Design Document

The key component of the design consisted of creating 4 different types of sensors (which all inherit from Sensor) that will be used by the autonomous robots to avoid other arena entities when appropriate. The Sensor base class can accept an EventRecharge, EventCollision, and EventCommand. Moreover, each type of sensor accepts different types of events (which all inherit from EventBaseClass) in order to update the speed and direction of the robot/player accordingly. The types of sensors and the events they accept are as follows:

- SensorDistress accepts an EventDistressCall
- SensorEntityType accepts an EventTypeEmit
- SensorProximity accepts an EventProximity
- SensorTouch accepts an EventCollision

The interface for these sensors is modeled after the observer pattern in that the entities themselves have their respective sensors and they notify such sensors of any changes by sending them events. The sensors then activate themselves based on the information provided. Afterwards, the distinct motion handlers for the mobile entities (i.e. Robot, Player and HomeBase) update the velocity and heading angle of such entities based on the activation status of the sensors and any other pertinent information contained within the sensors (e.g. angle of contact, point of contact, etc.)

My main design choice for how sensors are notified differs from Professor Larson's suggestion in that instead of having the Arena contain a struct or list of sensors and updating them directly, it instead has a list of the entities and events and sends the events to the entities which then update the sensors according to the event information provided. The reason for this design choice is that I wanted to be consistent with my implementation of iteration 1 where the Arena only had the populating entities and updated them by sending events. I didn't want to provide sensor information to the arena because it doesn't seem necessary to have the arena contain sensors when its arena entities are the ones utilizing the sensors and, therefore, should be the only ones to directly communicate with the sensors by sending them information. Furthermore, I wanted to model a real-life behavior where the sensors aren't a part of the Arena, but are physical parts of the robots and player. Therefore, like how it was implemented in Iteration 1, the Arena checks for the collision under the arena method CheckForEventCollision() and checks for proximity events under the arena method CheckForEntityProximity() at each time step and such events are sent to the populating entities so that they can pass along the information to the sensors and have their velocity and direction appropriately changed (if at all).

One major design choice related to this was to not use the Entity Type sensor or the Event Type Emit event. Instead, I just had each entity have a getter method that returns an entity id that corresponds to its respective type. The following ids were associated with each type of entity

- 0 : Player
- 1 : Robot/Superbot
- 2 : HomeBase
- 3 : Recharge Station
- 4 : Obstacle

So, instead of sending entity type events to each entity and their sensors and obtaining information from the entity type sensors, I streamlined this process into having each entity directly return its respective id. Both methods offer the same results, but I found that I was able to get my program running more efficiently when ignoring the entity type sensor and events altogether since they aren't as instrumental to the functionality of robots as the proximity, distress and collision events and sensors are. Perhaps in iteration 3 I'll tweak the design to make use of all sensors but I would find this redundant when I'm obtaining the same results in a similar manner without using the entity type sensor and focusing more on the other sensors that are more integral to the overall functionality of the robot. However, given more time, I'm sure I could implement functionality of the entity type sensor to make full use of all the sensors. This would require some refactoring but could be accomplished in iteration 3.

Collision events are sent to each entity but events of type EventDistressCall and EventProximity are sent only to robots since they're the only ones that have the appropriate sensors to handle such events. Then, CheckForEntityProximity() is only called for a sensing robot and any sensed entity besides HomeBase since robots need it to turn into superbots. Moreover, CheckForEntityProximity() will only be called for a robot that doesn't have its distress sensor activated. The code within CheckForEntityProximity() is nearly identical to the one provided by Professor Larson in the proximity.py file to determine if the sensed entity is within range of the sensing robot. The arena does the work in reading the sensor information from robot, but the proximity sensor is the one that determines if the sensed entity is within range of the sensing robot. Once again, I decided to have the sensor itself decide whether an object was within its range to model the real-life behavior of a sensor in the real world. If the sensing robot effectively detects an in-range entity, then the robot will bounce off the sensed entity at angle of reflection (similar to how it does form regular collision events) so that it avoids colliding with the sensed entity. The reason for having the robot bouncing off at an angle of reflection is because I already had the code working for collision events and it wasn't specified how exactly robots should avoid object so I decided to adopt this methodology for proximity events. After handling the proximity and collision events, the robots that collide with the player are frozen and have their distress sensor activated so that other robots and superbots don't avoid colliding with them.

The next major design choice was how to implement Superbot and the way I accomplished this is through the strategy pattern by changing the functionality of the robot's motion handler instead of creating an entirely new Superbot class. The way I accomplished this is by setting a flag in Robot whenever it collides with HomeBase that can be used to implement different functionality within motion handler. One such example is that instead of MotionHandlerRobot changing the heading angle of robot in wake of a proximity event, it would now keep the heading angle fixed and set the activation status of the distress sensor to off since superbots aren't ever frozen and shouldn't avoid any objects. Although I have a flag within Robot, I don't use that flag to execute alternate chunks of code within my robot class. I instead use that flag to implement a different functionality for the motion handler and maintain the distress sensor turned off. The reason for doing this is that superbots shouldn't avoid any objects since they're able to unfreeze robots and freeze the player. I found this method more efficient than creating an entirely new SuperBot class that'll behave exactly like robot with the exception of the motion handler. Thus, I decided to focus my attention on the robot's motion handler.

Ideally, I would've liked to create two different motion handler classes that can be used for robot and superbot. Either one will be used depending on the current collision status of the robot. However, I didn't have a chance to implement this but it's something I'd be interested in implementing in iteration 3 if it'll improve my code and functionality.