BRAITENBERG VEHICLE SIMULATOR
DESIGN DOCUMENT

BRIAN COOPER
CSCI 3081W

TABLE OF CONTENTS

Page

Brian Cooper (5310799)
coope824@umn.edu

Factory Pattern

The Braitenberg vehicle simulation software employs a factory design pattern, which involves generating entities as needed from "creation" calls to a respective factory class (think of an actual factory, forming a product systematically). 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.

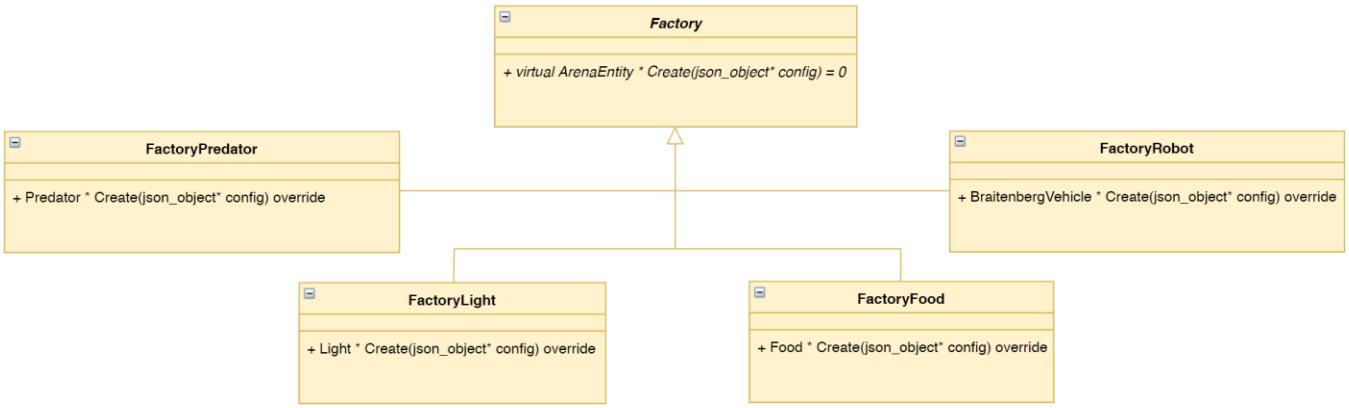
Approach #1: Instantiate the entities in the provided code.

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.

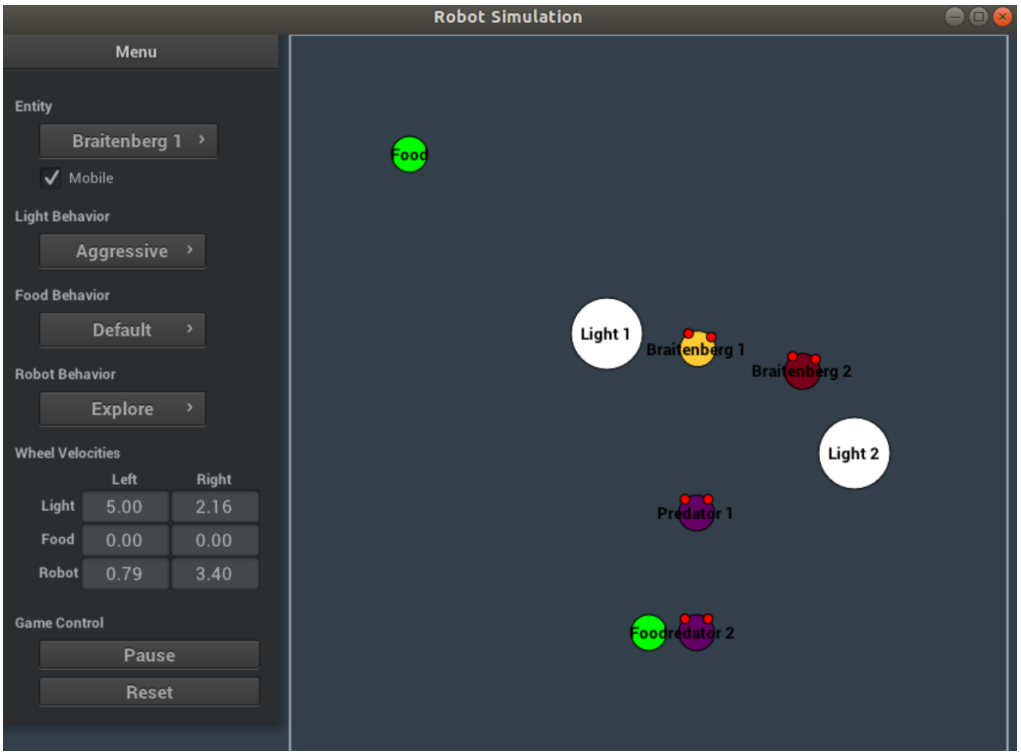
Approach #3: Use one concrete *Factory* class to control the instantiation of all entities.

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.

Brian Cooper (5310799)
coope824@umn.edu

Observer Pattern

Another design pattern implemented in this software is the observer pattern. The observer pattern is used to present the Braitenberg vehicle wheel velocity contributions to a user. The contributions in question are from food, lights, and other robots. The calculated values are displayed in the GUI below the entity selection menu. The functionality of this is further described in *Approach #1* below.



An example of the wheel velocity contribution display. Braitenberg 1 is in focus, and actively sensing a light and Braitenberg 2.

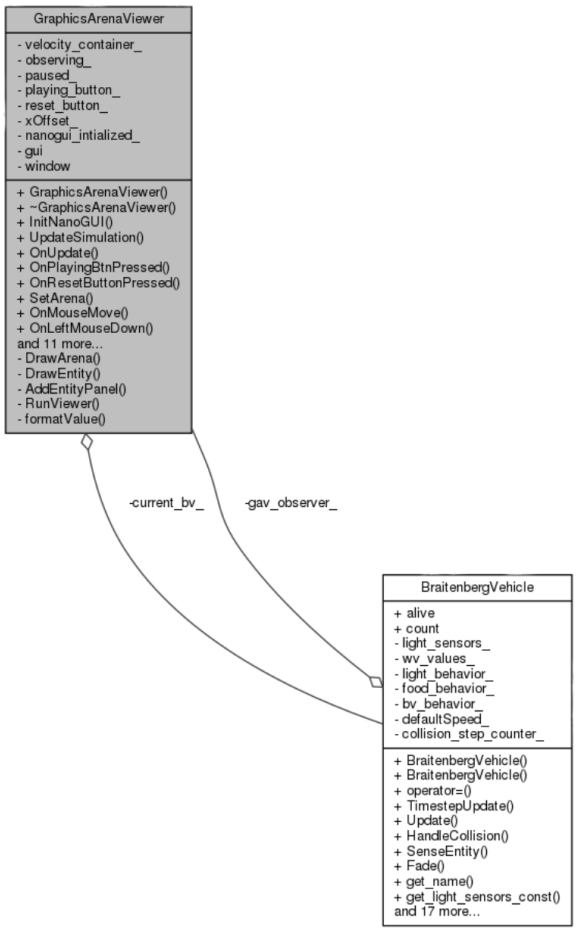
Two approaches for displaying wheel velocity contributions are outlined below: using the observer pattern (as is the current implementation), and simply using getters and setters between the BraitenbergVehicle class and the GraphicsArenaViewer class.

Approach #1: Use the observer pattern methodology to present the wheel velocity contributions.

The observer pattern involves communication between two parties: a subject and an observer. The subject, upon updating a feature of interest, notifies an observer after the update. Any observer that needs to be informed of a subject's update will "subscribe" to the subject in question. In the case of this software, the Braitenberg vehicles are the subjects, and the GraphicsArenaViewer ("viewer") is the sole observer.

Brian Cooper (5310799)
coope824@umn.edu

Upon selecting a Braitenberg vehicle from the entity selection menu at the top of the GUI, the viewer subscribes to that robot (by calling the robot's *Subscribe* method with a pointer to itself). Upon switching out of that robot in focus with the entity selection menu, the robot's *Unsubscribe* method is called, which removes the viewer from observing that robot. As long as a robot is in focus with the menu (i.e. the viewer is subscribed to that robot), the robot's *Notify* method is called with each timestep update (in the robot's *Update* function), where a state of the wheel velocity contributions is sent to the viewer. In the *Update* function, the aforementioned wheel velocity contribution values are stored in a vector, and this vector is what is handed to *Notify*.



My implementation of the observer pattern.

A "true" observer pattern implementation involves an abstract observer class and an abstract subject class from which a concrete observer (like the viewer) and a concrete subject (like a robot) inherits from, but I did not implement it this way for convenience, such as to avoid situations like multiple inheritance. It is also worth mentioning that my implementation has one observer pointer (which is either NULL or

Brian Cooper (5310799)
coope824@umn.edu

pointing to the viewer, depending on whether a robot is in focus or not). This could be changed to involve a container of observer pointers, which would allow multiple observers to be added and removed easily. Since the viewer is currently the only observer in this implementation, there is no need to have such a container. If more observers were to be added, this would be an easy approach, and would also increase the usefulness of this solution (in other words, this approach works well in the case of one subject to many viewers).

An advantage of this implementation is that it improves decoupling, due to the logic being handled by separate, dedicated functions. A disadvantage of this implementation is that it adds a level of complexity that is not inherent to approach #2, outlined below.

Approach #2: Use getters and setters between the viewer and the robot classes.

This method involves accessing a robot's wheel velocity contributions from a getter method within the viewer. After obtaining the wheel velocity contributions from a robot, the viewer would then use a setter to set the values in its GUI boxes. This process would repeat for each timestep update. This approach has an advantage over my observer pattern implementation: it is very easy to make a change regarding any of the code connected to this feature (viewing wheel velocity contributions). It is a very simple approach that does not involve any of the complexity of the observer pattern. However, this approach is highly coupled, since there is no separation of logic or states. Similarly, it is not as extensible as the previous approach. Adding another observer would involve adding more getters and setters, whereas approach #1 would only require adding another observer to a container of observers. The observer pattern has low interdependence between subjects and observers, since subjects simply broadcast a state to whoever is listening (subscribed).