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);
}
```

The code above shows that it is an easy 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.

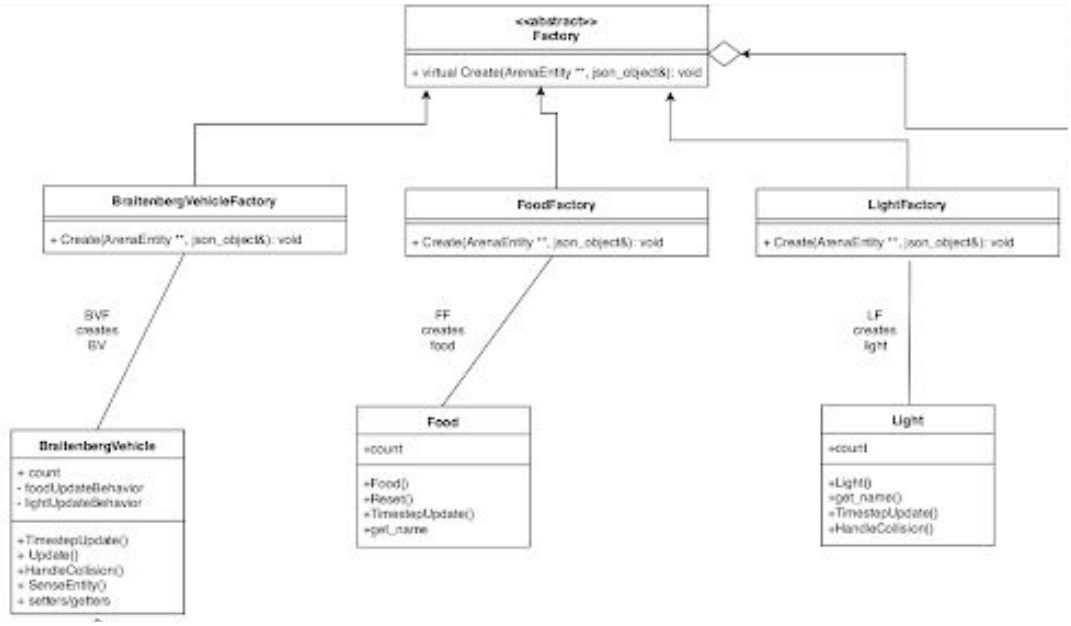
```
FactoryPointer::FactoryPointer(void) {}

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.



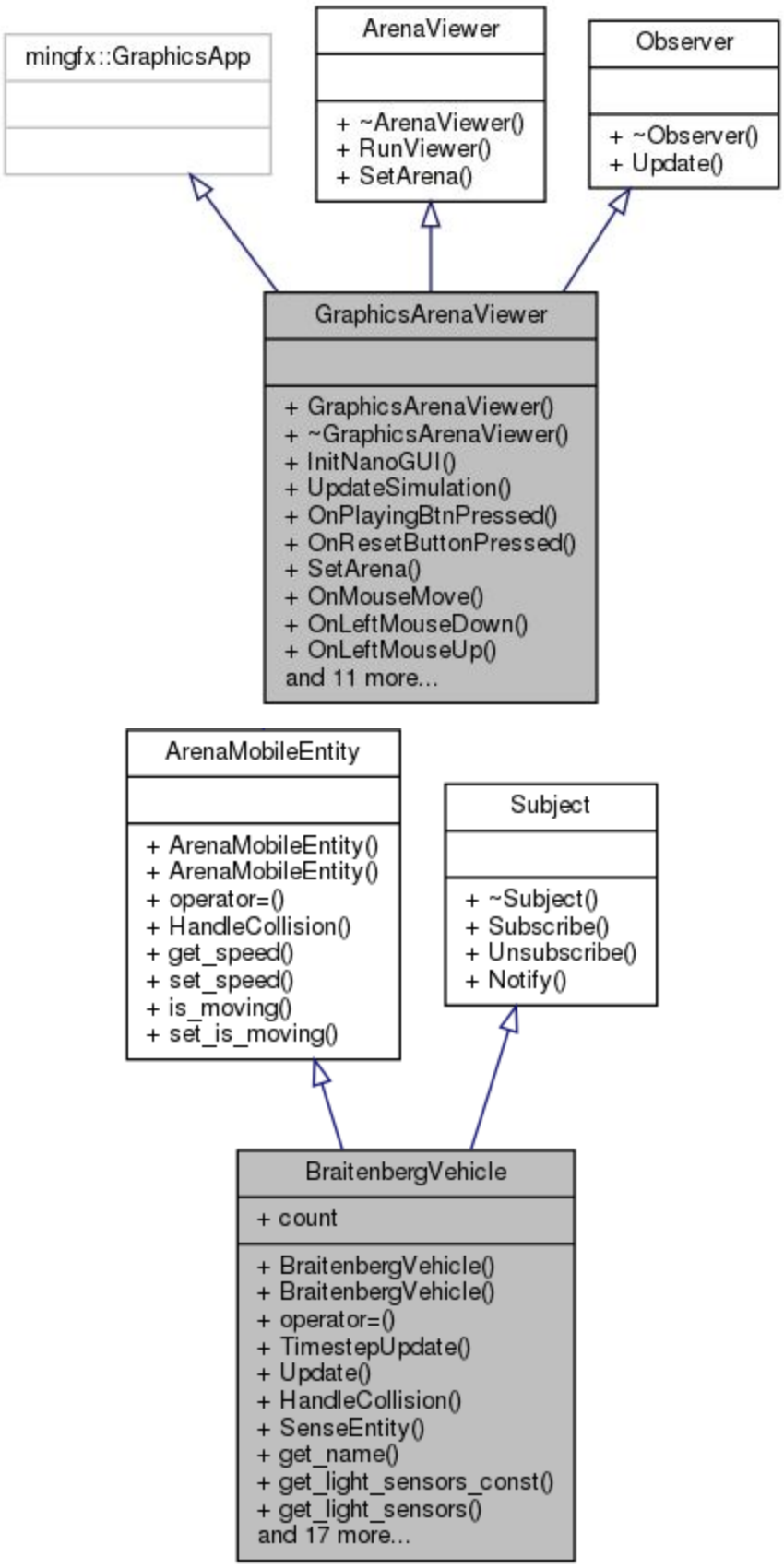
Another pattern that was used during the creation of the simulation is the observer pattern. Of the few ways that the observer pattern may be implemented, I decided to create an abstract observer class and have the GraphicsViewerApp use multiple inheritance and inherit from it. The same was also done for the subject class; the BraitenbergVehicle class similarly inherited from the subject abstract class. Using the setup that was described above, it is very easy to extend it to other observers and subjects which puts it at a great advantage to other versions. All that is required of the new classes is to implement the base classes. Multiple inheritance may make things a bit confusing, which may be a downside.

```
47
48 class BraitenbergVehicle : public ArenaMobileEntity, public Subject {
49  public:
50   /**
51    * @brief Default constructor.
52    */
53   BraitenbergVehicle();
54
```

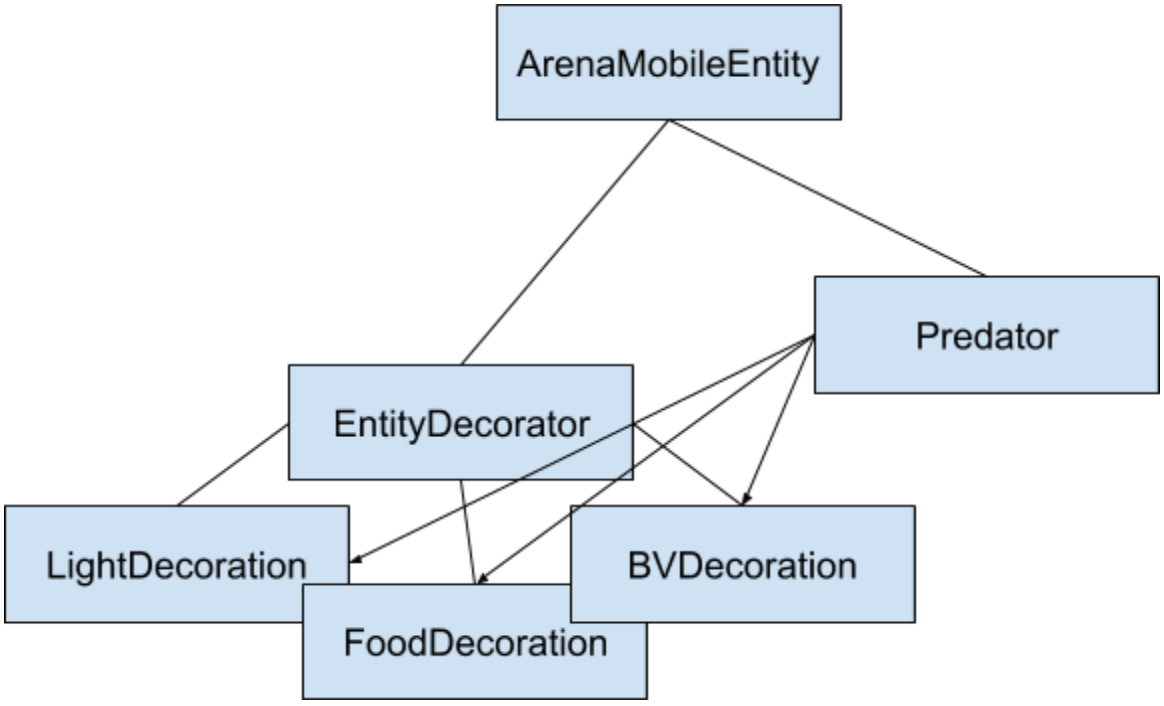
Using accessors would one alternative to creating the functionality that we have. The GraphicsArenaViewer would use accessors to get the needed information from the BraitenbergVehicle. Although with an advantage of being very simple, it likely would be poor program design due to having poor cohesion/being tightly coupled. This is because the GraphicsArenaViewer needs to have a BV object inside to use accessors. Whereas having two classes that are handling different functions together, in this case GraphicsArenaViewer handles the graphics, while BV is an entity, shows the poor cohesion.

The observer pattern that I designed is the same as the one implemented in the final submission. I saw no need to change it since it was functional and simple.

Below is the UML of GraphicsArenaViewer and BV to show the structure.


mingfx::GraphicsApp
ArenaViewer
+ ~ArenaViewer()
+ RunViewer()
+ SetArena()
Observer
+ ~Observer()
+ Update()
GraphicsArenaViewer
+ GraphicsArenaViewer()
+ ~GraphicsArenaViewer()
+ InitNanoGUI()
+ UpdateSimulation()
+ OnPlayingBtnPressed()
+ OnResetButtonPressed()
+ SetArena()
+ OnMouseMove()
+ OnLeftMouseDown()
+ OnLeftMouseUp()
and 11 more...
ArenaMobileEntity
+ ArenaMobileEntity()
+ ArenaMobileEntity()
+ operator=()
+ HandleCollision()
+ get_speed()
+ set_speed()
+ is_moving()
+ set_is_moving()
Subject
+ ~Subject()
+ Subscribe()
+ Unsubscribe()
+ Notify()
BraitenbergVehicle
+ count
+ BraitenbergVehicle()
+ BraitenbergVehicle()
+ operator=()
+ TimestepUpdate()
+ Update()
+ HandleCollision()
+ SenseEntity()
+ get_name()
+ get_light_sensors_const()
+ get_light_sensors()
and 17 more...

When introducing the decorator pattern, it was very difficult to visualize/comprehend. Initially, I made an abstract base class for the decorator called EntityDecorator that inherited from ArenaEntity. I later came up with many problems regarding encapsulation and had to move many functions and member variables up the tree. I later was able to move it back down the tree by inheriting from ArenaMobileEntity, which made it easier to implement. Compared to the version that used aggregation, this one is more loosely coupled, so it can be removed or extended with ease. For example, if you wanted to make it so a disguised predator moves slightly differently from the class it disguises as, it is not as coupled so you would only have to change the decorator that encapsulates the predator. Using aggregation is better in the sense that it is easier to implement, since you do not have to move functions around. Compared to another implementation that is not the decoration pattern, hard coding the functionality of a Light, Food, and BV into Predator is an alternative. You would need to change the appearance of the Predator and likely copy and paste a lot of code. This is a downfall due to the need to reuse code and the reduction in readability of your Predator class. Although there are these downfalls, the complexity is much less compared to the one that I implemented. Being simpler means it will take less time to code and less prone to errors. See below for a depiction of the relationships between the decorators and predator. Where the lines with no arrows are inheritance up the tree.



The final factory pattern implementation did not change at all. Instead, a new function was created in the controller class to parse a CSV file into JSON format. It is similar to an adapter class, but it is not encapsulated as such. If I were to implement the adapter pattern instead, I would need to separate the function in the controller to a new class which would make it more loosely coupled. Other than that, not much changes. Changes were also made to the controller to take in dimensions for the simulation. Simply concatenating parts to the string that gets

converted to a JSON object allowed it to function. I chose not to use the adapter pattern because it was already simple enough to create it inside the class and there would be no need to extend it after it was created. Below is a code snippet to show how simple the code to convert to CSV could be in the controller after a function was created.

```
  if (temp.substr(temp.length() - 4) == ".csv") {
    str = convertCSV(temp);
  } else {
    std::ifstream t(std::string(argv[3]).c_str());
    std::string jstr((std::istreambuf_iterator<char>(t)),
                     std::istreambuf_iterator<char>());
    str = jstr;
  }
  std::string preljson = "{\n  \"width\": " + std::string(argv[1]) +
    ",\n  \"height\": " + std::string(argv[2]) + ",";
  str = preljson + str.substr(1);
  std::string json = str;
  config_ = new json_value();
  std::string err = parse_json(config_, json);
  if (!err.empty()) {
    std::cerr << "Parse error: " << err << std::endl;
    delete config_;
    config_ = NULL;
  } else {
    arena_ = new Arena(&config_->get<json_object>());
  }
}
```