|  |
| --- |
| [Type the company name] |
| CS4227 2015 Project |
| Maze Game Framework |
|  |
|  |
|  |

Michael Hallinan 12134635

Emmylou Flores 12132403

Owen Ryan 12128015

Ryan Murphy 12122688

|  |
| --- |
|  |

Table of Contents

**Type chapter title (level 1)1**

Type chapter title (level 2)2

Type chapter title (level 3)3

**Type chapter title (level 1)4**

Type chapter title (level 2)5

Type chapter title (level 3)6

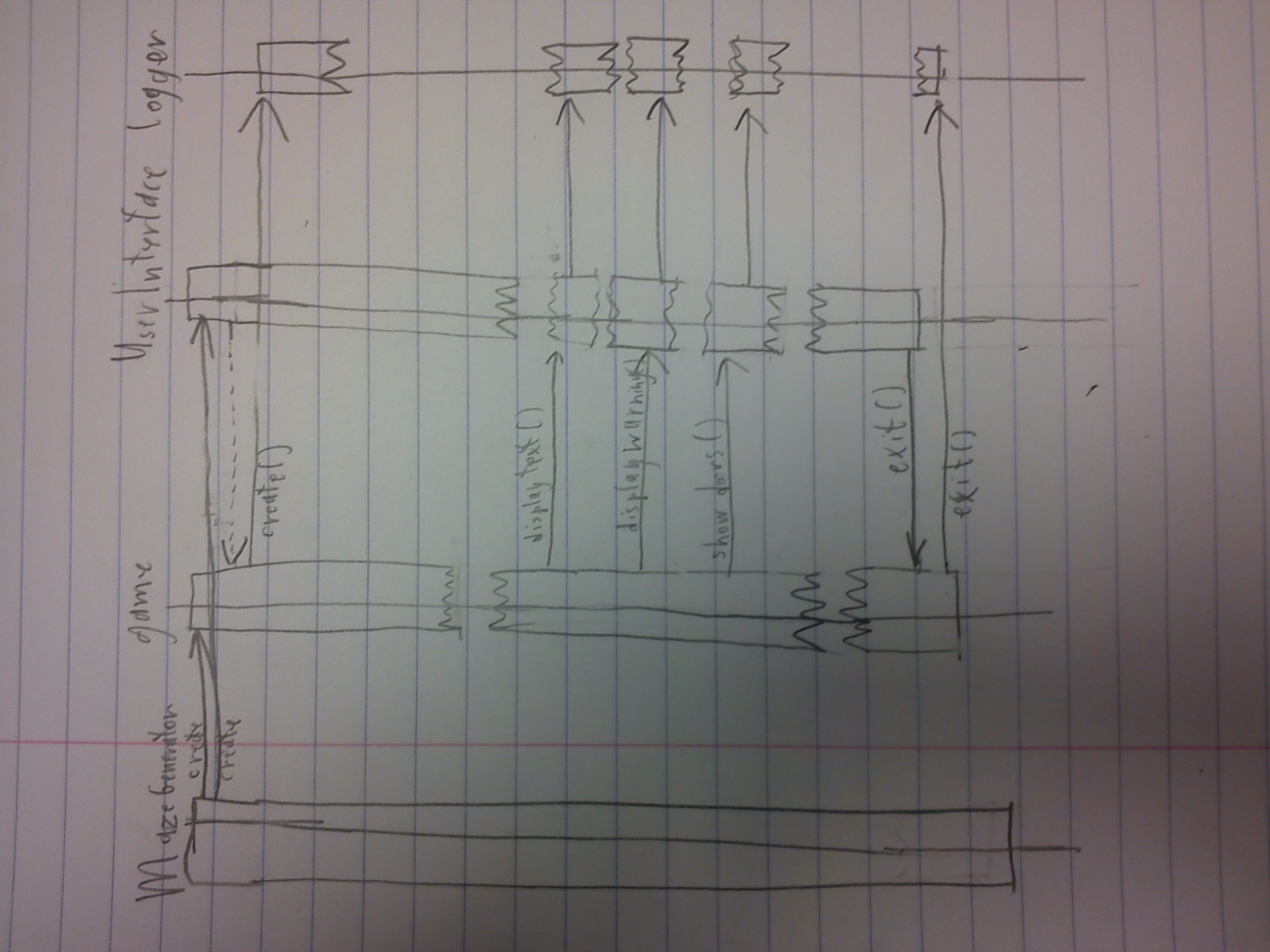
***Requirements***

***Scenario***

The Scenario for this project is that we have been asked to develop a basic framework for the generation of a game and game world. The world should consist of a series of connected rooms, that the player is able to move between. Each room should also have object within them that the player can interact with. Furthermore there should be support for the inclusion of Non-Player-Characters with options for third party clients to develop their own artificial intelligence systems for them.

# Structural Diagram

# Runtime Diagram of Interceptor



# Discussion of Architectural and Design Patterns

This section lists the various design patterns being used in the project, where they are being used and why they are being used.

## Abstract Factory

The Abstract Factory is being used for the construction of Rooms, in this way it is easy to extend the game with the introduction of new types of rooms while maintaining encapsulation by distancing the client from the actual instantiation of the Rooms.

## Decorator

Is being used to create items that would be found within rooms, it is not being used to add the items to rooms rather it is being used to create the items themselves. By using the decorator design pattern we can allow the creation of a plethora of items from a smaller pool of methods, e.g. we can use “makeBroken(makeGolden(makeChair())))” , “makeBroken(makeChair())” , “makeGolden(makeChair())”, “makeChair()” to create 4 different objects using only 3 methods, rather than having to create 4 separate classes.

## Observer

The Observer is being used to notify NPC’s as to when and where to the player moves, with the Player class being the subject and the NPC’s being observers. In this way NPC’s will be aware of the Player’s current position within the game, allowing 3rd party clients to create AI for npc’s that takes the player’s position into account.

## Strategy

The Strategy pattern is being used to allow for the AI of NPC’s, it will allow 3rd party developers to easily develop alternative AI programs for NPC and swap them out with easy, thus supporting extensibility.

## Façade

The Façade is being used to simplify the creation of the game world, as one class will be provided at the front end that will provide one method that calls all the commands for the creation of rooms(through the abstract factory), the creation of items(through use of the decorator) and the adding of items to rooms. This will make it far easier for 3rd party developers who simply want to create a replace GUI for the game rather than modify the game itself, as it will hide all the game’s construction behind 1 method in the Façade Class. This is the Pattern we selected through self-guided research.

## Interceptor

The Interceptor is being deployed in-between the game and the GUI, this is in order to provide 3rd party developers with hooks between the User-experience and the game program itself. These hooks will allow them to create interceptors to extend or listen for events and use these in their own implementation of the GUI.

***Performance Engineering***

Performance Engineering is supported in our project through the use of multi-threading within the observer design pattern. Specifically it is used to multi-thread the notification of all Observers by the subject. To achieve this a sequentialNotifyAll() method was included in the Player class that called notifyThisObserver() on each Observer, alongside a threadedNotifyAll() method which created one thread for each observer which then called notify() on its observer.

Coding Fragment:

Within Player.java:

public void threadedNotifyAll()// a multithreaded version of notifyAll

{

for(int i = 0; i < myObservers.size(); i++)

{

Thread athread = new Thread(new RunnableNotify(myObservers.getElementAt(i), "Notifier Thread " + i));

//notifyThisObserver is called inside run() of RunnableNotify()

athread.start();

}

}

RunnableNotify.java:

package characterStructure;

public class RunnableNotify implements Runnable

{

private ObserverOfCharacter observerToNotify;

private String name;

public RunnableNotify(ObserverOfCharacter observerIn, String inName)

{

observerToNotify = observerIn;

name = inName;

}

public void run()

{

observerToNotify.notifyThisObserver();

System.out.println(this.name + " has completed.");

}

public String getRunnableName()

{

return this.name;

}

}

As our system was intended as a framework for a game we used simple tests to simulate process’s being executed by the Observers in response to being notified. After running the system using both methods the run time’s for each of these were:

SequencialNotifyAll(): 1000 Millisecond

ThreadedNotifyAll(): 200 Millisecond

However without any such processing by the Observers in response to being notified, the results were:

SequencialNotifyAll(): 1 Millisecond

ThreadedNotifyAll(): 8 Millisecond

Thus we concluded that although in our implementation of the code, the extra overhead of threading every notify() only reduced speed, but in the interest of supporting extensibility, clients would undoubtedly extend the NPC’s that would serve as observers, and thus add extra processing to when these classes are notified thus warranting the inclusion of this multi-threaded NotifyAll() in our implementation of the Observer Design Pattern.

# References