Name: Quang Phung

Course: CSCI 3081

Instructor: Amy Larson

Date: 04/09/18

DESIGN DOCUMENT

Project: Robot simulation of Braitenberg vehicles

Table of contents:

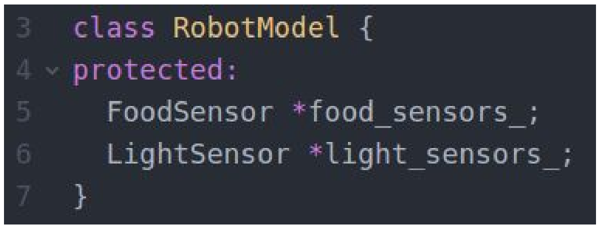
1.a. <u>Design decision:</u> Observer pattern

There are different ways to share information with sensors. The first approach is to use the Observer Pattern in which sensors are observers and stimulus are subjects. Each stimuli will have a list of sensors in the arena. Whenever a stimuli updates its status, it will notify all sensors to the arena about its position, type, etc. Each sensor then decides what to do with the information it just received. To implement this, two new classes - Subject and Observer - will be added. The Subject class will have a list of Observer objects as its member. It also provides a method to register new observer, a method to remove an unwanted observer, and a method to notify all of its observers when a status gets updated. The Observer class will have a method to manipulate the data it has received from the subject. Then, the programmer will create a new class for the new stimuli and allow it to inherit from both the ArenaMobileEntity (or ArenaImmobileEntity depends the motion behavior of the new stimuli object) and the Subject class. The sensor class in turn will inherit from the Observer class because sensors keep waiting for the entities' signals to take the next action. One modification to the Observer Pattern is that the Sensor class now has a list of stimulus which it wants to register to in the arena. As the result, every stimuli in the arena will be able to notify all sensors when it updates its status, e.g. position, size, type, etc. One advantage of this approach is that it creates a convenience for a programmer if they want to add a new kind of sensor such as light sensor. All they need to do is to create a corresponding class for the new sensor and let it inherit from the sensor class. Since the Sensor class has a list of stimulus in the arena, the new type of sensor class can reuse this data.

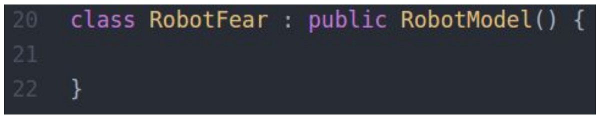
Another approach is to let the arena notify all sensors at once. After advancing the time for all entities, the arena will send the information of each entity to all sensors. To implement this, besides a list of entities, the arena should also have a list of sensors to which it will notify. This can be accomplished by adding a new member, e.g. sensor_list_, in the Arena class. The Sensor class also has a reference to the arena so that it can register to get the updated information. This approach does not use the Observer Pattern so no new classes need to be added. When a programmer wants to add a new type of sensor, they only need to create a new class and make it inherit from the sensor class.

1.b. Design decision: Strategy pattern

The different models of the robot (e.g. Fear, Aggressive, Love, Exploratory) in the arena provide a good condition to apply the strategy pattern. The idea is that we would create a new general class for different models of robot called RobotModel class. This class will have all information about how many sensors the robot has, including light sensors and food sensors, how sensors are connecting to both wheels (e.g. direct connection or crossed connection), how sensors impact the velocity of both wheels (e.g. positive connection or negative connection). After creating this new class, the robot class now would have an additional member which is a pointer to the RobotModel class. This member helps to determine what kind of model the robot is simulating. For each model of the Braitenberg vehicles, we will add a new class and let it inherit from the RobotModel class. This new class connects all sensors properly to both wheels to reflect the correction motion of this model based on the Braitenberg vehicles. When we want to create a particular robot, for example a robot exhibits Fear behavior, we simply need to initialize the pointer to the RobotModel class.

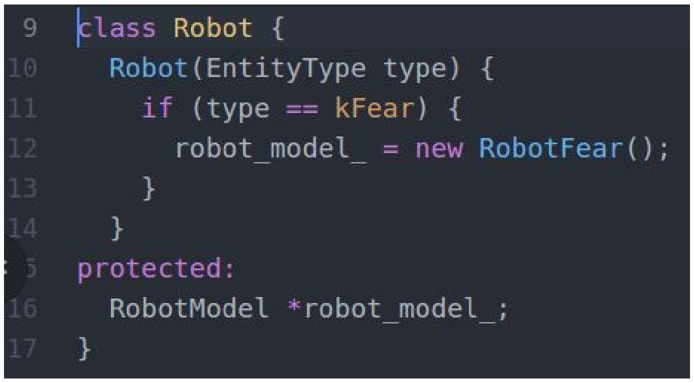


```
class RobotModel {
protected:
  FoodSensor *food_sensors_;
  LightSensor *light_sensors_;
}
```



```
class RobotFear : public RobotModel() {

}
```



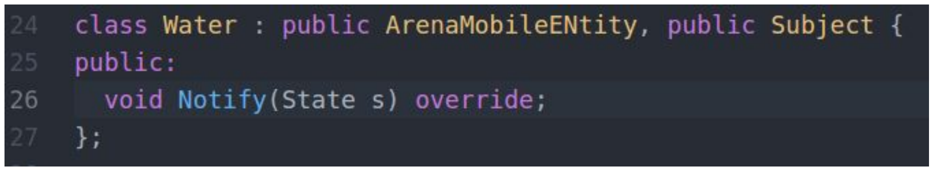
```
class Robot {
  Robot(EntityType type) {
    if (type == kFear) {
      robot_model_ = new RobotFear();
    }
  }
protected:
  RobotModel *robot_model_;
}
```

There are many advantages of this approach. First, a programmer can add as many new models as they want for the robot in the future. All they need to do is to create a new class for that model and make it inherit from the RobotModel class. They also need to initialize the robot_model_ pointer to the new added class in the Robot constructor. Another advantage is that we can change the model of any robot easily at any time in the simulation. For example, when the robot is very hungry, it should exhibit aggressive behavior to the food. We can implement this by reinitialize the robot_model_ pointer to the RobotAggressive class.

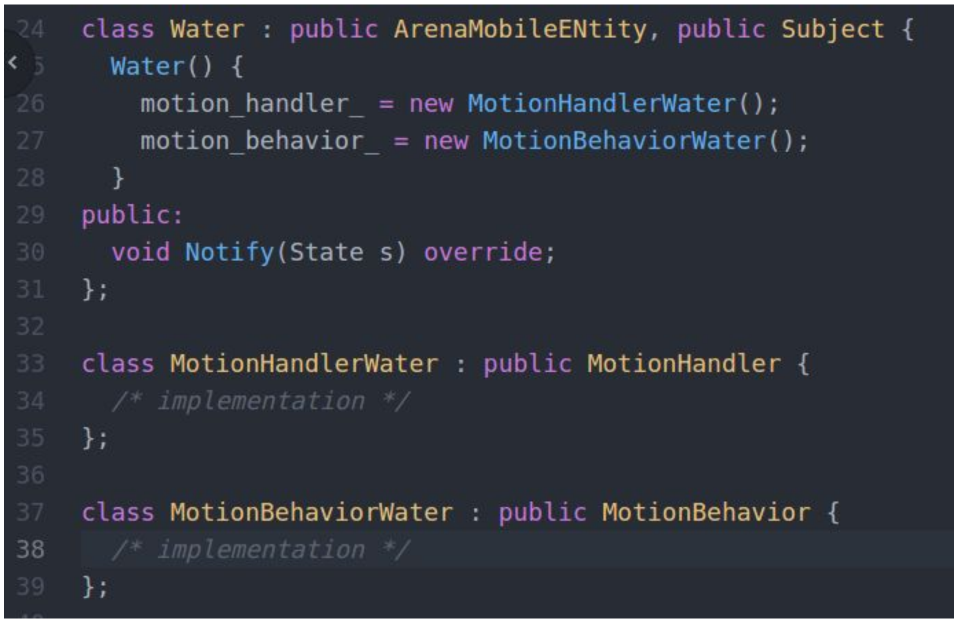
Another approach that we can do is to create different kinds of robot by adding new classes for FearRobot, AggressiveRobot, LoveRobot, ExploreRobot and make all of them inherit from the Robot class. Doing it this way, however, requires us to override all needed fuction in the Robot class which causes redundant codes and makes the program look messy. Furthermore, this approach does not provide a way to change the model of the robot, e.g. from Fear to Aggressive, as we have to create the branch new robot and copy all information from the old one to the new one (e.g. position, hungry time, velocity, sensor readings, etc.)

2. <u>Tutorial Outline</u>

In order to add a new stimuli to the simulator, the programmer needs to create a new class for that stimuli and let it inherit from both the ArenaMobileEntity class (if it is a mobile stimuli) or the ArenaImmobileEntity class (if it is an immobile stimuli) and the Subject class which is part of a Observer Pattern. The programmer also needs to implement all method from the Subject class because the Subject class only acts as an interface. This will include overriding the Notify method which allows it to send its information (e.g. position) to all of its observers (e.g. water sensors). If it is a mobile entity and it has a different motion behavior, the programmer also needs to create two new classes for the motion handler and motion behavior of the stimuli. The new stimuli class then requires to have these new motion controllers as its members



```
class Water : public ArenaMobileENtity, public Subject {
public:
  void Notify(State s) override;
};
```



```
class Water : public ArenaMobileENtity, public Subject {
  Water() {
    motion_handler_ = new MotionHandlerWater();
    motion_behavior_ = new MotionBehaviorWater();
  }
public:
  void Notify(State s) override;
};

class MotionHandlerWater : public MotionHandler {
  /* implementation */
};

class MotionBehaviorWater : public MotionBehavior {
  /* implementation */
};
```