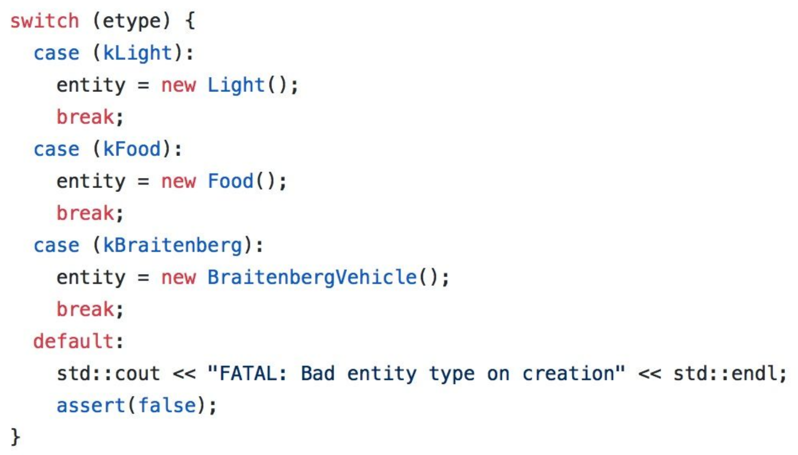
Iteration1 Design Document

Braitenberg vehicles are moveable objects that could exhibit complex behaviors based on simple design. Braitenberg vehicles depend on different sensors to change direction, position and speed. The goal of this project is to design a robot simulator that could be visualized in a graphics window and allow users to freely choose their preferred sensor types and behaviors.The project consists of an arena that allows arena entities to stay and move on it. Currently, there are three entities on the arena: braitenberg vehicles, food and lights. Braitenberg vehicles are mobile objects while food is stationary. Arena size and entity information are stored in JavaScript Object Notation (JSON) object. In order to make the interaction of Arena, entities and JSON closer and the whole project cohesive, we consider implementing a factory pattern.

Now, there are three choices to implement the factory pattern
1. In the provided code, directly instantiate the object
2. Use an abstract Factory class and derived classes
3. Use a concrete Factory to implement all the required objects

For the first method, it directly instantiate the object using a switch statement. The advantage is that the users/code readers can directly know which values are involved in the program and it is easier for error detection because switch statement divide the cases into different modules. By directly instantiating the object, we do not need to create multiple classes or include more files in the project. However, this method depends heavily on other concrete classes Light, Food and BraitenbergVehicle.we need to change a lot of code to make a small modification, which is not good for our implementation. For a small project, it may be a good choice but if we have 20 objects, it is complex to have 20 cases in a switch statement.

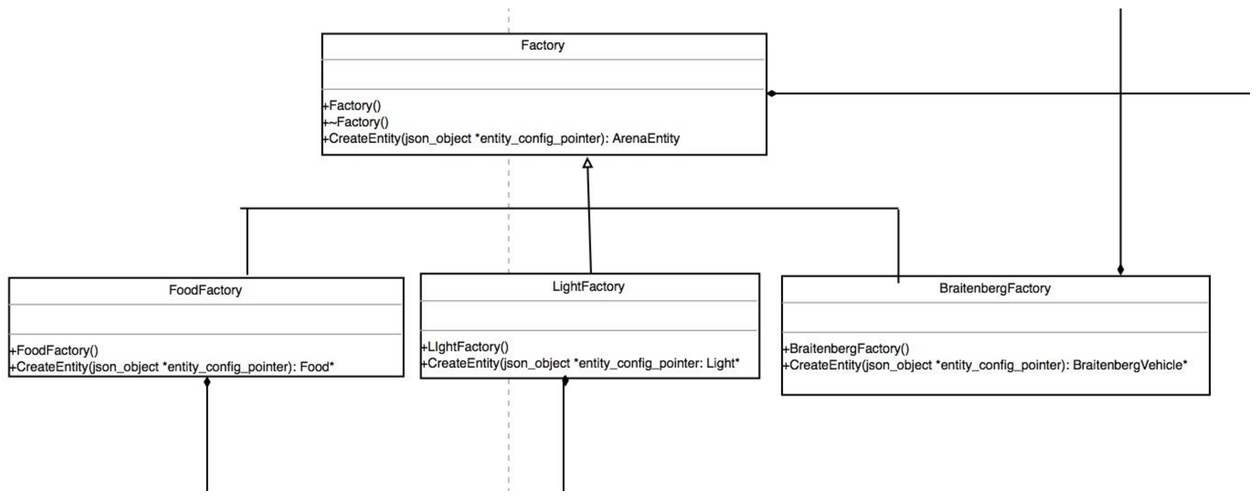


```
switch (etype) {
  case (kLight):
    entity = new Light();
    break;
  case (kFood):
    entity = new Food();
    break;
  case (kBraitenberg):
    entity = new BraitenbergVehicle();
    break;
  default:
    std::cout << "FATAL: Bad entity type on creation" << std::endl;
    assert(false);
}
```

For the second method, we have an abstract class and several derived classes. The advantage of this method is that it allows software design extensibility. It is easier for us to add more entities by just adding more derived classes, which is more flexible. The derived class only need to inherit from base class and it does not heavily rely on other class. The disadvantage is that with the increasing number of entities, we need to implement derived factory class for each newly added entities. Because some of them share the same function and property, we need to produce some duplicated code.

For the third choice, we can implement all the objects in one concrete class. The advantage is that we only need to write one header and one cc file to include all the functions we want. It is better for small project if we only have a very few kinds of entities. In addition, we can write methods that shared by all the object , which is convenient.The disadvantage is that for a large project, we need to write a very long class to include all the entities. It is not appropriate for debug purpose and it makes it difficult for other people to inspect the code.

In iteration1, I choose the second method to design factory patterns. First I created an abstract Factory class that serve as the “manager” of derived classes. Then I designed three different factories: Food Factory, LightFactory and BraitenbergFactory which are all derived classes from the abstract Factory class. The abstract class factory has a pure virtual method called create and allow the 3 factories to override. In Factory class, the function returns an ArenaEntity Pointer. In the 3 derived class, the overridden function returns Food, Light and BraitenbergVehicle, which are all instances of ArenaEntity.


Factory
+Factory()
+~Factory()
+CreateEntity(json_object *entity_config_pointer): ArenaEntity
FoodFactory
+FoodFactory()
+CreateEntity(json_object *entity_config_pointer): Food*
LightFactory
+LIghtFactory()
+CreateEntity(json_object *entity_config_pointer: Light*
BraitenbergFactory
+BraitenbergFactory()
+CreateEntity(json_object *entity_config_pointer): BraitenbergVehicle*

The implementation works as follows: in each derived factory class, it has a constructor. In create method, it creates a default entity and then use loadfromobject method and takes in a json object pointer and assign the values store in json object to the created object and return the produced object.

In arena, we can set a factory pointer and when a specific entity needs to be created, we just need to do a dynamic binding and create the corresponding factory and call create method. We do not need to rely on the concrete class Food, Light and Braitenberg vehicle to produce entities and it is easier to make changes,