Brian Cooper (5310799)
coope824@umn.edu

The Braitenberg vehicle simulation software employs a factory design pattern. There are three core approaches to such an implementation, and their implications are outlined below. Specifically, the approaches describe various methods of instantiating entities within the software.

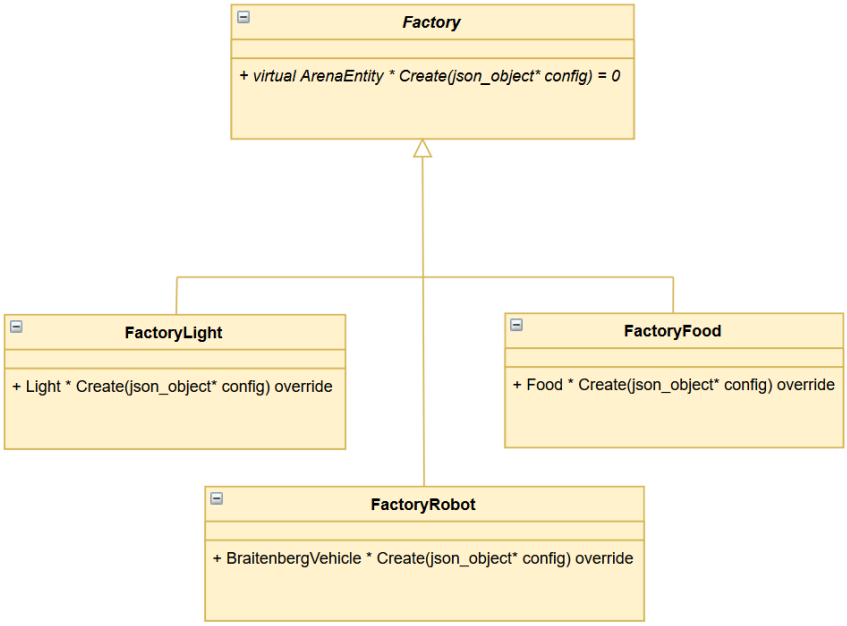
Approach #1: Instantiate the entities in the provided code.

This approach does not necessarily employ the factory pattern at all. This approach represents the “control” group of the use of a factory pattern in the software design. This is a poor approach due to failing to represent the “single responsibility principle” in software design, which states that each module should be responsible for a single part of the software functionality. In this approach, the code would involve control over many aspects of functionality in one source file. With that being said, one may desire this approach due to the convenience of having the entity instantiation code completely tied in with the rest of the source, so they wouldn’t have to worry as much about inheritance and understanding the overall structure of the software. This approach has high coupling, low cohesion, and maintainability is difficult.

Approach #2: Use an abstract *Factory* class with derived factories for each corresponding entity.

This is the approach used in the design of this software. Essentially, one class (an abstract factory class) serves as a “master” factory that is simply an interface for individual derived classes for each of the entities that require instantiation. Each derived class (called such because they inherit from the main abstract factory class) uses the interface provided by the abstract class to generate a specific entity corresponding to the derived class. This approach isn’t very flexible compared to the other two approaches discussed here, but it does make traversing the code easy, since someone looking at the code could very clearly see the various factory classes tied to each entity type when analyzing the source code file structure. This closely follows the “single responsibility principle” described above (in approach #1). As such, this improves software cohesion and maintainability. Below is a UML diagram of this approach, which reflects how the factory is currently implemented in the software.

Brian Cooper (5310799)
coope824@umn.edu



Abstract factory class with derived classes, as currently implemented.

Approach #3: Use one concrete *Factory* class to control the instantiation of all entities.

This approach would be excellent if one wanted to modularly expand the simulation to include many entities. Since there would only be one class file to control the entity instantiation, it would be easy to add an entity because a developer would only need to change the code in this class. However, since this project only currently involves lights, food, and robots, there aren't too many entities to worry about. As such, I decided to create separate class files for each entity factory. One drawback to this approach is that if there are many entities, the code could become cluttered and would remove from the modularity quality of the design.