Name: Khoa Tran – tran0707
Due date: 03/15/19

Iteration 1: Design Document
Compare and Contrast Three Versions of The Factory Pattern

The instantiation of entities in the provided code is the simple way to code and that is the way I have been coded for the past two years. The first advantages that we can see from this strategy are easy to read and understand the code. Everything is built inside of one file which is easy to find and grab a function or a variable when needed. A good code flow is presented by combing all variables and classes in one constructor class instead of breadth out into many small classes like the factory pattern. Despite this advantage, there is a big disadvantage is the data protection and code reuse issue. A good example of this is in Arena's constructor (Code1 below). The arena can see and modify anything in the process of creating new entities for food, light, and Braitenberg's vehicle. This is not ideal for good software or may potentially create bugs in the future. If we use the factory pattern to create food, light, and Braitenberg's vehicle instead, Arena won't be able to directly manipulate any other classes; furthermore, it improves the loose coupling of the code. It is best to use this direct object construction when objects need to create is used only in one single place or the create logic is really simple.

```
Arena::Arena(json_object& arena_object): x_dim_(X_DIM),
    y_dim_(Y_DIM),
    entities_(),
    mobile_entities_() {
 x_dim_ = arena_object["width"].get<double>();
 y_dim_ = arena_object["height"].get<double>();
 json_array& entities = arena_object["entities"].get<json_array>();
 for (unsigned int f = 0; f < entities.size(); f++) {
  json_object& entity_config = entities[f].get<json_object>();
  EntityType etype = get_entity_type(
   entity_config["type"].get<std::string>());

  ArenaEntity* entity = NULL;

  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);
  }

  if (entity) {
   entity->LoadFromObject(entity_config);
   AddEntity(entity);
  }
 }
}
```

Code 1: Directly instantiate of Entities

The use of an abstract Factory class and derived factory classes has many advantages compared to the other two strategies. The first advantage is easy to add more features and allow the loose-coupling in the abstract class and derived classes. This pro is so important for a coder because any application or software always need to upgrade, improve, or add new features in the future. For instance of figure 1, what if we want to add a new feature for the factory to create a traffic stop sign so that a robot will stop if the distance between the robot and stop sign is ten iterations. It will be super easy if we use this strategy of abstract factory and derived factories. This is because all we need to do is add another derived class and override the Create function from Factory class to create the stop sign. Are the other two strategies can do it? Sure they can do it, but the amount of code that needs to be modified is way larger compared with the abstract Factory strategy. Another important advantage of this method can be unit-testability. By implement each class separate from the entity_factory, we have the ability to write an unit-test class for each class and test for functionality, bugs, limitation, or test for meeting our expectations for each class. Next advantage will be the ability to reduce the number of dependencies per class. With the separation of the big factory into many small factories to do a specific job, each class now only required to include the necessary header files. Besides all these advantages, there are some disadvantages that we need to consider. When we use this method to code, it adds more complexity to the code. Instead of showing data and classes in one code, we create a new class to hide all data and the implementation which can cause the readers confused about the connection and usage of the code flow. A problem that I had in this robot project is that it's so hard to find where is a function definition is and why it got to call inside of this function or class. Then I have to look at all the header files and find that function which sometimes annoys and confuse me. This issue can be solved by a good detail UML graph and adding a good document on where this function from and what is that function will do.

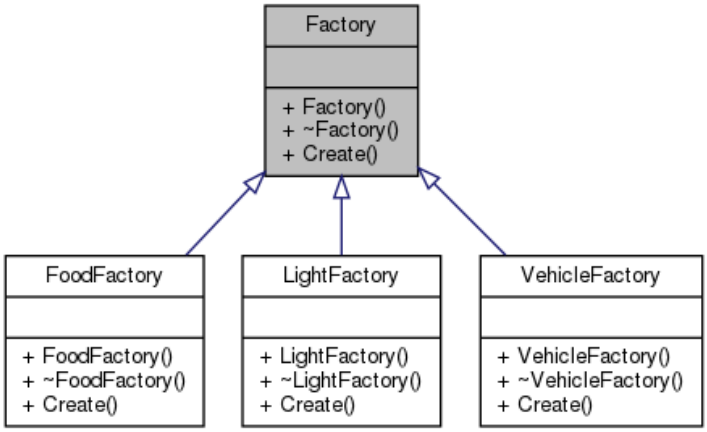

Figure 1: An Abstract Factory class and Derived Factories Class

Another strategy to create entities can be the use of a concrete Factory class that is responsible for the instantiation of all Entity types. This method is better when compared with the instantiation of entities in the provided code but still worse when compared with the derived class from the factory. It still gains some of the level protection of the private data but not as much as building a big abstract factory and derive it into many small factory classes. For example (figure 2), if Arena wants to create any objects using factory class, it can call the factory inside of its code to create objects. In this way, Arena can't directly manipulate any private data inside of the factory but the Factory class can. In contrast with the advantage, It is not loose-coupling because changing one part of the factory code may cause of changing the other parts of this class.

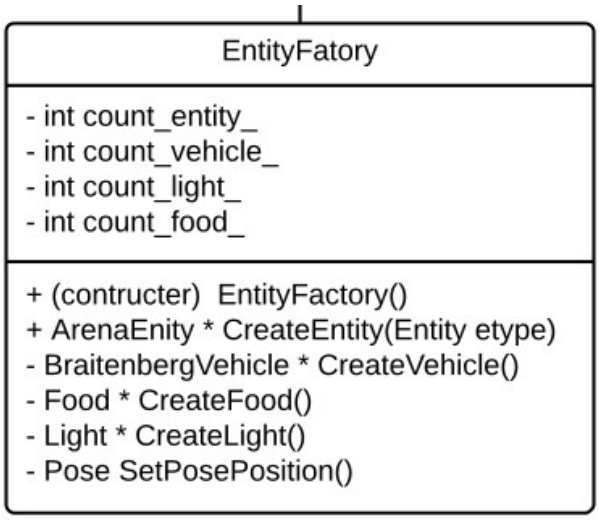

Figure 2: Concrete Factory Class

In conclusion, factory pattern design is a great way to create and control on certain criteria what objects should be created, so it is easy to maintain the factory in one place, instead of breath out throughout the whole project. Both abstract and concrete Factory design both have the ability to hide information, and keep classes loosely coupled; however, the extensibility is only provided by the abstract version of Factory design. With all the advantages of the abstract version of the Factory pattern design, I decided to choose this design for my Factory.