Design Document Draft for Observer Pattern

Design 1:

First observer pattern would be the subjects would be the food, light, and robot (basically all entities). These subjects would then provide the observers, food sensor, light sensor, etc. its position information which would be a method implemented in ArenaEntity.

Then these sensors would get the position information for all relevant entities such as Light Sensor would only grab the positions of all the Lights in the arena. With these positions it would calculate the distance from the sensor to all these Lights or whatever entities its keeping track of. It will then apply a weight to these distances as well as place a direction to it. The weight could be 1 / distance^2 and the direction is needed so robot knows where to turn the wheel to. It would package this information with a vector of structs where the struct would have the information about the weighted distance and the direction as well as any additional information required.

Finally robot with this information would then based on what kind of sensor motor it has change the appropriate wheel velocity for each entry in this vector.

Design 2:

Second observer pattern could be the subjects would still be the food, light, and robot (all entities) and the position information would still be a method in ArenaEntity.

Then the arena would be the observer instead of the various sensors. The arena would get the position information for all the entities in the arena and then pass that information along to all the sensors. The sensors would then do all the calculations it has to and either moves the robot within the sensor or gives the processed data back to robot so the robot could do what it needs to.

The robot would again based on the sensor motor move appropriately the wheel velocities after processing the data from the sensors.

Design Choice:

I will implement design 2 instead of design 1. The reasons for this is because it makes more logical sense and is easier to extend.

The subjects themselves cannot be passing information directly to the observer as in Design 1 because that validates the Observer Pattern.

In Design 1 to add another sensor it will need a pointer to a robot as well as a vector of entities it should get information about. Also including the robot would have a circular dependency problem. If the arena is the observer that to extend to another sensor the arena would send information to that sensor in a loop that may or may not need a new loop depending on what entity the sensor is tracking.

Tutorial to Add Additional Sensors

In this example we will assume a water sensor must be created and that water entities already exist within the application. First step is to make a header and implementation file called water_sensor.h and water_sensor.cc.

The header file needs to include <math.h>, "src/light_sensor.h", and "src/params.h". The water sensor will inherit from sensor so the class declaration would be:

```
Class WaterSensor : public Sensor
```

This class needs a constructor and a Notify function that is overridden as follows:

```
void Notify(Pose position) override;
```

The implementation file would define the constructor and the Notify method. The public constructor would provide default values for the color_, position_, and reading_. Ex.

```
WaterSensor::WaterSensor() {
        color_.Set(kBlue);
        position_.x = 0;
        position_.y = 0;
        position_.theta = 0;
        reading_ = 0;
}
```

The Notify implementation would increment the reading based on the distance of the sensor and position passed in. Example implementation:

```
void WaterSensor::Notify(Pose position) {
        reading_ += (1200.0 / (pow(calculateDistance(position), 1.08)));
}
```

Robot will need two private data members to represent these sensors as well as getters and setters. The two private data members would be:

```
WaterSensor * left_watersensor_;
WaterSensor * right_watersensor_;
```

Next arena would need to change because it needs a vector of pointers to water entities as well as notify the water sensors on the robots. Arena's vector of water pointers could be std::vector<class Water *> waters_. In Arena::AddEntity add another case in the if-else statement such as:

```
else if (type == kWater) {
    Water * water = dynamic_cast<csci3081::Water *>(entity);
    waters_.push_back(water);
}
```

Next have arena notify the water sensors on the robot in Arena::UpdateEntitiesTimestep() by adding another for loop in the nested for loop:

```
for (auto &robot : robots_) {
  for (auto light : light_entities_) {
    robot->get_left_lightsensor()->Notify(light->get_pose());
    robot->get_right_lightsensor()->Notify(light->get_pose());
  }
  for (auto food : foods_) {
    robot->get_left_foodsensor()->Notify(food->get_pose());
    robot->get_right_foodsensor()->Notify(food->get_pose());
  }
  for (auto water : waters_) {
    robot->get_left_watersensor()->Notify(water->get_pose());
    robot->get_right_watersensor()->Notify(water->get_pose());
  }
}
```

Finally, we can change Robot::UpdateTimestep() to make sure the velocity of the robot changes depending on the reading of the water sensors. For example, the robot would only go after water if the robot is not hungry add this code to the else part of the if-else where the if is “if (collision_override_):

```
  if (!hungry_) {
        velocity = robot_behavior_->processReading(
        left_watersensor_->get_reading(), right_watersensor_->get_reading());
  }
```

Note: depending on what readings should be in effect depending on the robot state will change whether you add the velocities together or overwrite the previous values.

At the end of the function be sure to reset the readings:

```
left_watersensor_->set_reading(0.0);
right_watersensor_->set_reading(0.0);
```

Design Document Draft for Strategy Pattern

Design 1:

In this design the interface is RobotBehavior. This is the parent class which has 4 subclasses: AgressiveBehavior, FearBehavior, LoveBehavior, and ExploreBehavior.

The parent class has a virtual method called processReading which takes in 2 parameters which are the readings of the left sensor and right sensor respectively. It then returns a WheelVelocity after running an algorithm to determine how to convert the sensor readings into a WheelVelocity object. It also has a protected member called light_max_reading_ for the algorithms used in the child classes LoveBehavior and ExploreBehavior.

In robot.h there is a RobotBehavior * robot_behavior_ member that is initialized to a new instance of one of the 4 child classes in arena's constructor depending on the type of the robot through a setter.

In robot's TimestepUpdate function, the WheelVelocity object is made by calling robot_behavior_->processReading(), passing in the left and right light sensor readings in that order for the light sensor readings. For the food sensor readings there is a RobotBehavior * food_behavior_ member that is set by default to new AgressiveBehavior() because robot's will always be aggressive towards food if they are hungry.

Design 2:

In this design the parent class is MotionHandlerRobot. This parent class has 4 subclasses: MotionHandlerRobotAgressive, MotionHandlerRobotFear, MotionHandlerRobotLove, and MotionHandlerRobotExplore.

MotionHandlerRobot would have a virtual method called UpdateVelocity which would be overridden in the child classes. This function would take in 2 doubles which would be the left and right sensor readings and then modify the robot's WheelVelocity accordingly and return nothing.

Robot would now have a MotionHandlerRobot * motion_handler_ instead of a MotionHandlerRobot private data member. This would be initialized in arena.cc constructor depending on the type of robot it is: kFear, kExplore, kLove, and kAgressive. The getter for motion_handler_ would have to return a pointer to a MotionHandlerRobot instead of a MotionHandlerRobot object.

In robot's TimestepUpdate function, the call would be motion_handler_->UpdateVelocity() and then would be passed in the left and right readings. For the food sensor there will be a motion_handler_food_ that would be initialized to new MotionHandlerRobotAgressive() because robot's are always aggressive towards food if they are hungry.

Design Choice

I implemented Design 1 over Design 2 because it follows OOP principles and it is extending functionality vs. changing existing functionality more than Design 2. In Design 1 the class RobotBehavior and its subclasses is to implement a strategy pattern to convert readings from sensors to WheelVelocities that will be passed to the robot's motion_handler_. In Design 2 MotionHandlerRobot not only controls altering the robot's WheelVelocity but also implements a strategy pattern that allows robot to determine what algorithm to run at runtime. In OOP a class should have a singular purpose so Design 1 is better in this respect. Design 1 also has less changes to existing code than Design 2. In Design 2, motion_handler_ has to be changed to a pointer, the getter has to be changed, MotionHandlerRobot now has to be a parent class with a virtual method, and the calls to UpdateVelocity() have to change through motion_handler_. In Design 1 the interaction between robot and motion_handler_ is not changing at all but instead it inserts a strategy pattern in between the robot's sensors and its motion_hander.

Tutorial to Add Additional Robot Behaviors

To add a new behavior we will use the example of Aggressive behavior. First make a new header and implementation file such as agressive_behavior.h and agressive_behavior.cc. Make sure to use include "src/robot_behavior.h" and "src/wheel_velocity.h". This class will inherit from RobotBehavior.

This class will need a destructor explicitly defined as "~AgressiveBehavior() = default;". The reason for this is because the parent class RobotBehavior has a virtual method processReading that the child class will override. Next make declaration of the function processReading as follows:

```
WheelVelocity processReading(double leftReading, double rightReading) override;
```

Next in the implementation file do not forget to include the header file, "src/agressive_behavior.h". Next we will define the method as follows:

```
WheelVelocity AgressiveBehavior::processReading(double leftReading,
  double rightReading) {
  // (+) Crossed
  WheelVelocity v;

  v.left = rightReading;
  v.right = leftReading;

  return v;
}
```

The reason for left velocity to be assigned the right velocity and vice versa for the right velocity is because aggressive behavior is (+) and Crossed. The positive means that the velocity is proportional to the sensor reading and crossed means each sensor is hooked up to the opposite wheel it is closed too. This means the right sensor is hooked up to the left wheel hence assigning the right reading to the left wheel.

Next define a new macro in params.h to determine number of robots that are aggressive and then add it to N_Robots:

```
#define N_ROBOTS_AGRESSIVE 0
#define N_ROBOTS (N_ROBOTS_FEAR + N_ROBOTS_EXPLORE + \
  N_ROBOTS_LOVE + N_ROBOTS_AGRESSIVE)
```

Next in Arena::AddRobot() add another for loop creating the robots:

```
for (; i < N_ROBOTS_FEAR + N_ROBOTS_AGRESSIVE; i++) {
  robots_.at(i)->set_robot_type(kFear);
  robots_.at(i)->set_robot_behavior(new FearBehavior());
}
```

The reason for the loop condition adding the previous loop condition variable, N_ROBOTS_FEAR, to the current type of robot being added, N_ROBOTS_AGRESSIVE, is so that the robots actually get added otherwise if it was simply i < N_ROBOTS_AGRESSIVE, i may already be greater than N_ROBOTS_AGRESSIVE due to the number of exploring robots.

The reason you do not have to change the code in Robot::TimestepUpdate() is the point of the strategy pattern. When the code:

```
velocity = robot_behavior_->processReading(
        left_lightsensor_->get_reading(), right_lightsensor_->get_reading());
```

is ran, at runtime the algorithm to determine how robot's react to light is determined and it will run AgressiveBehavior::processReading since robot_behavior_ is a pointer to a AgressiveBehavior object.