Factory Choices Design Document **Michael Ung - ungxx028**

There are three design choices that are suggested in the creation of the project: instantiation of the entities through the switch statement, creating a concrete factory class that instantiates all the objects, and creating an abstract factory class and deriving classes based on the type of object that it creates. Each provide different advantages and disadvantages and they should all be considered when choosing which one to implement, which is key to the design process.

In the given code, you can see the instantiation of the entities using a switch statement inside the constructor of the Arena class.

```
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);
}
```

Looking at it, I'm sure that the majority of people will see that as a pretty simple way to instantiate the Arena entities. In short, it is a simple way to do it and is probably what most will initially think of doing when looking to instantiate the entities since simple is good. It is comprehensible to most people. They will see the code and understand that objects are being instantiated. Although, this implementation seems to have the benefit of being understandable, you must take into consideration that this is not that loosely coupled and have little cohesion. Code related to the instantiation of entities will be littered throughout your code if you wish to create entities in more than one place and if you wish to add more entity types to the switch, then you must go find all the places that you create entities and add the new entity to the code that instantiates objects. This makes it less than ideal when following the idea of "closed to change, open to extension."

Next we must consider Factories, which is a better option. Usage of the Factory pattern allows for looser coupling since creating objects is separate from the rest of the classes that do not have deal with instantiation. This enforces better cohesion and allows you to get a better idea of how the overall process works. Within the factory implementation there lie two different options, using a concrete factory or using an abstract factory.

Containing all the Create methods for a factory in one factory is how a concrete factory works. An advantage of using a concrete factory, other than the fact that it is a factory and holds all the advantages of using a factory is that it has strong cohesion. Within the factory class that you would create, all the methods to instantiate are in the same place. The goal is to have all related functions in one place and that how a concrete factory is. The downside is that it is pretty complicated how you would use one factory to create different types of objects. If there were a scenario that required you to create a few different ArenaEntities from an array, you would create a factory object and then call Create<entity_type>. The problem is that depending on the way you create the factory, different difficulties may arise. For example, you could create it similar to how we did exercise 5 and have the parameter of the Create method to be one of the derived types such as below.

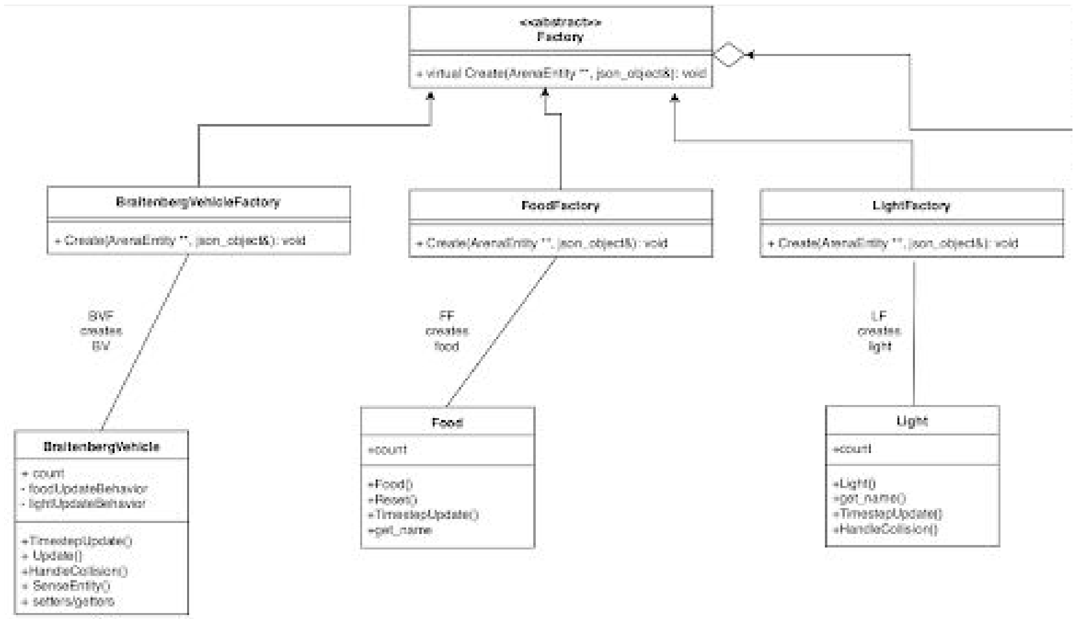
```
FactoryPointer::FactoryPointer(void) {}

void FactoryPointer::Create(Robot ** r) {
        cout << "Inside Create(Robot ** r)" << endl;
        *r = new Robot();
}

void FactoryPointer::Create(LeggedRobot ** lr) {
        cout << "Inside Create(LeggedRobot ** lr)" << endl;
        *lr = new LeggedRobot();
}
```

It would then require that you have an explicit type for wherever you declare your entities in order to create a LeggedRobot in this case. This way does not allow for good polymorphism and is a bit complicated. Although, there is also the benefit of being easy to apply extensions. Since all you would need to do if you wanted to implement a new ArenaEntity is creating a Create method of that entity in the pointer, given that you already programmed its behavior, there is not much other work to be done.

The final factory option is using an abstract factory. With an abstract factory you are able to clearly see the modularity of each factory and understand what each one does. With the abstract factory serving as the base class of the derived factories, making a LightFactory, FoodFactory, and BVFactory has a rather simple structure. It is loosely coupled since it doesn't deal with anything else other than creating the objects. One can control what is being made by choosing one factory type. Although not as prevalent in this project, a disadvantage is the extensibility of the abstract factory. You cannot simply add a new type into the factory without first changing the base class, then the other derived classes since the abstract class serves as an interface to them. For example, if I were to for some reason want to add a type called "Bug", I could not unless I added it to the base class, since "Bug" would not be of type "ArenaEntity". It the scope of the project, this should not be a problem since were are only dealing with ArenaEntities that include Light, Food, and BV. It is also my choice for implementation. In the UML diagram below, is the implementation idea that involves an abstract base class. They take in both an entity pointer to a pointer as well as a json_object& and is able to create object without any problem.



<<abstract>>
Factory
+ virtual Create(ArenaEntity **, json_object&): void
BraitenbergVehicleFactory
+ Create(ArenaEntity **, json_object&): void
FoodFactory
+ Create(ArenaEntity **, json_object&): void
LightFactory
+ Create(ArenaEntity **, json_object&): void
BVF creates BV
FF creates food
LF creates light
BraitenbergVehicle
+ count
- foodUpdateBehavior
- lightUpdateBehavior
+TimestepUpdate()
+ Update()
+HandleCollision()
+ SenseEntity()
+ setters/getters
Food
+count
+Food()
+Reset()
+TimestepUpdate()
+get_name
Light
+count
+Light()
+get_name()
+TimestepUpdate()
+HandleCollision()