Conquer the Seas

Design Document

Benson Perry Matt Dannenberg Brian Shaginaw

Section 1: Introduction

1.1: Purpose

This section of the Design Document will outline the purpose of the entire design document. It should be read regardless of the further intentions of the reader, as it will describe which sections will be most relevant for different types of readers using this document. This is a living document, in that it will continue to be edited and modified throughout the design process until its eventual completion and submission.

1.1.1: Foreword

Conquer the Seas is the first game being developed by the in-house software team LIFDOFF for the McGill University Software Engineering Project course. The purpose of this document is to describe the technical design of the software in detail. The purpose of this document is to describe the software design of the Conquer the Seas game and to explain how and why these designs were chosen. This document will detail the design, specifications, programming management, and test cases and testing procedures of the game.

1.1.2: Scope

Section 2 will start with the architectural considerations, then move to design considerations, and finally go in-depth into describing the high-level design of the game. This includes the discussion of the design pattern, the final domain model, and the final deployment diagram. **Section 3** will detail the actual software organization, including a view of the subsystems in both text and drawing. This will contain a detailed breakdown of one specific subsystem, with UML class diagrams and a list and explanation of class variables for this subsystem. There will also be a critical section which contains algorithm selection, critical code snippets, a state chart, and calling sequence diagram. **Section 4** discusses programming management by outlining the directory structure and programming tools, software building method, coding agreement, mitigation procedures, installation procedures, and training guidelines. **Section 5** will describe the test cases and testing procedures used for this project. This includes a description of test-driven development, functional test cases, the error logger, and how we built classes for auto-testing. **Section 6** is the appendix, which contains definitions and other useful information.

1.2: Audience

This document can be read from several different perspectives. It assumes a rudimentary knowledge of computer games and ideally some experience in software development. Regardless, the types of readers and which sections apply most importantly to them are as follows:

1.2.1: Game players

For the reader whose main focus is playing the game, **Section 4** will be the most pertinent. This section will contain information about how the game is stored locally on the user's machine, and will contain information about the install and setup of the game. It also contains the system requirements of the game. **Section 3** may also be of interest if the player wishes to modify code or see the inner workings of how the game is designed.

1.2.2: Developers

For the reader whose main focus is understanding and possibly modifying the code, **Section 3** will be the most important. This section outlines the specifics of the code, going into great depth about all aspects of the structure of the code. It also details algorithms used and has many detailed diagrams to explain the module/object decomposition. Individual subsystems are described in detail, so a developer could read this to understand how the software was written from the ground up.

1.3: References, Terms, Definitions

Conquer the Seas is a large software project that uses many technologies that will be discussed throughout this document. This section contains a listing of terms and abbreviations that will be used and may be referred back to for maximum clarity in further sections. It also contains references to the other two documents written for this software project so far.

1.3.1: References

Two documents have been written previous to this document. The Preliminary Design Document contained information about the basic functionality and gameplay of the game. It also contained a timeline of milestones and deliverables for which the project still holds to. The second document is the Requirements Document, which describes all of the functions and specifications of the game and is based on the IEEE 830 standard. It also contains many terms and definitions (including acronyms) that will be useful throughout this document as well. Reading these two documents before this document would give the reader a much greater understanding of the project. An implementation document will be released in the weeks following the submission of this document.

1.3.2: Terms and Definitions

This section is a list of terms and definitions that are essential to understanding the rest of the document. Other acronyms have been previously defined in the Requirements Document in Section 1.3.2.

- **pygame**
  - pygame is a set of Python modules designed for writing games in Python. It is free and released under the LGPL License (http://www.pygame.org/LGPL) and will need to be installed on all user machines along with the corresponding version of Python. For more information about pygame, visit the pygame website: http://www.pygame.org
- **Unified Modeling Language (UML)**
  - Unified Modeling Language (referred to as UML throughout the rest of this document) is a standardized modeling language for graphical representations of software design and software systems. Diagrams in this document will follow UML specifications.
- **Sprite**
  - A sprite is a 2-dimensional image or animated image used in video game graphics. In the context of Conquer the Seas, sprites will be used to draw and animate the different units and items. These graphics will be created entirely by the development team.
- **HTTP**
  - Hypertext transfer protocol. A request-response protocol that is used to send and receive data over the internet. In this project, it is used for non-LAN clients to connect to the server.

1.4: Polices and Tactics

This section outlines the goals of this document and this version of the software build, as well as the guidelines and development methods used for creating this document.

1.4.1: Goals

The software project outlined in this document is intended to achieve and address the following goals:

- Create a video game that is enjoyable to a diverse user base
- Incorporate advanced features like networked multiplayer and loading/saving of networked games
- Use detailed design methods that allow for easy-to-understand code which in turn may be modified by players or other developers

1.4.2: Guidelines

This document will be developed and maintained according to specific guidelines to maintain consistency throughout the document. The guidelines this document adheres to are as follows:

- Abbreviations and terms that are used will be defined in **Section 1.3.2** unless otherwise specified

1.4.3: Development Methods

This section outlines the methods used in developing this document. This is so that the design of this document can be traced back to these steps. The methods used in developing this document are outlined:

- A general outline of the sections of the document was created first
- Using this general outline, the introduction was fleshed out, with information filled in as relevant sections were completed
- Descriptions and other important explanatory text was added to diagrams
- Terms and definitions section was updated with newly added terms
- Proofreading and extensive editing

Section 2: The Design

2.1: Architecture Considerations

In designing Conquer the Seas, we have been using several different techniques and some specific technology. For writing the code, we are using a combination of different text editors: Vim, jEdit, and Notepad++. For maintaining our code and as an online repository, we are using github. Links to further info about these technologies can be found in **Section 6: Appendix.** We did not use an IDE.

The code itself is written in the python programming language. The team designing Conquer the Seas felt that this was the best choice of language because of our familiarity with python. Python is being used in conjunction with a set of modules called pygame. These modules provide lots of support for the creation of games in python, and provides many features we required.

For multiplayer functionality, we are using network sockets to allow multiple machines to connect to a central server. This is built directly in python, without using pygame.

2.2: Design Considerations

When designing Conquer the Seas, certain assumptions were made and code was made to fit certain specifications. There were design constraints and assumptions that dictated how the code was to be designed according to the client, and there were also constraints and assumptions made according to real-world issues, like installation or player computer capabilities.

2.2.1: Design Constraints and Assumptions

In designing Conquer the Seas, certain constraints were initially placed by the client:

- The game must be turn based
- The game must be multiplayer
- The game must have AI
- The game must have terrain generation
- The game must have resource management

There was also a time constraint placed in that the game must be completed before the end of the semester.

2.2.2: Real-world Constraints and Assumptions

In addition to the constraints placed on us by the client, we have also designed according to several real-world constraints and assumptions. The following assumptions were made:

- Users have internet access or are on a networked LAN if they wish to play multiplayer

- Users are running Windows XP or later

In addition to this assumptions, the basic assumptions that user machines are fast enough to run the game, they have a working mouse and keyboard, and so on. The following real-world constraints were kept in mind during design:

- Game must be able to be installed and played simply
- Game must have simple enough controls and/or detailed enough information for first-time players to successfully begin and play through a game

There are also constraints that come from using pygame and python:

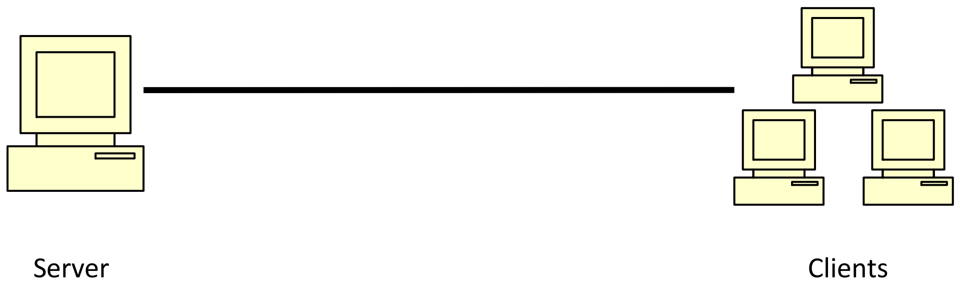
- mp3 files are not compatible with pygame, so all sounds will be .ogg files recorded locally by the design team

2.3: High Level Design

This section describes the basic architecture of Conquer the Seas. It contains a system overview, a more detailed discussion of the design patterns used, and a final domain model, all with explanations.

2.3.1: System Overview

The architecture of Conquer the Seas is at its most basic level a Client - Server architecture. There is one server (hosted on a player's machine) and multiple clients can connect to this server. The user running the server also runs thes client, which connects to the server on the user's local machine. This means a user running the server always is a player in the game they are hosting.

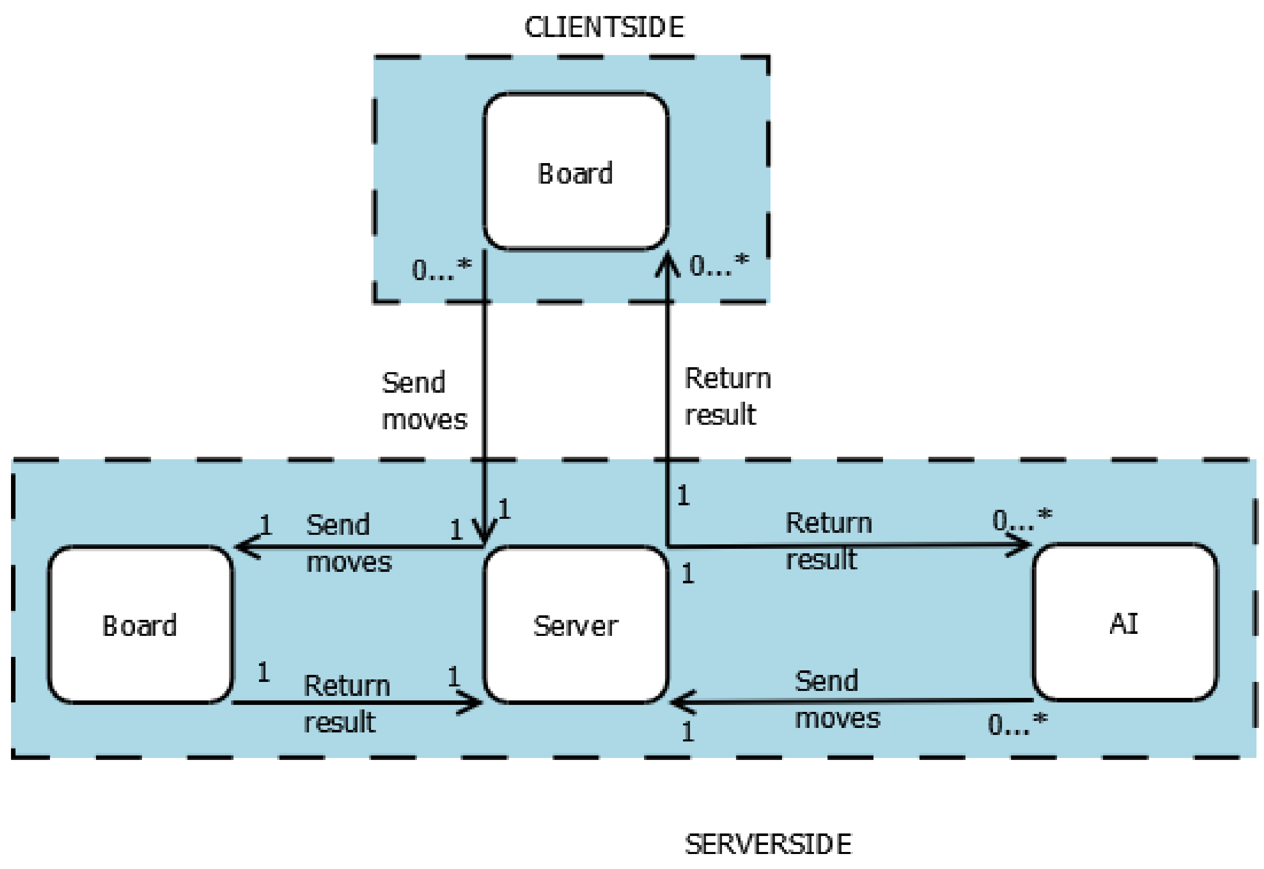


This architecture allows for two distinct roles to be filled by users of the software. The server needs to be able to handle multiple incoming connections over LAN or HTTP.

2.3.2: Design Pattern Identification

The design pattern used in Conquer the Seas is strictly Client-Server. Each client sends requests (commands) to the server, to which the server responds appropriately. This model provides several benefits. One is that each machine does not need to do all of the computations on the game end, they only need to send requests and receive information. The machine the server is on has the bulk of the computation to do, so ideally it would be hosted on the most powerful computer out of all machines being used to play the game. Another benefit of this design pattern is synchronization - all machines will be submitting their moves to one central server, which will only process and respond to requests once all players have locked in their moves.

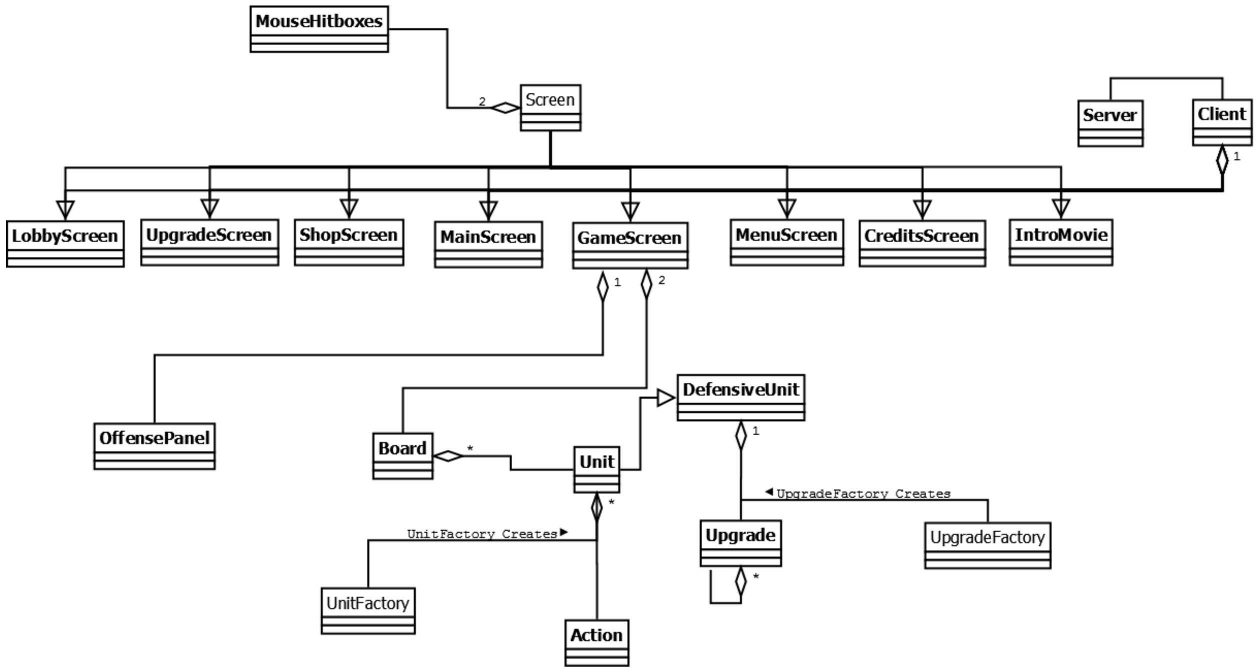
The other potential design pattern that could have been used would have been peer-to-peer. Under this pattern, users would connected directly to each other and send moves directly between clients. We opted for the Client-Server model because centralizing all computations to one location made the most sense for this game.



This diagram shows the interaction between the Client and Server. The blue rectangles indicate individual machines. One machine hosts the Server, a Board (which is what the Player uses and interacts with), and possibly the AI (if playing a single-player game). The smaller blue rectangle on the top indicates a second player connected to the player hosting the server. The Player classes send moves to the Server, which calculates moves and updates information and returns this information back to the Board of each player. This information is then displayed on the player's screen, even though it is never calculated by the Board class.

2.3.3: Final Domain Model

The Final Domain Model shows the class structure and design patterns by displaying all classes and how they connect to one another.

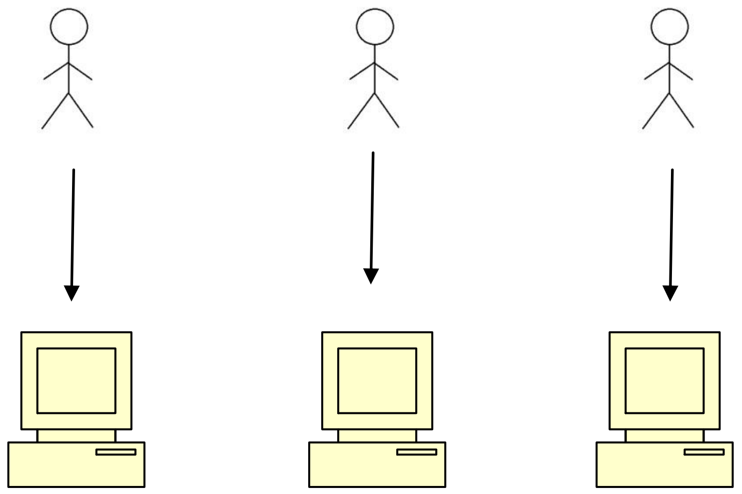


This diagram can be read as follows:

- Client and Server are associated, as the Client sends moves to the Server and the Server returns information to the Client
- Client holds one of each screen
- Each screen contains 2 MouseHitboxes
- GameScreen, the main screen of the actual game being player, contains 1 offense panel, and 2 instances of Board
- Board contains * units
- Unit contains * actions
- UnitFactory creates Units
- DefensiveUnit is a subclass of Unit
- DefensiveUnit aggregates one Upgrade
- Upgrade aggregates * Upgrade
- Upgrades are created by UpgradeFactory

2.3.4: Final Deployment Diagram

Our final deployment is extremely simple. Individual users will download and install the game on their individual computer (or computers). Each computer will have the full software installed, which includes the capability to host a server and connected to other hosted games as well as play single player against AI. All software is installed equally on all machines wishing to play the game.


Individual Users' Machines

These individual machines can connect to each other if they wish to play a networked multiplayer game. They can connect either over a local network or over HTTP (assuming they all have internet connections).

Actual deployment to individual machines is very straightforward. Users will unzip the conquertheseas.zip file to the location they choose. This .zip file contains all game files and an executable file that runs the game. This executable launches the full game regardless of the user's final intent - from this executable, the user can host a multiplayer game, join a multiplayer game, or play a single player game.

One the executable is run and the game begins, the user is presented with a menu with several choices. "New Game," "Load Game," "Options," "Credits," and "Exit" are the options presented on the main screen. Clicking "New Game" opens a submenu that allows the user to choose from "Single Player," "Host Multiplayer," and "Join Multiplayer," which are the three main choices of beginning a game. "Load Game" allows the user to load a previously saved single player game. "Options" opens a submenu with different preferences such as Multiplayer Name, AI Level, Sound Effects Volume, and Music Volume. "Credits" displays information about the development team of Conquer the Seas. "Exit" quits the game.

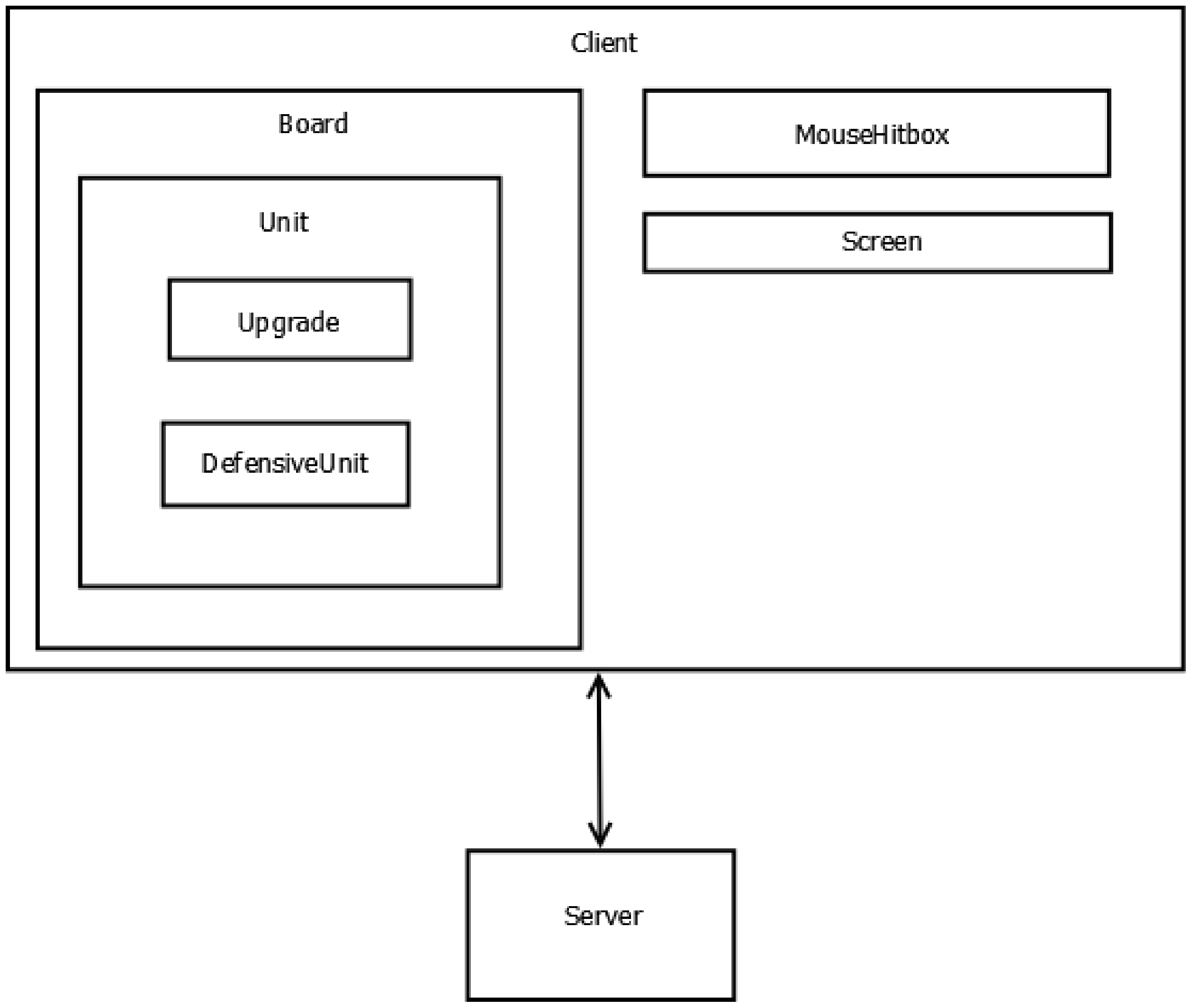
"Single Player" (from the "New Game" menu) begins a single player game against AI with the difficulty as specified in the "Options" submenu. "Host Multiplayer" launches a server, which begins listening for connections. It also places the user in the Lobby of this multiplayer game, with them as a client. "Join Multiplayer" opens a box to type in the IP address of the user's machine the multiplayer game server is being hosted on.

All of these options stem from running the executable file in the unzipped conquertheseas.zip file.

Section 3: The Specifications

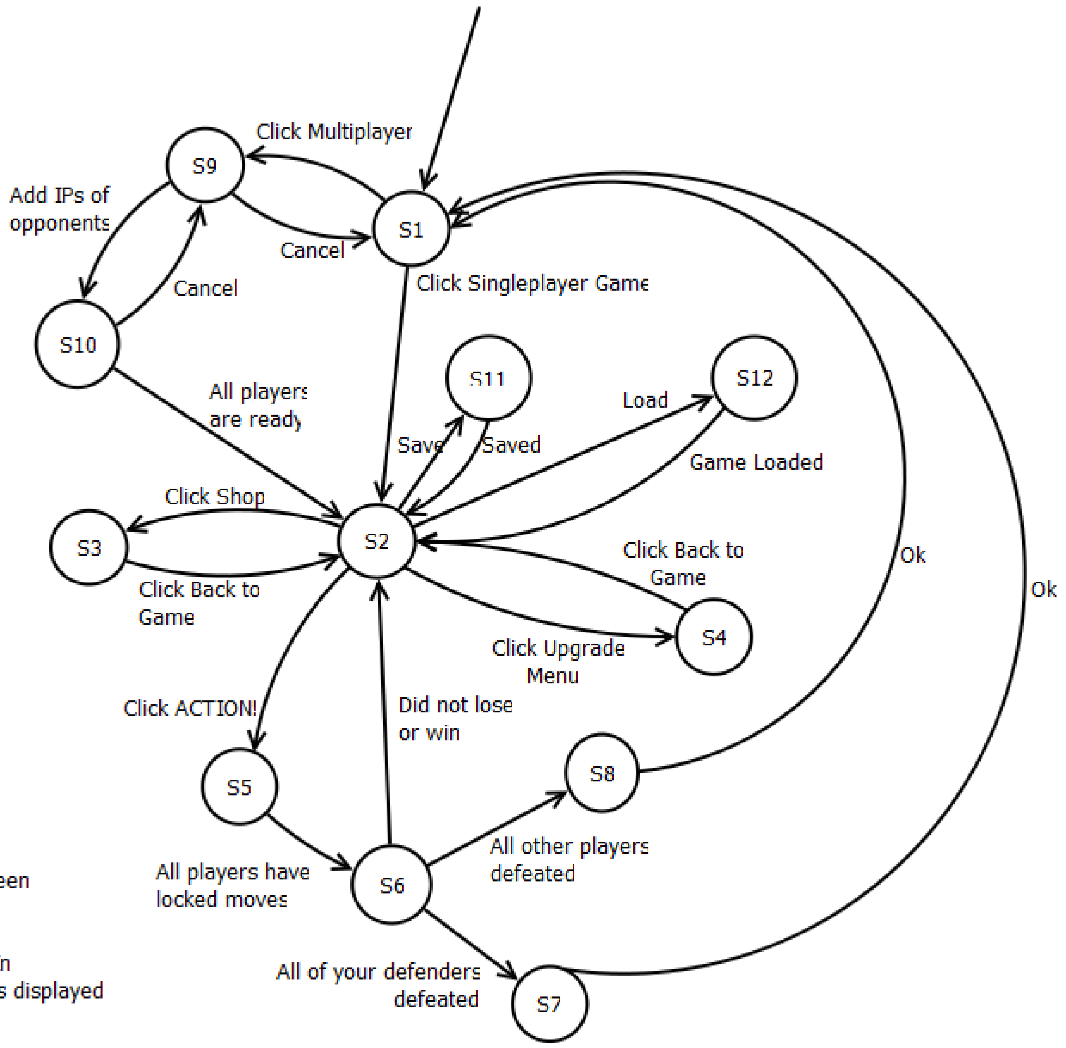
3.1: Software Organization

This diagram will show the software organization in a block diagram. Different modules will be grouped together based on common purposes. **Section 3.2** will show the specific decomposition of one module, the "clickable" module, which is a combination of MouseHitbox and Screen.



This block diagram shows the general organization of the Client-Server setup, including the Board module which contains most other functionality. The client module holds the Board, MouseHitbox, and Unit classes. Unit is contained within Board, as Units only appear instantiated within the Board class. MouseHitbox interacts with Screen to obtain the proper functions on clicks.

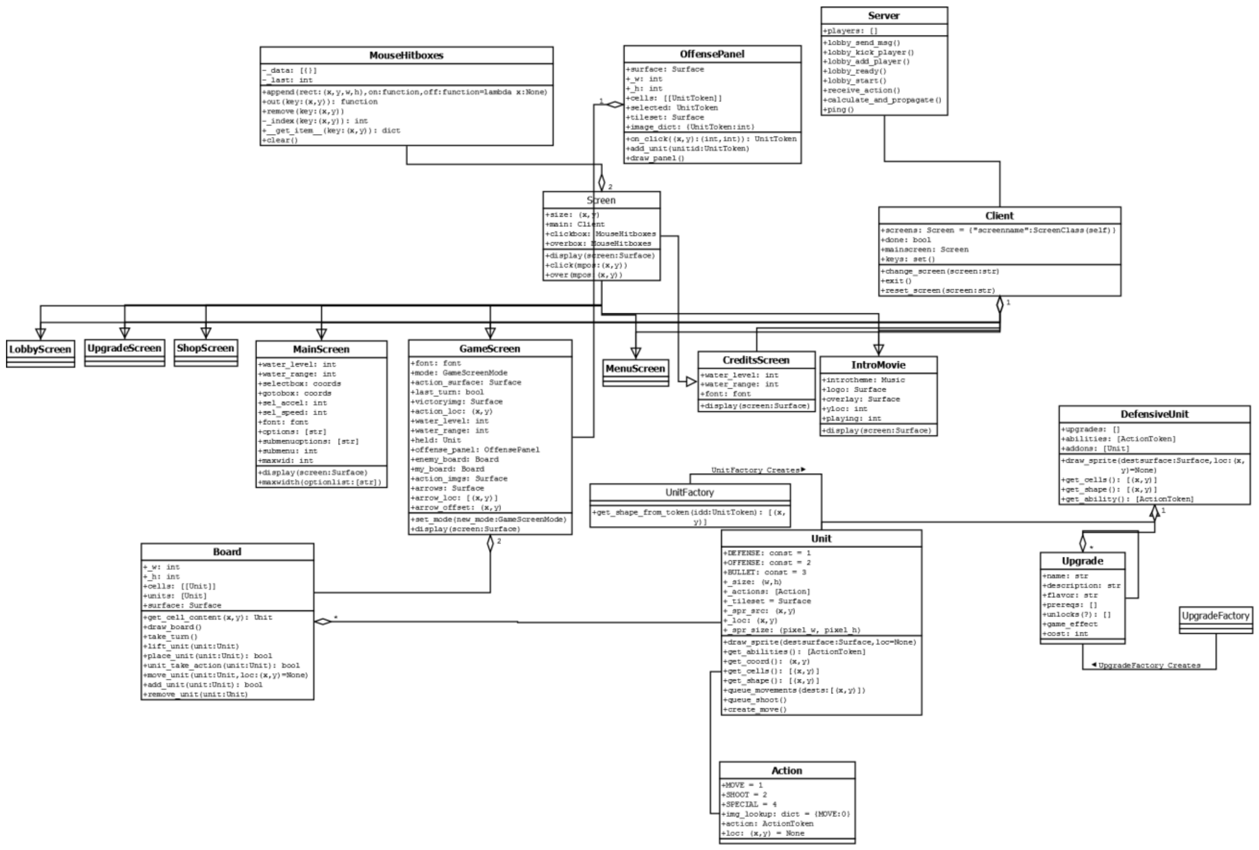
The following state diagram shows the flow of the program.



S1: Main Menu
S2: Main game screen
S3: Shop Menu
S4: Upgrade Menu
S5: Moves Locked In
S6: Result of moves displayed
S7: Loss Screen
S8: Win Screen
S9: Multiplayer Screen
S10: Waiting for players
S11: Save file
S12: Load file

This state diagram shows the transitions between all different states of the game.

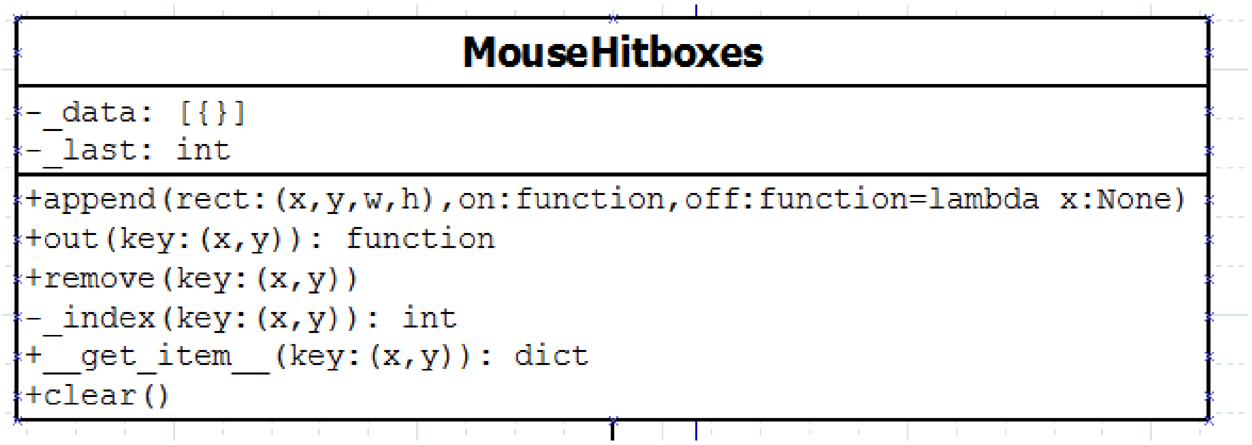
3.2: Module / Object Decomposition



PUT LINK TO WEB VERSION OF THIS

The module that will be fully explained is the "clickable" module, which is MouseHitbox and Screen. This module will be explained as we have and will continue to use MouseHitbox as an example.

FIX IMAGE



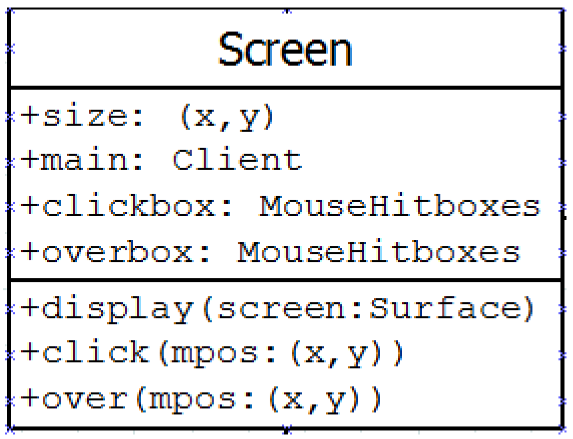
```
MouseHitboxes
-_data: [{}]
-_last: int
+append(rect:(x,y,w,h),on:function,off:function=lambda x:None)
+out(key:(x,y)): function
+remove(key:(x,y))
-_index(key:(x,y)): int
+__get_item__(key:(x,y)): dict
+clear()
```

Variables and methods in MouseHitboxes:

_data	A list of dictionaries which hold hitbox data, which is accessed later
_last	The index of the last hitbox that has been moused-over
+append(...)	Takes the rectangle for the hitbox location, a function for mouse-over or clicking, and a function for mousing out of the hitbox
+out(...)	WHAT DO HERE
+remove(...)	Removes the hitbox at the coordinate location
-_index(...)	Gets the index of the hitbox at mouse location
+__get_item__(...)	Returns the hitbox at the mouse location
+clear()	Empties all MouseHitboxes

CRC Card for MouseHitboxes:

CLASS **MouseHitboxes**
RESPONSIBILITY 1. Return which hitbox is being clicked on 2. Return which hitbox is being moused over 3. Returns which hitbox is being moused out of
COLLABORATORS 1. **Client** gives mouseEvents to **MouseHitboxes** 2. **Screens** instantiate and give functions to **MouseHitboxes** 3. Calls functions from **Screens** on mouseEvents from **Client**



Variables and methods in Screen:

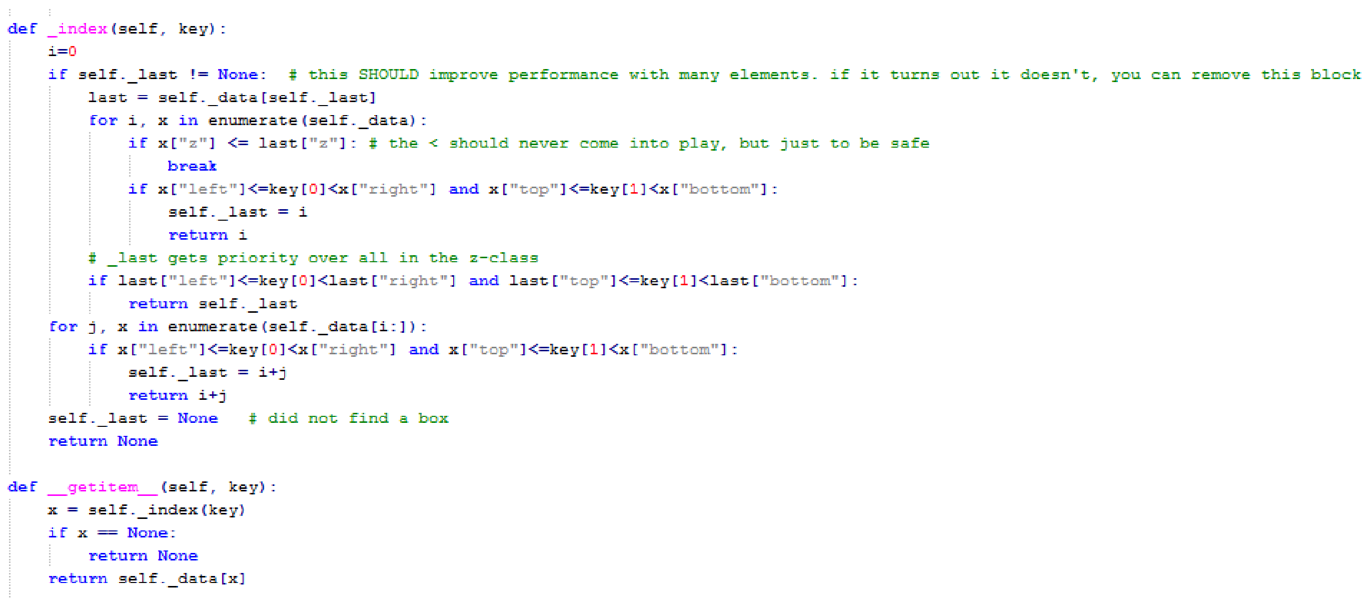
+size	
+main	
+clickbox	
+overbox	
+display(...)	
+click(...)	
+over(...)	

CRC Card for Screen:

CLASS **Screen**
RESPONSIBILITY 1. Handles click events by sending click location to **MouseHitboxes** 2. Handles mouse-over events by sending mouse location to **MouseHitboxes** 3. Draws screen background
COLLABORATORS 1. Subclasses **GameScreen, LobbyScreen, UpgradeScreen, MainScreen, MenuScreen, CreditScreen, IntroMovie** 2. Aggregates 2 **MouseHitboxes**

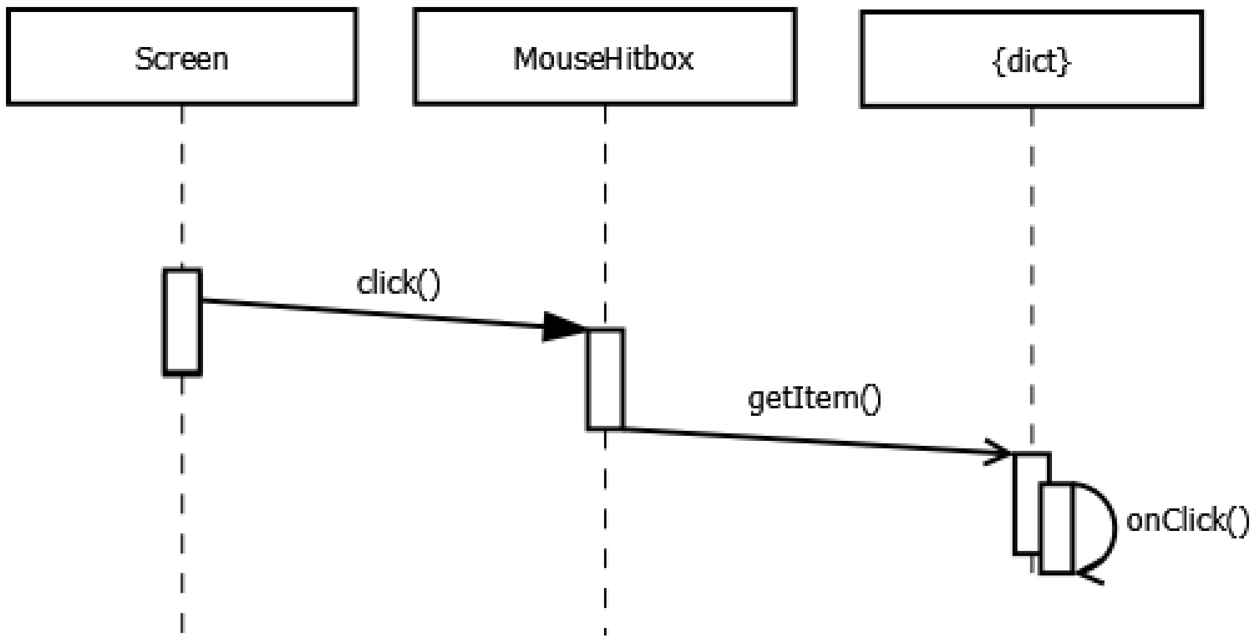
3.3: Critical Section Specification

This section outlines important aspects of code. The critical code that will be shown and explained is a the mousehitbox.py code which was used in **Section 3.2** and will continue to be used as an example through the rest of this document. The reason this code is being used for the critical section is because it is fully complete and should remain largely unchanged for the rest of development and is a good example of code that is critical to the game: having the mouse be able to click on different locations and having them respond appropriately.



```
def _index(self, key):
    i=0
    if self._last != None:  # this SHOULD improve performance with many elements. if it turns out it doesn't, you can remove this block
        last = self._data[self._last]
        for i, x in enumerate(self._data):
            if x["z"] <= last["z"]: # the < should never come into play, but just to be safe
                break
            if x["left"]<=key[0]<x["right"] and x["top"]<=key[1]<x["bottom"]:
                self._last = i
                return i
        # _last gets priority over all in the z-class
        if last["left"]<=key[0]<last["right"] and last["top"]<=key[1]<last["bottom"]:
            return self._last
    for j, x in enumerate(self._data[i:]):
        if x["left"]<=key[0]<x["right"] and x["top"]<=key[1]<x["bottom"]:
            self._last = i+j
            return i+j
    self._last = None   # did not find a box
    return None

def __getitem__(self, key):
    x = self._index(key)
    if x == None:
        return None
    return self._data[x]
```

These sections are critical to the code. The first method, _index, keeps track of the x,y pair of pixels on the screen of the current position of the mouse. This is used for both mouseover and for on clicks. The method __getitem__ returns the item whose hitbox the mouse is currently over or clicking on. The UML calling sequence diagram for this critical section is shown:



Screen sends a click to MouseHitbox, which uses this to call getItem(). getItem() returns a dictionary which contains the clicked item's onClick function, which is then executed. This function only exists in MouseHitbox's dictionary. This explains how each click translates into a function being executed (assuming something was clicked on). If nothing is clicked on, MouseHitbox returns None if there was no hitbox at the location of the click (or mouse-over, or mouse-out).

Section 4: Programming Management

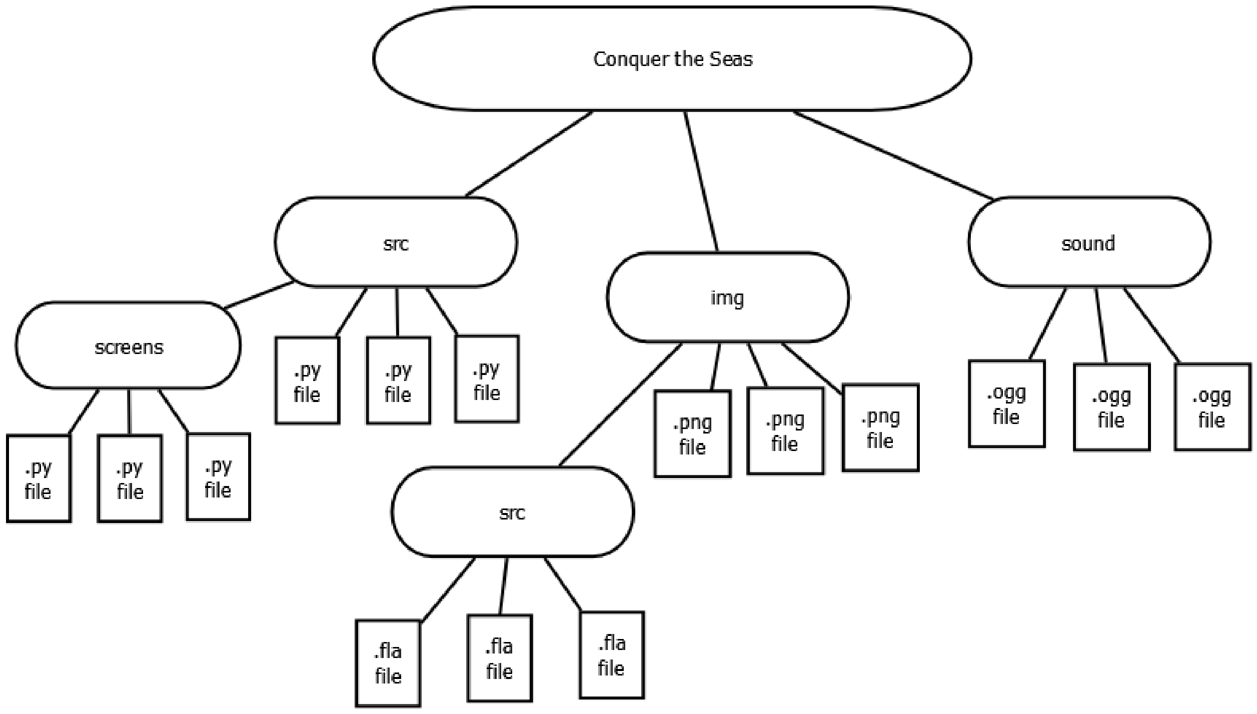
4.1: Programming Management Overview

Section 4 will deal with explaining the tools and processes used in the actual programming of Conquer the Seas. It will explain the different tools used and why they were chosen. It will also describe the software building method we used (Incremental) in more detail, and how revision history is managed.

4.2: Directory Structure and Programming Tools

This section describes the organization of the directory used for the game, and the tools used to write and manage code.

4.2.1: Directory Structure



This diagram shows the directory structure of Conquer the Seas. In the main folder, named Conquer the Seas, there are three subfolders, as follows:

The first subfolder, src, has the python source files of the game. These are the files that contain the game logic, networking code, and the rest of the source code. There is also a subfolder inside src, named screens, which contains the python files for each of the game screens.

The second subfolder inside the main folder is img, which contains the .png image files used in the game. This subfolder also contains a subfolder, src, which contains .fla flash files used for menus and animations.

The third subfolder, sound, contains the .ogg files that are the sound effects and music in the game. All of these sound files are recorded in-house by the LIFDOFF team.

Saved games can be stored anywhere on the user's machine, and be loaded from any location on the user's machine as well.

4.2.2: Programming Tools

While writing the code for Conquer the Seas, several different tools have been used, mainly different text editors and an online code repository. The first text editor, used mainly by one member of the team, is jEdit. jEdit is a text editor for programming that has many useful features, including customizable syntax highlighting (built in for python). The second text editor, used by one member who uses Mac OS X, is Vim. Vim is a vi clone with some additional features and again has syntax highlighting and other conveniences. The third text editor is Notepad++, used by the final member of the team. Much like the other two, it is only a choice of familiarity, and has similar features to the first two. More information about these text editors can be found in the **Appendix** in **Section 6.**

The only code repository used for this project was github, an only code repository and version control system that uses git. It is free for open-source projects and had many features that were useful to the LIFDOFF team during work on Conquer the Seas. Because it uses git, there are easy to use clients available for any operating system that the team is using, which is convenient as one member uses Mac OS X while two members use Windows 7. github also makes it easy to browse and edit our code from any machine with a web browser, alleviating the need to have code hosted on any one specific machine. For more information about github, see the **Appendix** in **Section 6.**

4.3: Software Building Method

As stated in our Preliminary Design Document, the method of programming development that we have been using has been incremental. Following this method, we first developed a single player prototype with a simple board and moveable pieces. The next step was to create a second board, with the capability to send units to the other board. The third module was to add multiplayer support to a single computer, then, to add single player versus AI to that single computer. The final necessary module is to add networked multiplayer. Additional modules could be completed after this point, such as improved graphics, dynamically generated terrain, and expansive upgrade trees.

4.3.1: Revision History

By using github as an online code repository and version control system, the need to keep track of all revisions is taken care of automatically. Through github, we can always revert back to a previous version of code or of a document. This made our revision history a non-issue. Again, see the **Appendix** in **Section 6** for more information about github.

4.4: Coding Agreement

The coding style and practices agreed on by the team were largely unimportant. Because python is being used, the style of the code is already very clean and neat without much room for customization. Whitespace indentation is used to delimit blocks of code, so unlike braces, which allow for variation in placement (using whitespace), there is no customizability. For this project, soft tabs composed of 4 spaces makes up the whitespace used to delimit blocks. Python code is very readable, but it was agreed upon to place as many comments as necessary to understand the code.

4.5: Migration Procedures

As all code is being created (aside from the modules given by pygame) from scratch for this project, there was no code to migrate at the beginning of or during this project. As we are not using a database for any storage, all data is maintained locally and simply in the directory of the game. Because of these reasons, no data should ever have to be migrated to any other location or machine apart from the machine the game is installed on.

4.6: Installation Procedures

Conquer the Seas will not require installation, merely unzipping the folder to the location the user wishes the game to reside in on their machine. Running the game file will run the game, and a shortcut to this file can be created and placed anywhere on the user's machine for easy access. TALK ABOUT EXE, ETC

Section 5: Test Cases and Procedures

5.1: Test Cases and Procedures Overview

This section of the Design Document will outline the test cases and procedures used during the continued programming of Conquer the Seas. The following subsections will describe how test-driven development was used, the different methods of error logging that were used, and how classes were designed with testing in mind.

5.2: Procedures

This section details the different procedures used for testing of Conquer the Seas. It outlines the scope and depth of different testing procedures.

5.2.1: Testing Scope and Depth

Testing scope and depth:

- Most simple functions have unit testing
- Our unit tests check that the functions produce the correct results, which is functional testing
- Integration testing occurs before each commit to the git repo

The scope and depth use three different types of testing: Unit, Functional, and Integration. The reasoning for each of these:

- Unit testing: this allows for very easy automated testing, entire classes can be written to test modules
- Functional testing: this allows us to make sure that changes to a class do not cause any of its functionality to change
- Integration testing: this makes sure that one programmer's changes do not break any previously written code

On top of these testing method, user-interface and feature testing occurs periodically through development. Any actual errors or bugs should be caught by the other testing methods, but this will pick up on visual imperfections, ease-of-use problems, and common sense errors. An additional way to test user-interface and feature testing is to bring in users who are not on the development team to try to software. This allows for a fresh perspective, and often times can point out flaws that the development team has become used to or not realized.

Other types of testing techniques will be used later on in development. Configuration testing will begin once the software is complete and working on one operating system. Performance testing will not be formalized until software is complete as well. Optimization can only occur once all facets of the game are set, we feel that changing things for performance reasons before fully understanding the performance of the software would be premature. Stress testing will be performed periodically, and once the multiplayer component is finalized, testing how many simultaneous connections can occur will begin.

5.2.2: Testing Method and Agreement

Testing occurs before each commit, or whenever a new feature is added. After making sure all unit tests are still passed after any changes are made, updated code can be committed and pushed. Programmers agree that all tests will be completed before committing and pushing, such that no errors remain. This keeps errors from bouncing to different programmers. If code needs to be pushed that still contains errors, such as in the case that a programmer cannot fix code to the point where all test cases are passed, a note must be made so that this can be fixed immediately.

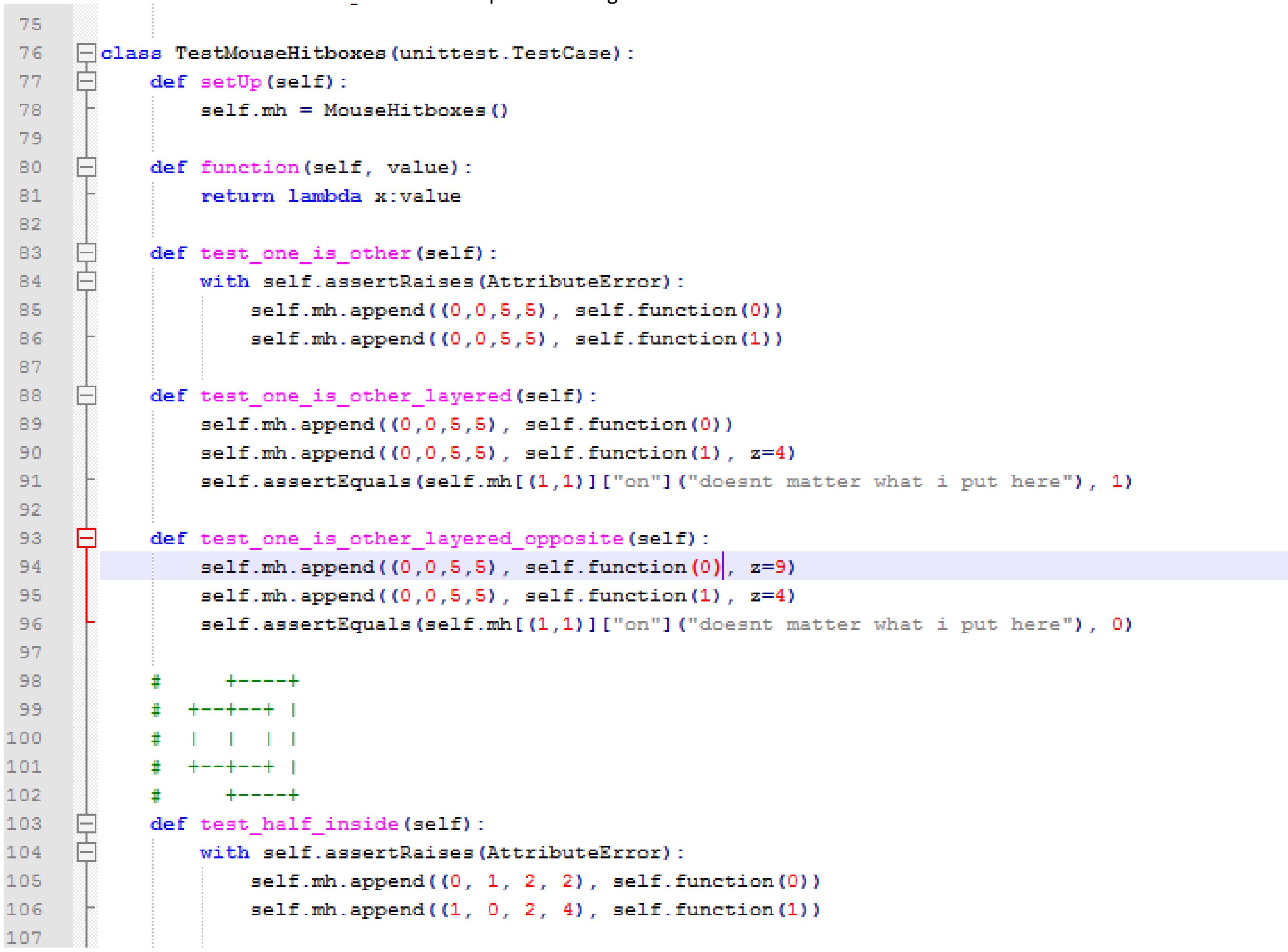
Our automatic testing procedures applies to unit tests built into each class. Double-clicking on a python file in the src code will run all unit tests for the file. For example, double-clicking on mousehitbox.py runs the unit tests in the class TestMouseHitboxes. This makes testing very easy after any modifications to the code.

5.3: Test Cases

Strict test-driven development requires first designing test cases that fail, and then writing code to pass the test cases. We chose not to follow this strictly, as we felt we needed a working framework to build more specific test cases from. In writing the code for Conquer the Seas, the first priority was to get certain modules functional before testing. For example, a display of the game board was first created before testing any qualities of the board, such as mouse hit boxes, placing units, etc. Writing test cases prior to even understanding how certain code would function did not make sense for code starting from the ground up.

We opted to create testing classes inside each module after the basic framework had been laid down. We used three types of testing in our code, and one outside. A detailed description of each type will be given for one module, mousehitbox.py. Test cases were designed using a mix of intuition and black-box test cases. In this case, it was to make sure that all hitboxes in the game are functioning properly - how to react when two hitboxes are overlapping, on top of one another, crossed over each other, or other scenarios.

Example of Testing Class

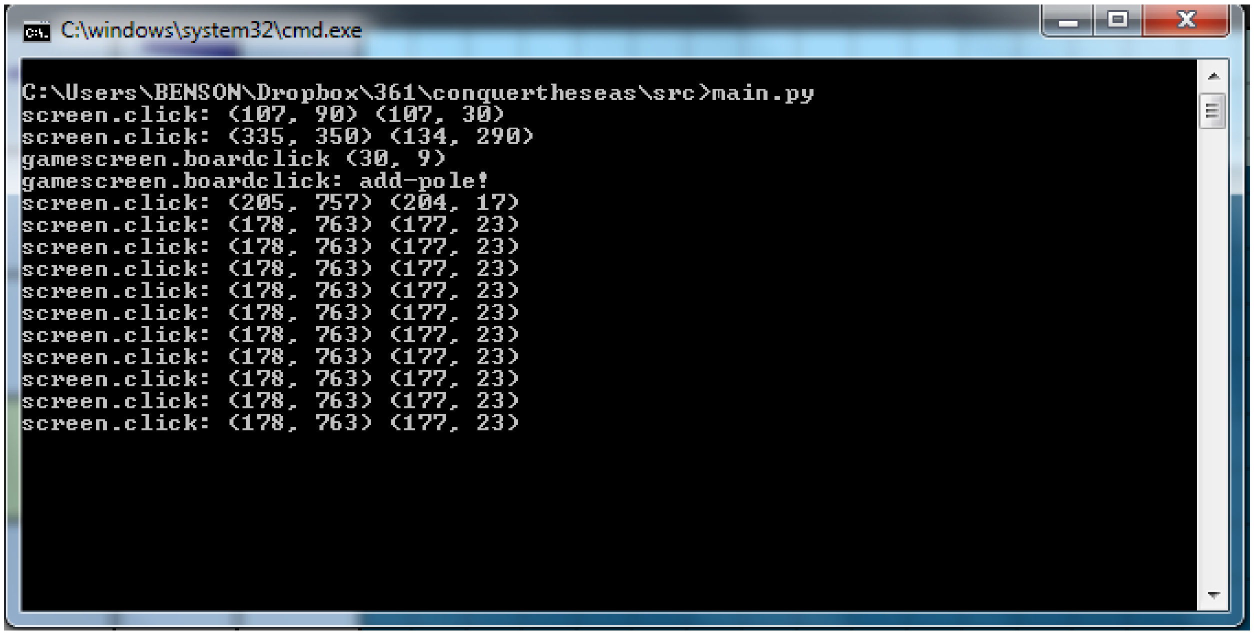


```

class TestMouseHitboxes(unittest.TestCase):
    def setUp(self):
        self.mh = MouseHitboxes()

    def function(self, value):
        return lambda x:value

    def test_one_is_other(self):
        with self.assertRaises(AttributeError):
            self.mh.append((0,0,5,5), self.function(0))
            self.mh.append((0,0,5,5), self.function(1))

    def test_one_is_other_layered(self):
        self.mh.append((0,0,5,5), self.function(0))
        self.mh.append((0,0,5,5), self.function(1), z=4)
        self.assertEquals(self.mh[(1,1)]["on"]("doesnt matter what i put here"), 1)

    def test_one_is_other_layered_opposite(self):
        self.mh.append((0,0,5,5), self.function(0), z=9)
        self.mh.append((0,0,5,5), self.function(1), z=4)
        self.assertEquals(self.mh[(1,1)]["on"]("doesnt matter what i put here"), 0)

    #     +----+
    #  +--+--+ |
    #  |  |  | |
    #  +--+--+ |
    #     +----+
    def test_half_inside(self):
        with self.assertRaises(AttributeError):
            self.mh.append((0, 1, 2, 2), self.function(0))
            self.mh.append((1, 0, 2, 4), self.function(1))

```

- Unit testing: Unit testing is easily implemented in TestMouseHitboxes by using python's unittest package. By calling unittest.main(), all subclasses of unittest.TestCase in the file are called. Then, setUp is called, followed by the first method that begins with test_, then setUp again, and so on, until all test methods are called.
- Functional testing: this is built in to the unit testing. The unit tests test the functionality of each class, for example, in this class, they test to make sure all possible cases of hitbox overlaps perform correctly.
- Integration testing: this is not built directly into the code. Rather, after any code is modified, all classes can quickly be run using the method described above. If all tests are passed, code can be pushed and committed. This prevents the case where one user pulls from the repo, finds lots of errors, and doesn't know what has been changed to cause these errors. As stated above, any failed tests that must be pushed for extraneous circumstances must be accompanied by detailed descriptions of the problems and the steps that need to be taken to fix them.

5.4: Error Logging

Currently, errors are printed to the command line (stdout). This allows for quick viewing of errors, and allows for debugging using print statements. While this is a good process to use during earlier development, as the game nears completion, it will make more sense to begin writing errors to a log file.



```
C:\Users\BENSON\Dropbox\361\conquertheseas\src>main.py
screen.click: (107, 90) (107, 30)
screen.click: (335, 350) (134, 290)
gamescreen.boardclick (30, 9)
gamescreen.boardclick: add-pole!
screen.click: (205, 757) (204, 17)
screen.click: (178, 763) (177, 23)
screen.click: (178, 763) (177, 23)
screen.click: (178, 763) (177, 23)
screen.click: (178, 763) (177, 23)
screen.click: (178, 763) (177, 23)
screen.click: (178, 763) (177, 23)
screen.click: (178, 763) (177, 23)
screen.click: (178, 763) (177, 23)
screen.click: (178, 763) (177, 23)
screen.click: (178, 763) (177, 23)
```

As it stands currently, writing to a log file would be too time-consuming. Opening up a file to check errors and other debug print statements would not be appropriate at this point in development.

Section 6: Appendix

6.1: Further Info

- Vim: http://www.Vim.org/
- jEdit: http://jEdit.org/
- Notepad++: http://notepad-plus-plus.org/
- github: https://github.com/
- python: http://python.org/
- pygame: http://pygame.org/news.html