Observer Design Pattern implementation:

My implementation of the observer design strategy takes advantage of the arena and it knowledge of all the entities that are in our simulation. I realized that the arena would be the best candidate to be the subject in my implementation of the observer strategy. I modified my arena to be my subject by creating several new pointer vectors that would keep track of the robot sensors and light and food entities. Therefore, when it comes time to notify and update the observers it would have all the info about those entities and can pass them along to its observers.

```
// All entities mobile and immobile.
std::vector<class ArenaEntity *> entities_;

// A subset of the entities -- only those that can move .
std::vector<class ArenaMobileEntity *> mobile_entities_;
// A subset of the entities -- lights only
std::vector<class ArenaEntity *> light_entities_;

// A subset of the entities -- lights only
std::vector<class ArenaEntity *> food_entities_;

// All of the lightsensors in the arena
std::vector<class LightSensor* > lightsensor_observers_;

// All of the foodsensors in the arena
std::vector<class FoodSensor* > foodsensor_observers_;
```

My choice for the observers of the arena was a bit tricky at first. It was a choice between making the robots observer or the sensors of each robot directly the observers of the arena. I end up implementing the strategy with the sensors as the observers of the arena because first it would mean less methods being called since if the robot it self was the observer it would then need update it sensor whenever it was updated itself.

```
void Arena::RegisterLightSensorObserver(LightSensor* ob){
  lightsensor_observers_.push_back(ob);
}
void Arena::RegisterFoodSensorObserver(FoodSensor* ob){
  foodsensor_observers_.push_back(ob);
}

void Arena::Notify(){ // called at each timesupdate
   for (auto observer: lightsensor_observers_) {
    observer->update(light_entities_);
  }
   for (auto observer: foodsensor_observers_) {
    observer->update(food_entities_);
  }
```

When the arena is creating robots it also registers the pointers to the sensor of that robot. Then the notify function is called at each times stamp update and that updates all the sensor with a vector all the entities its tracking.

Strategy design pattern implementation:

I implemented the strategy pattern in the way that robots exhibit different behaviors towards entities with the arena. I created different motion handler classes that inherit from the parent class. In particular, I have 4 different motion handler child classes that represents the four different behaviors robots exhibit (fear, love, explore, and aggression). Each of the classes then implement the update velocity function and thereby allowing it to implemented it with respect to the behaviors that class is representing.

```
// assiging the apporipate motion handler with respect to the robot type
switch(get_robot_type()){
  case(kAggressive):
  motion_handler_ = new MotionHandlerRobotAggressive(this);
  break;
  case(kLove):
  motion_handler_ = new MotionHandlerRobotLove(this);
  break;
  case(kCoward):
  motion_handler_ = new MotionHandlerRobotCoward(this);
  break;
  case(kExplore):
  motion_handler_ = new MotionHandlerRobotExplore(this);
  break;
  default: break;
}
```

Alternatives implementation:

Another way the behaviors could have be implanted would be by having a lengthy switch statement within the single motion handler robot class's update velocity function. The cases of this switch statement would be the robot's behavior type and according set the velocity. That inefficiency in this implementation would that a single class would be taking care

of so much. Its also not optimal for the future because behaviors can changed, added and removed.