Iteration 2 Design Documentation

Nathan Samuelson

April 10, 2018

1 Introduction

This design document was created with the intent of describing two different design patterns for the robot simulation. The two patterns discussed will be the Observer pattern and the Strategy pattern. There are two options given for each of the designs. The goal of analyzing these design patterns is to conclude which has the best application within the robot simulation. The remaining portion of the document includes a tutorial on feature enhancement which describes the process of adding more stimuli and their respective sensors to the simulation. There are various robot reactions to lights and food depending on the type of robot and the hunger level at the time. [H]

2 Observer Pattern

NOTE: This section has a glaring misinterpretation for the intention of the original design document. I had previously thought that we were considering an alternative design pattern altogether. However, I did not receive grading for my initial design document after submission – where I would have seen this error. As a result, I have left the error in the interest of time. Professor Wendt has been quoted stating these errors will not be double docked for both initial submission and this draft.

2.1 Introduction

In this section the two potential Observer patterns are discussed before concluding on the best implementation for the robot simulation.

2.2 First Implementation Idea

The Observer pattern represents a one-to-many dependency between a number of objects that ensures when one object changes its state, all of its dependents are automatically notified and updated. When an observer is created, it is registered with the subject with which it will be associated. From this point on, whenever the subject changes it broadcasts these changes to the registered observers so that they can be updated. This is the "push" model where the subject updates the observers as necessary. The alternative is the "pull" method where observers consistently poll the subject for relevant information. This is less efficient as it must constantly be receiving updates even when they are not necessary. The "push" model of the observer pattern would be a good choice for our project as in certain wheel and sensor configurations there will be frequent occurrences where a robot is sitting still. Therefore, it would be a waste of computational efforts for the sensor to constantly request updates from the subject or robot. Rather, the robot would update its sensors only when its position changes.

2.3 Alternative Implementation

On the other hand, the Memento design pattern allows for unique functionality in that the user can undo or redo state changes through the use of an external "memento." This memento maintains a history of the subject's states so that as needed, the subject can be returned to a previous state. In the case of a video game, this would be a "checkpoint" or a save game feature which could later be returned to. Another example would be the ability to undo and redo actions in a word processor. In this design pattern, the observer is

the "care-taker" of the memento, but only the subject can store and retrieve information form the memento. When a rollback (or undo) is required, the memento is handed back to the subject so that the designated state can be returned to. If an infinite history of states is required, then it can be implemented in the form of an ever growing stack. In the case of our project, the robot is the subject which the states are generated from and some other construct would represent the caretaker. However, this design pattern does not focus on communication between the robot and sensor so it would not be well suited for the relationship.

2.4 Design Chosen

Given these considerations, the obvious choice becomes the observer pattern implemented with the "push" method to ensure efficiency. As a result, the sensors will be updated only when the position or rotation of the robot changes. There is still opportunity to implement a form of the memento design pattern in later project iterations but it will not be necessary at this point in time.

3 Strategy Pattern (*Incomplete)

3.1 Introduction

3.2 Strategy Pattern

3.3 First Implementation Idea

3.4 Alternative Implementation

3.5 Design Chosen

4 Feature Enhancement

4.1 Introduction

In this section, the scenario in which a new developer team would like to add further stimuli and sensors for that stimuli is considered. The process that this team would need to take is detailed with the use of code snippets as applicable.

4.2 Scenario

The scenario to be considered will be the addition of a fire stimuli and new sensors for fire (heat sensors).

4.3 Tutorial

The following tutorial will detail the exact process required to add the fire class and sensors.

4.3.1 Adding the Stimulus

Create the class files for fire. This will inherit from the ArenaImmobileEntity class. The header will include a member variable which determines whether or not the fire object is currently lit (lit_). Another variable will be included as a timer for the duration that the fire is lit (lit_time_). There will also be methods to reset the class, extinguish the fire, light the fire, and set the duration for lit_time_. The next step will be to add an entity type, namely kFire. This will be as simply as adding the name to the entity_type.h enumeration. The third step is to add a CreateFire() method to the EntityFactory class. After this, one can add a pointer to the Arena class for fire_ which will be used as a placeholder for the various fire objects. Fire sensors will be added as member variables of a robot which inherit from sensor.h. In Arena, all robots will be notified of all fire object positions. This will update the robots fire sensor readings. Finally, the fire sensors must be drawn in the GraphicsArenaViewer.