Design Document

Approach 1: The instantiation of entities provided in the original code

Advantage: The instantiation of entities within the constructor of arena.cc allows for simplicity throughout the entirety of the entity creation routine. This ensures that any other developers that may want to understand how the system works are allowed to do so without information being hidden through various levels of abstraction that may have otherwise complicated the entire process.

Within arena.cc

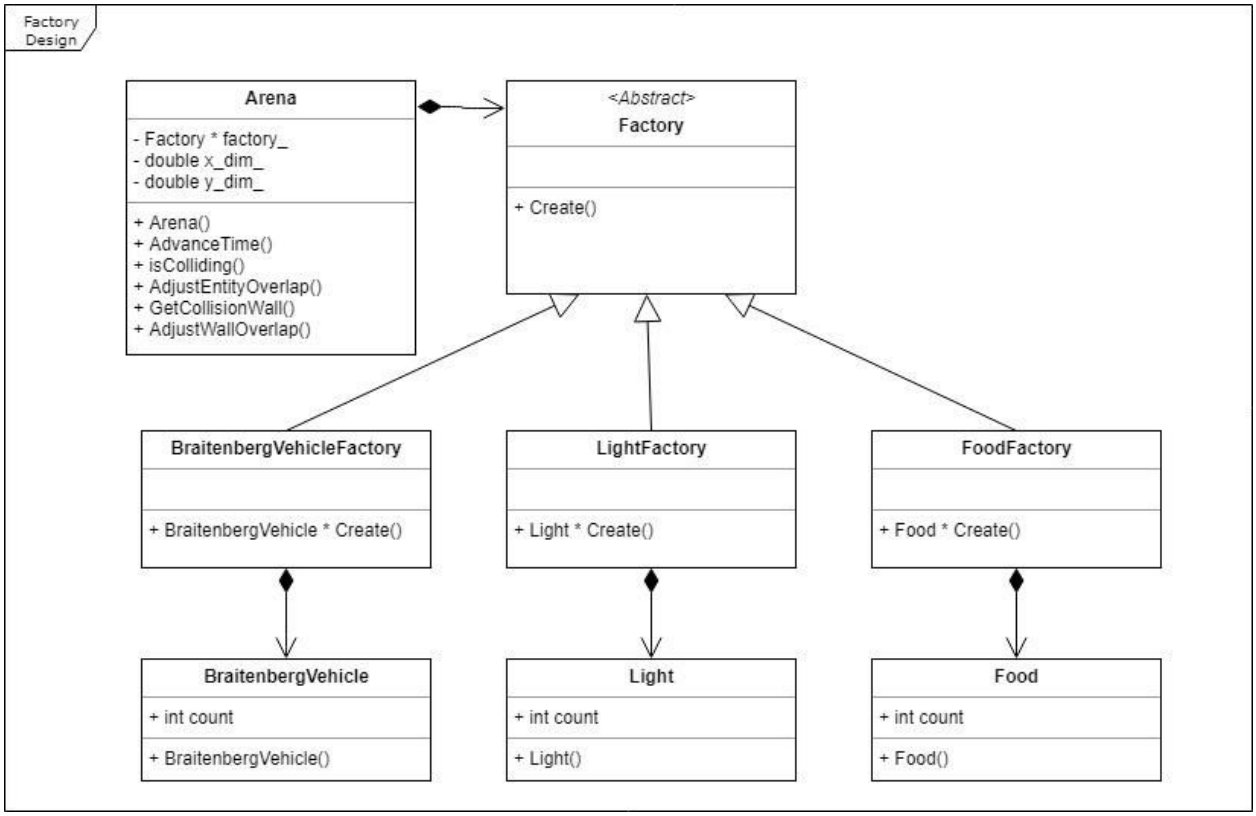
```
Arena::Arena(json_object& arena_object): x_dim_(X_DIM),

    ...
    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);
    }
    ...
```

Disadvantage: This approach creates many disadvantages to the entity creation routine. Firstly, this process chases simplicity but falls behind to flexibility. While the routine is easier to understand when laid out within a single constructor, the process doesn't allow for any feature extensions/enhancements for how each individual arena entity may be instantiated later in the development process. Secondly, encapsulation and information hiding is non-existent seeing as how the entire creation process for all of the various entities are visible within a single constructor.

Approach 2: The use of an abstract Factory class and derived factories

-Advantage: By utilizing this factory strategy, the arena class itself becomes unaware of the implementation of various entities within the arena, which not only keeps information hidden and data protected, but also allows the entities themselves to be open to future extension. This pattern also ensures that classes are loosely coupled meaning that by separating the creation of arena entities through factories, the 'entity creation' routine allows for small, direct, visible, and

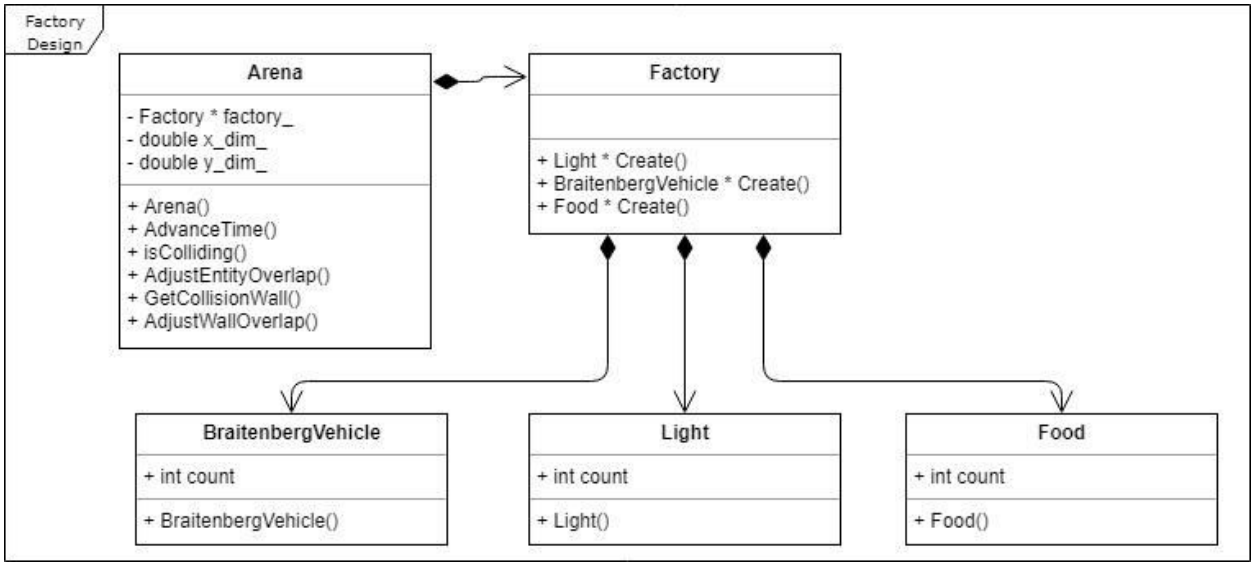
flexible class relationships. In addition, by creating separate factories for the various arena entities that inherit from a base abstract factory, classes are kept strongly cohesive in that all of the different routines (such as the creation of a Braitenberg object, a Light object, and a Food object) are situated within a respective factory class where all members of that class support the same purpose.



Disadvantage: By utilizing this factory strategy, it can become harder for developers to follow the subroutine of the creation of arena objects seeing as how the code is hidden behind different levels of abstraction. In essence, simplicity is being given up in exchange for flexibility.

Approach 3: The use of one Factory class that is responsible for the instantiation of an entity

-Advantages: While this approach is fairly similar to the previous factory strategy in which a factory class is responsible for the instantiation of entities, this version helps the main issue of its predecessor: the routine is kept simpler. By having a singe factory class in charge of the instantiation of all Entity types, the process removes a level of abstraction that might have otherwise complicated the overall routine.



-Disadvantage: Because a level of abstraction is lost in this implementation, we lose the advantage of encapsulation in that expectations to how we instantiate one entity type may cause issues with the entirety of the factory routine. In addition, we also give away how the various subroutines of each entity type work meaning we lose the ability to utilize good design through information hiding.

Version Implemented

The factory pattern implemented for this project was the one detailed in approach 2, seeing as how the advantages of utilizing this implementation outweighed the disadvantages that came with it. While increasing the levels of abstraction between interacting classes can lessen the simplicity of the overall structure, encapsulation and data protection is an important essence of software design that must always be sought after.