile to find the root of my issues. The main stumbling block to resolving bugs there was that my function had gotten too long. I believe this was because I did not break down the fight loop enough in my original design. I ultimately was able to resolve the issues in the long function and got it working, but I decided to refactor it by creating two private helper functions, attack_all() and attack_one(). This removed around 250 lines of redundant code.

I had off-by-one issues with menu options, but these were easy to diagnose and fix. One issue that came up when I tested the menus was something that I whipped myself with a wet noodle for. It turned out that some neighboring systems were not displayed in the system menu at certain times. Luckily, I was able to use the map feature to look into the issue more deeply. I discovered that systems located next to the edge systems did not realize they had neighbors on the edge. This was due to simple modular arithmetic errors in the Map class. This disturbed me because I did not catch these issues when testing my Map class, which was responsible for setting up all the systems and linking each with its neighbors. Clearly I did not thoroughly unit test the get_right() and get_left() functions.

Jason Goldfine-Middleton
932675265

The most exciting part of testing was actually playing the game. I discovered many issues with my print statements, but these were easily repaired. In the end I was fairly satisfied with the results, but unfortunately the flip server terminal is not too smart, and does not word wrap the lines at blank spaces. This leads to line breaks in the middle of words. The IDE I used to develop this program was Xcode and it broke lines at logical places. Unfortunately, this is not an issue that can be easily resolved through code because of my design choice to use string templates for printing interesting monologues from the characters the player encounters on each system. I think it is fine the way it is though. The best part about the file input design is that I can easily add new strings for the templates whenever I come up with them in the future. There will not even be a need to recompile.

Thanks to many run-throughs on the part of myself and allies, I decided to change a few things in the game. I decided that the original maximum number of turns, 250, was way too high and have since dropped it to 75. Via smart playing it is very likely that the player can win before the time runs out, but with unintelligent playing the time will run out. This makes the game more challenging and stressful, but I think more fun. I also increased the price of items and made them less effective. This was to encourage the player to upgrade his ship, rather than simply buying overpowered items and keeping a cruddy ship. Another big change was dropping the prices of the more expensive ships, so that they would be more attainable. I also decided to decrease the average tax because I found that it wiped out player's savings too easily. I am very satisfied with the final result and I hope to eventually update it so that the code makes use of C++11 features, which are not fully supported by the version of GCC on the server.

I also tested the program using Valgrind and did not see any memory leaks. I tried to ensure that all classes allocating new objects on the heap took responsibility for deallocating them. This was not always possible but I did find ways to ensure that there were never memory leaks by keeping heap objects constrained as much as possible. I look forward to using smart pointers in C++11 to make these tasks trivial.

Personal Thoughts:

I truly enjoyed this project and although I could have done far less and satisfied the requirements, I wanted to see if I could make a fun game that I would want to play at random points in the future. I think the final result fits the bill. I think others will find it amusing and challenging as well. I have injected a bit of my humor into it, so if you want to get to know me a bit better, just play the game. :P

There is a lot of room to expand on this game and I feel like it was a great capstone for a very enjoyable and educational class. I have learned a fair amount about software design from this class and I am getting a little better with object-oriented programming. I intend to continue my learning and this program has only deepened my passion for continuing forth on this journey. May the Tao of Programming continue to guide us all!