John Strobel Strob105

Design Document

There are two difficult elements of design when it comes to massive object-oriented projects: object creation and coupling. The first is difficult because there are many occasions when there is some amount of work that needs to be done upon instantiation of an object, but the work cannot be done by the object constructor itself. The second is difficult because objects need to remain distinct and completely separate from each other, but still able to communicate. In this document, I will address solutions to both problems.

Between the three ways of creating the various simulation entities, instantiating them as is in the provided code is probably the worst, objectively. From our point of view, relegating everything to a constructor (that can be easily modified without having to modify the points of instantiation) fits our purposes perfectly. This is true regardless of which version of instantiation you choose to employ. This leaves three instantiation strategies for the proposed factory: having a single class with many different construction methods, an abstract factory class in which one instantiates subclasses to do the construction, and finally a single class that 'has' another class called FactoryBehavior.

Of the three strategies, creating subclasses the least efficient and controllable. The whole purpose of this style of design is to abstract away the creation process from the user-programmer (aka. future me) and ensure that modification will not break any existing code. I believe that if you have a class for each type of object that you are trying to instantiate, the problem of having to hunt through code in order to fix each point of instantiation will not be solved. The only difference is that you will be changing the factory instead of a class definition, which is little consolation. There is an upside to this, though: if you want to change the factory for a single object, you can easily avoid change to the other types of object factories.

Having a single class that 'has' a behavior class that determines how the factory operates is actually quite similar to creating factory subclasses. Instead of a single creating entity, you must create a factory and factory behavior for each type of entity that you want to create. The

cons are much the same as above, but there is a pro: because the behaviors would be loosely coupled with the factory main class, it should be easy to modify the behavior parent or children classes.

Finally, the best option is using a single factory class and writing a method for each different desired construction type. The advantages to this system are, primarily, that you don't have to deal with *another* complex class structure on top of the one that you are trying to manage by creating factories. In this case, each type of object is represented by a single function, and the factory can be stored as a global object. The downsides to this would be that, in large projects, you would likely end up instantiating multiple versions of this factory anyways, somewhat offsetting the benefit. Additionally, with a large library of objects, having a single factory with 80 functions would likely be overwhelming for developers. However, in a project like the one given where there are perhaps 10 different types of objects, a single factory is the optimal solution. This is the solution I opted to implement, using a unique method for each object that the factory should instantiate.

Coupling is, of course, a completely different issue. In this project, we decided to implement a feature that would display the wheel velocities of each Braitenberg Vehicle in real-time. However, the class in charge of the actual display did not have access to the WheelVelocities of the Vehicles it was drawing (by design). So, that left us with just a few options: public getter methods, or and observer pattern.

Public getter methods are fairly standard practice in object-oriented programming, and here is no exception. However, writing 6 getters, one for each component of the wheel velocities, would be clunky and would not represent the kind of elegant solution that we as programmers try to accomplish. That being said, getter methods are very straight forward, requiring very little explanation to understand. In a simpler project with less room for growth, getters may have been a good option.

Instead of using getter methods, we decided to use an Observer pattern. An Observer class functions as a passive watcher, recording any data sent to it by a subject. The subject, in turn, is only responsible for notifying its observers when its member fields change. Once the data

is in the observer class, it can be sent out to a database, accessed via the getter methods of the observer, or any other convenient solution. This is a good strategy for extensibility because each entity that wishes to view information from a subject can just create a corresponding observer and process the data as it wishes. The subject doesn't have to worry about processing the data differently, and the observer class can be used for a large number of different uses. However, it is much more complicated than just using getter methods and may not lend itself well to smaller projects.

I chose to implement the observer pattern. My observer class is abstract, but can be easily inherited by any class that wishes to make use of the pattern. For this project, GraphicsArenaViewer inherits from Observer in order to get access to the WheelVelocities of BraitenbergVehicle entities:

```
class GraphicsArenaViewer :/* … */ public Observer {
```

It also instantiates two other methods in order to communicate with the subject. `Update` is called by the subject in order to update the observer's data, and `RequestUnsubscribe` is used to, unsurprisingly, request that the subject manually unsubscribe this observer. In the BraitenbergVehicle class, three standard methods are used to communicate with the subject. `RegisterObserver` is used to add a new observer to a maintained list of observers, `RemoveObserver` is used to remove any observers from the list of subscribed observers, and `NotifyObservers` runs through the list of observers and calls `Update` on each of them.