Name: Khoa Tran
x500: tran0707
Class: Csci 3081W

I. Iteration 3 Design Document

Decorator Pattern

Decorator Pattern is a pattern that allows a user to add functionality on top of the existing object without destroying the structure of that object. In this iteration 3, I used the Decorator Pattern to implement Predator that has the ability to disguise itself as food, light, or Braitenberg Vehicle. Once a new entity is wrapped on top of the original object, this disguised Predator should behave exactly as the new wrapped entity should behave. For instance, if a Predator that disguise a Food, it should remain at one position but not get eaten by a Braitenberg Vehicle.

There are two main alternatives to implement the disguise Predator including inheritance and aggregation. My version of disguise Predator is implemented by the decorator pattern using inheritance from Light, Food, and Braitenberg Vehicle classes (figure 1). I had three decorator classes called Food_Decorator, Light_Decorator, and BraitenbergVehicle_Decorator. Food_Decorator class inherited from Food class, Light_Decorator class inherited from Light class, and BraitenbergVehicle_Decorator inherited from BraitenbergVehicle class. The other approach that also used inheritance for the Decorator design is to inherit all threes class Food_Decorator, Light_Decorator, and BraitenbergVehicle_Decorator from Predator class (figure 2). One advantage of my version of inheritance (figure 1) is the ability to minimize the amount of duplicate code for this project by sharing common code among classes. In figure 2 approach, if a Predator wants to behave exactly like Light, Food, or Braitenbergvehicle, each Decorator class need to copy-and-paste all get_name, TimestepUpdate, HandleCollision… from the entity class which it wants to disguise. This leads to the result of unnecessary duplicate codes.

Besides the advantage above, the main disadvantage of using inheritance is that two classes get tightly coupled. This means one class change will cause the other class behavior also changes. For example, what happens if we want to change the TimestepUpdate method in Food class, it will lead to the behavior of the Food_Decorator also change. This change can cause Food_Decorator to behave in a way that the programmer doesn't want it. Different from the inheritance, aggregation approach is a "has-a" relationship which means for every modifies in Food class, we can expect the same behavior on the aggregation class.

Another approach for the Decorator Pattern is using the Aggregation approach (figure 3). In this project, the aggregation approach has the encapsulation advantage that is all disguise activities of Predator happen inside of Predator class only instead of spreading out to many decorator classes like the inheritance approach. To implement the Decorator Pattern in Predator class, we can add all the ability to Predator that can disguise, wrap, unwrap, die… which helps us in binding the data and the member functions in only one class. It also helps to hide the data from the other classes and no duplicate code. In contrast with aggregation, inheritance needs three different classes to inherit from three entity classes and everything will be handled inside of Arena class. With aggregation approach, it also has a disadvantage in the behavior of the system that may be harder to understand just by looking at the source code. It is because aggregation is more dynamic and more interaction between classes happens in run-time, rather than compile time.

Because of the advantage of limiting a lot of duplicate code, easy to control, and the ability for later development of this project, I chose the inheritance approach as my design for the Decorator Pattern.

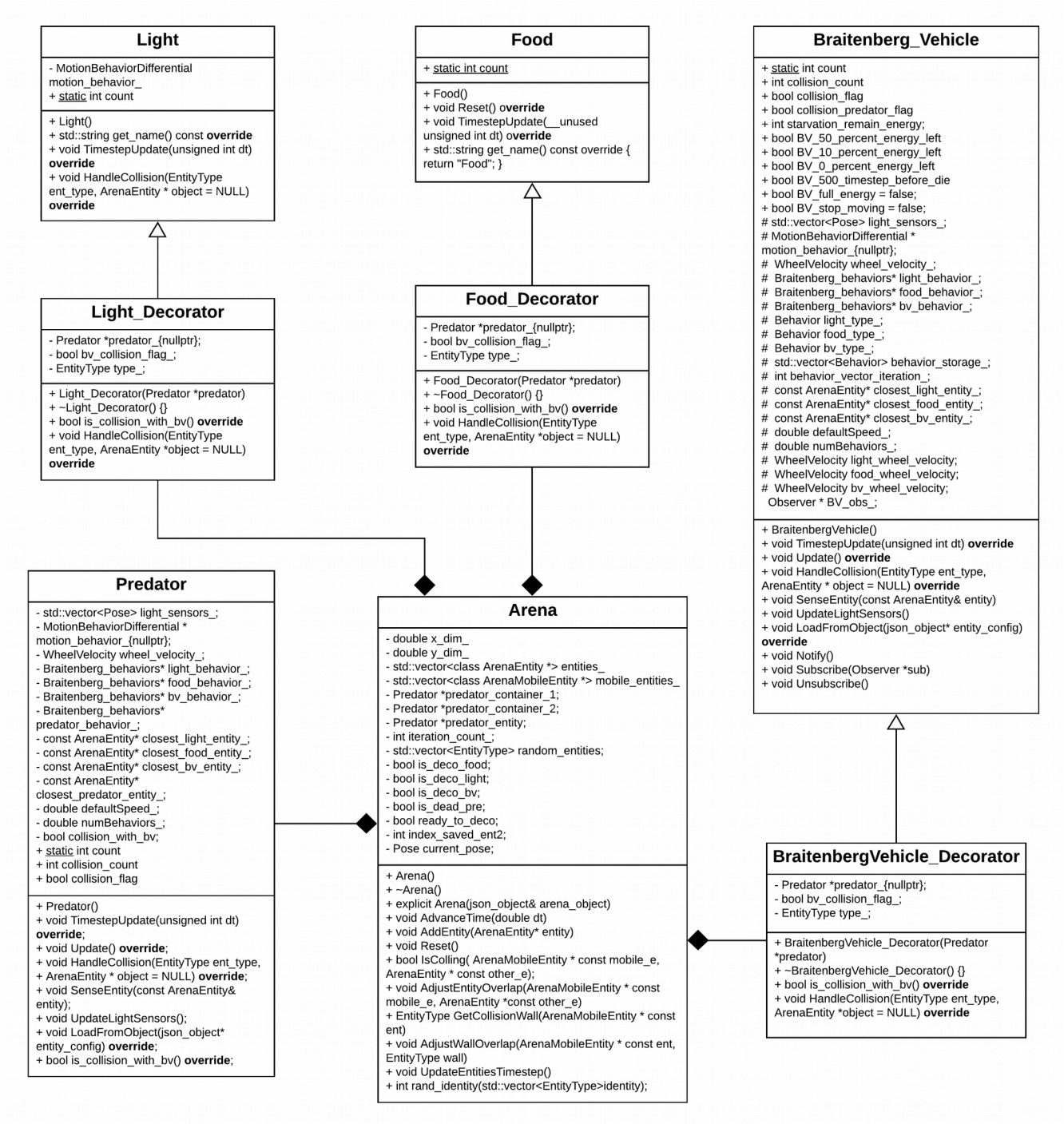


Figure 1: Inheritance Approach of Decorator Pattern (version 1)

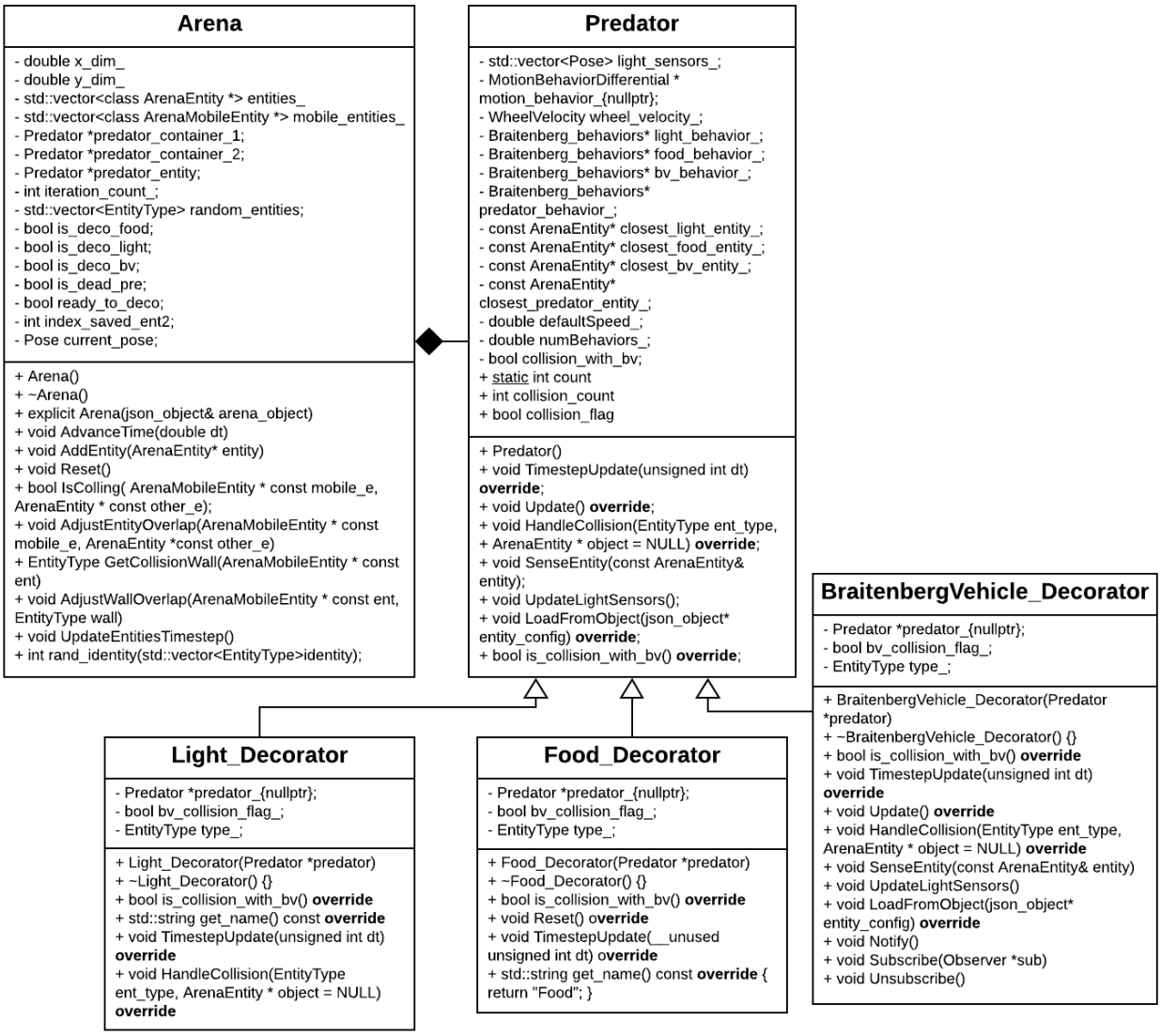


Figure 2: Inheritance Approach of Decorator Pattern (version 2)

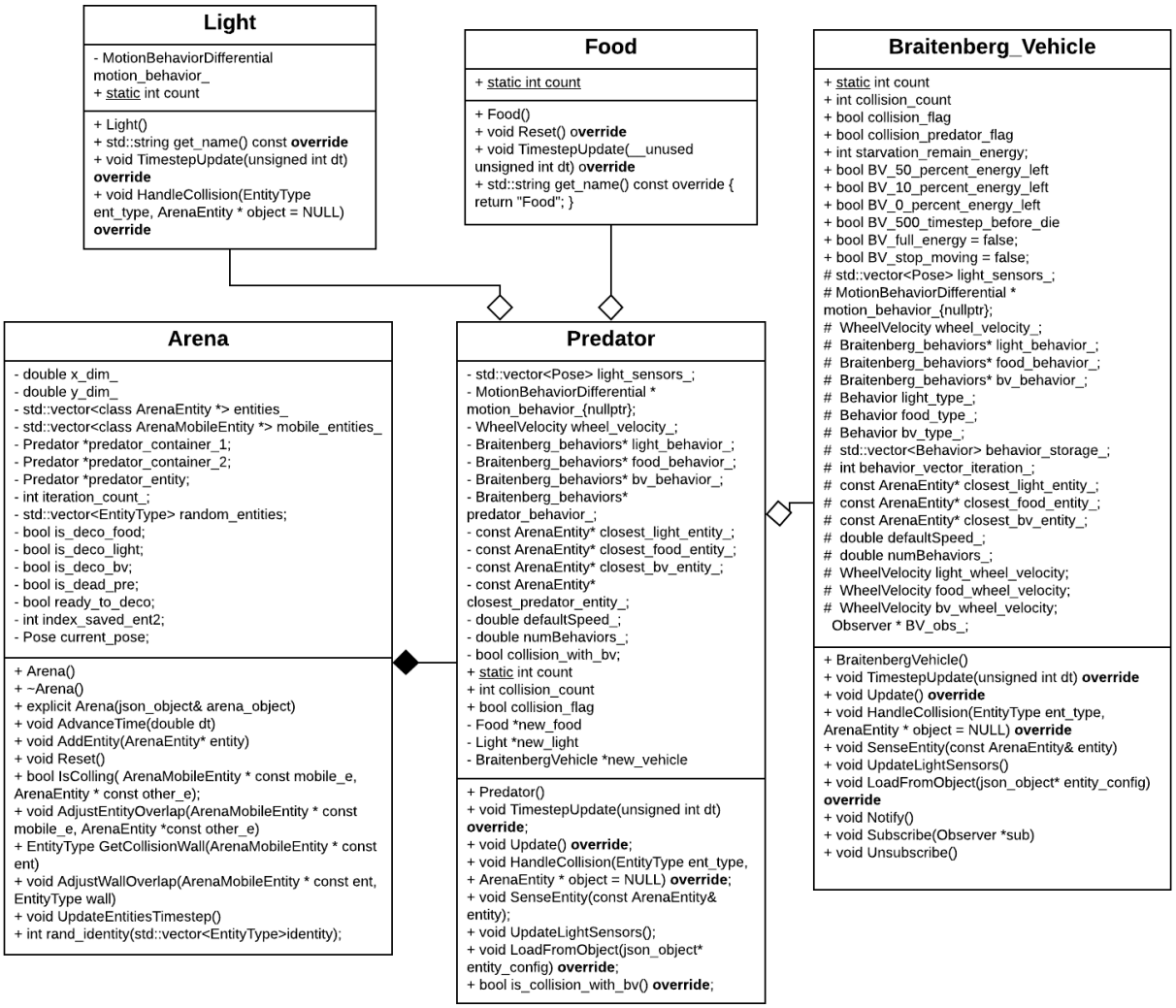


Figure 3: Aggregation Approach of Decorator Pattern

Final Factory Pattern

For factory pattern, there are three main commands that can use to configure Arena. The first command is "*./build/bin/arenaviewer xdim ydim file.csv*" which uses the CVS file and convert it back to JSON file using a function called apapt_csv in Controller class (figure 4). The result of adapt_csv function will use to set up for all entities in Arena. The second command is "*./build/bin/arenaviewer xdim ydim file.json*" which uses the JSON file to set up for all entities in Arena. The second way works by overriding the current value of "height" and "width" inside of JSON file with the input xdim and ydim from the user. With the third command of "*./build/bin/arenaviewer file.json*", this way is exactly the same with the second command except not override "height" and "width" inside of the JSON file. The other commands start with "*./build/bin/arenaviewer"* will set up Arena with the default setting.

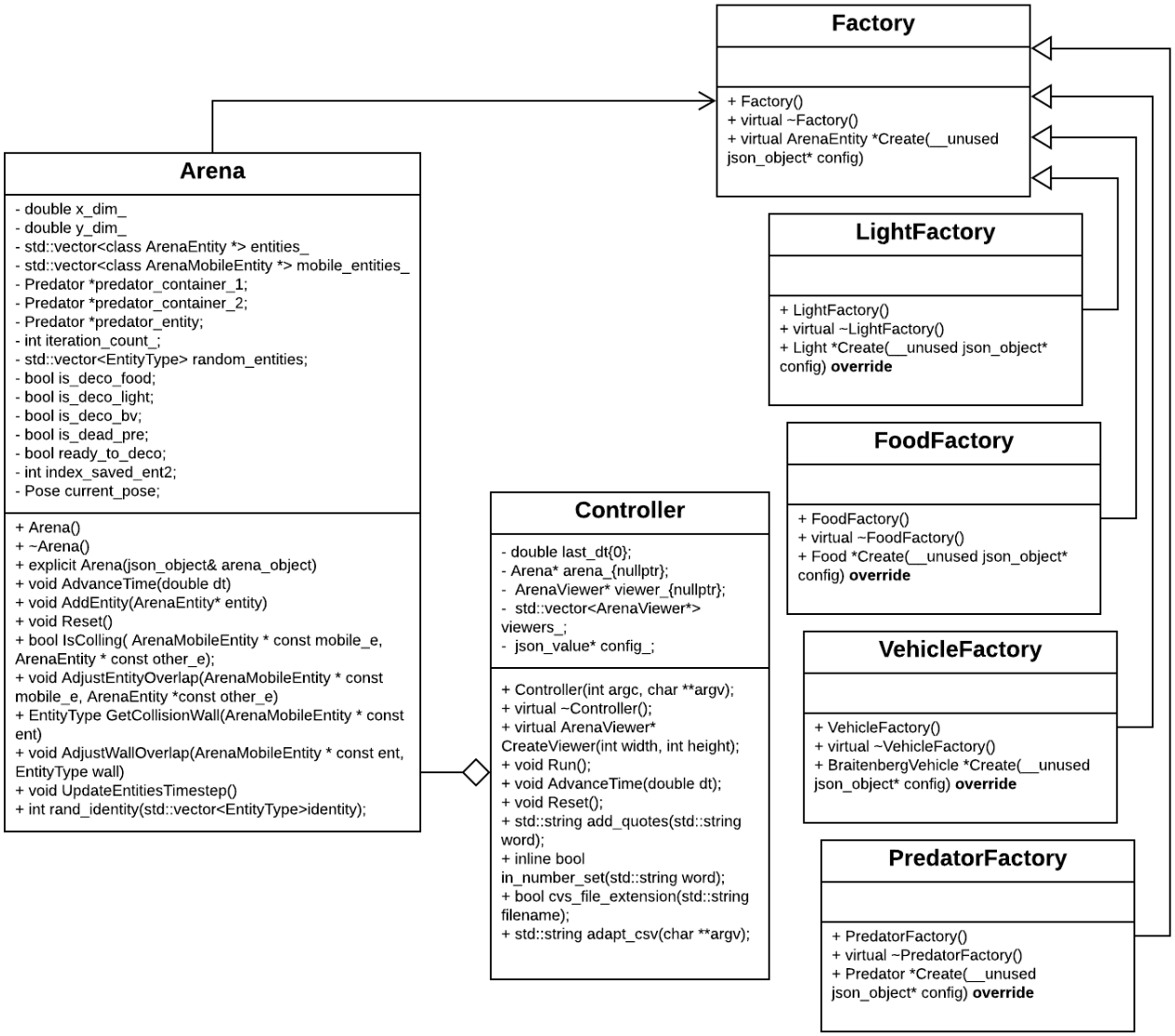


Figure 4: Factory with a conversion function inside of Controller

One advantage of this approach versus using the adapter pattern is the encapsulation of coding. All the code related to the CSV file and JSON file will be handled inside of Controller class which is easy to keep track of the flow of control of this factory. However, it added another responsibility of converting CSV file and set up the entities into the Controller class. This disadvantage can be solved by using the adapter pattern (figure 5). The Adapter pattern satisfies the single responsibility principle by separate the interface or data conversion code from the Controller class to another class. It also helps to improve the data protections because Arena doesn't know about the existence of the adapter class. The adapter pattern also has an advantage for later development of the project because we can introduce a new adapter class into the project without breaking or modifying any existing code. In the case of my approach, it is a different story. All the changed in Controller will directly affect to Arena, and it violated the "Open/Closed Principle" which means open for extension, but closed for modification. In spite of all the advantages of the adapter pattern, the overall complexity of the code increases can be one of the disadvantages of this pattern because a set of new interfaces and new classes needs to add

into project. It is simpler to do all conversion and set up for all entities in Controller class instead of creating new classes which can increase the potential of errors or bugs.

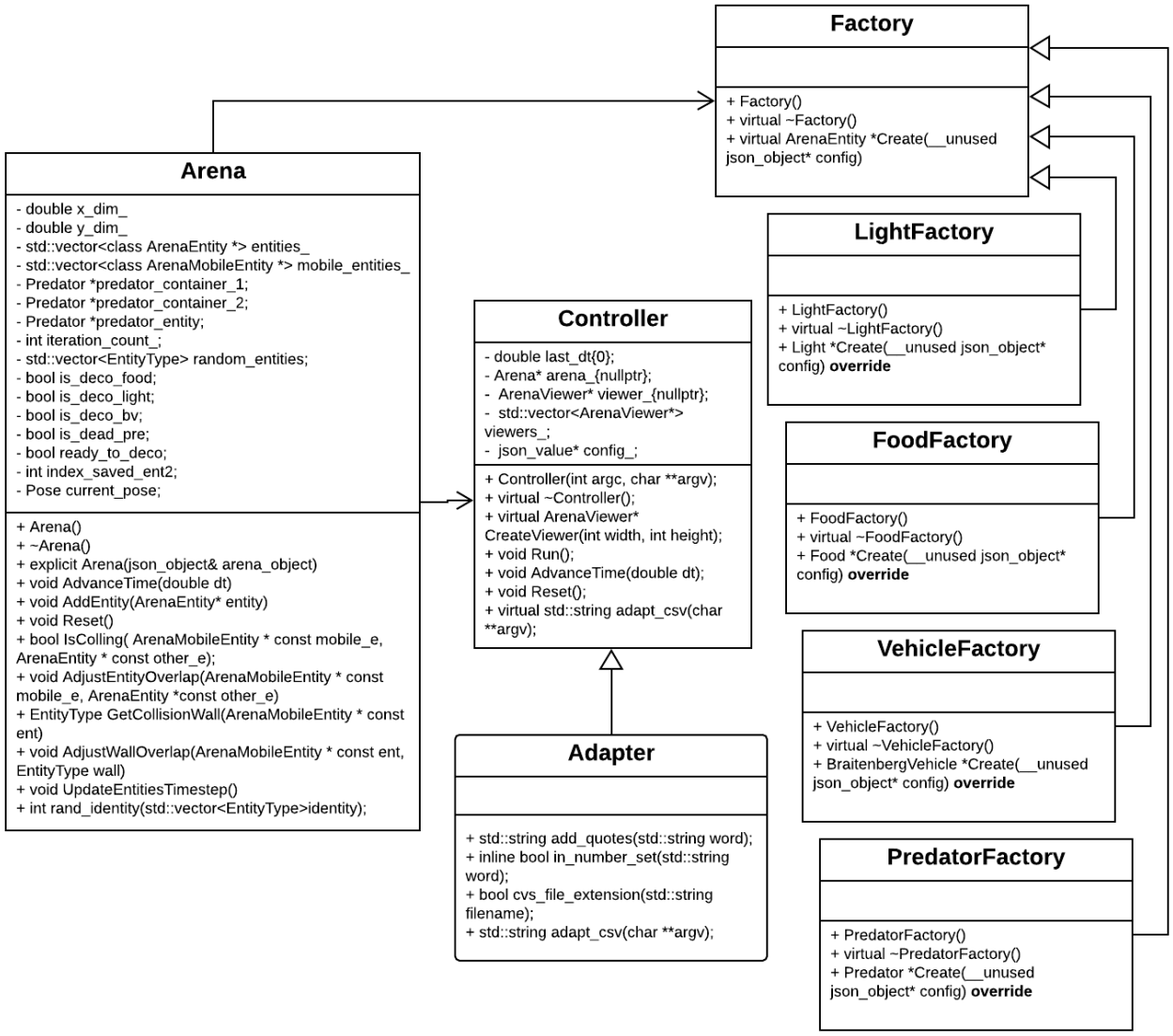


Figure 5: Factory with Adapter Pattern Design

After consideration on advantages and disadvantages of both the adapter pattern design and direct implement conversion function from CSV file or JSON file in Controller, I chose the second approach to keep the simplicity of the factory pattern and lower the chance of errors or bugs in the future.

Observer Pattern

I didn't change my observer pattern design (figure 6) on my final implementation because my design satisfied the requirements of observer pattern and also gained some advantages including loose-coupling and allows Braitenberg Vehicle to send WheelVelocity to GraphicsArenaViewer in a very efficient manner.

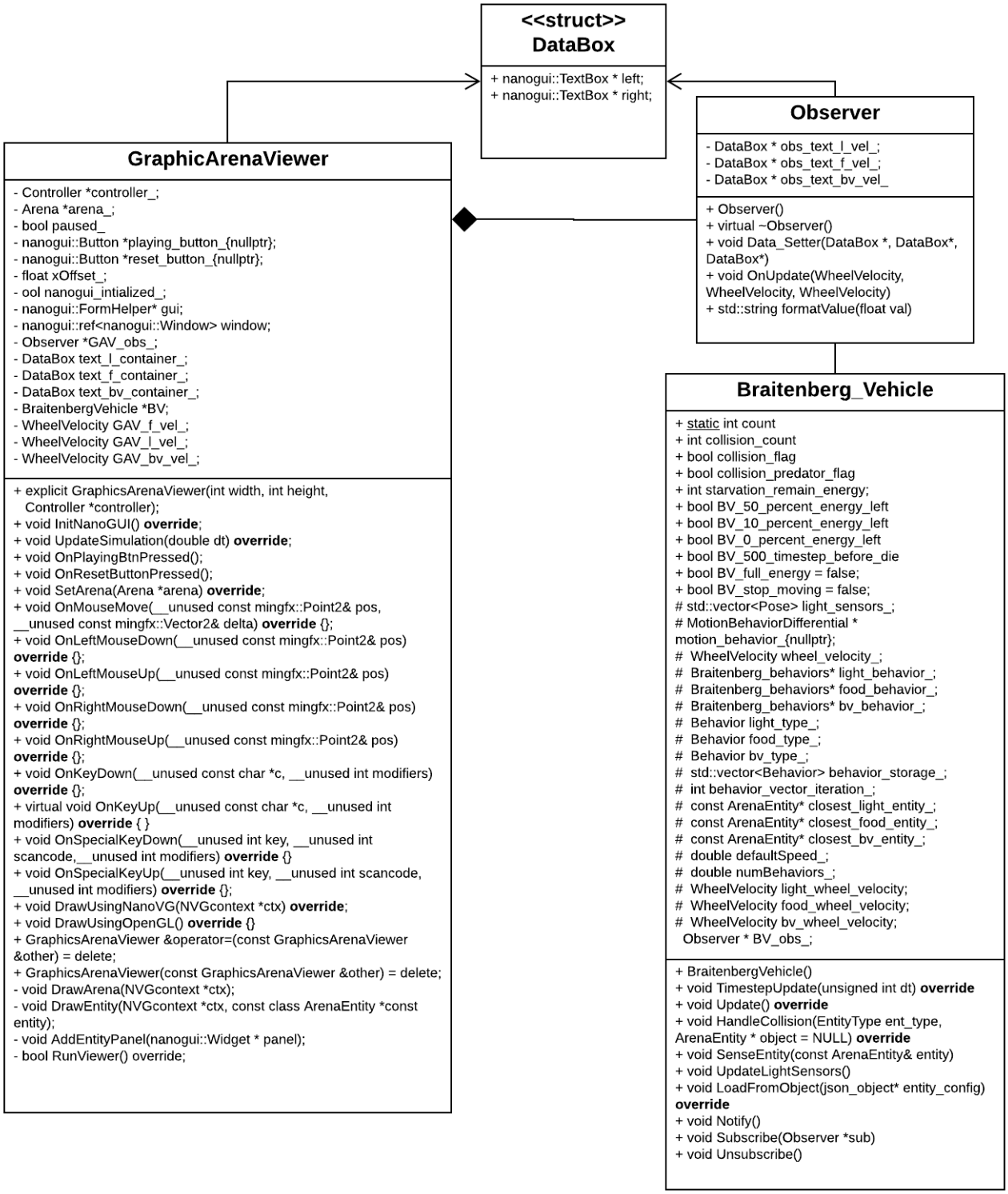


Figure 6: Final Observer Pattern Design

II. Iteration 2 Design Document

Observer Pattern

My approach to implement of the viewing of the wheel velocities from GraphicsArenaViewer to Braitenberg Vehicle is used the Observer pattern. In general, The Observer pattern defines as a one-to-many dependency between objects. When the subject changes state, then the subject notifies all its observers for the changing information. In this project, we use this pattern to send wheel velocity from BraintenbergVehicle to GraphicsArenaViewer instead of GraphicsArenaViewer constantly pulling information from BraintenbergVehicle.

I created an Observer class that contains three methods including Data_Setter, OnUpdate, and formatValue. The Data_Setter uses to set data after observe it from BraintenbergVehicle and store it into the corresponding private variable in Observer class. The OnUpdate uses to update the TextBox in GraphicArenaViwer each time BraintenbergVehicle call Notify method on the new change of WheelVelocity. The formatValue uses to convert float number back to a string, so we can set it into the TextBox.

In BraitenbergVehicle class, we have Subcribe, Unsubcribe, and Notify methods. The Subscribe uses to add a new observer, so it can update the Wheelvelocity of that observer later on. The Unsubscribe uses to remove an observer. The Notify uses to notify any change to the currently subscribe observer. This class associated with the Observer class.

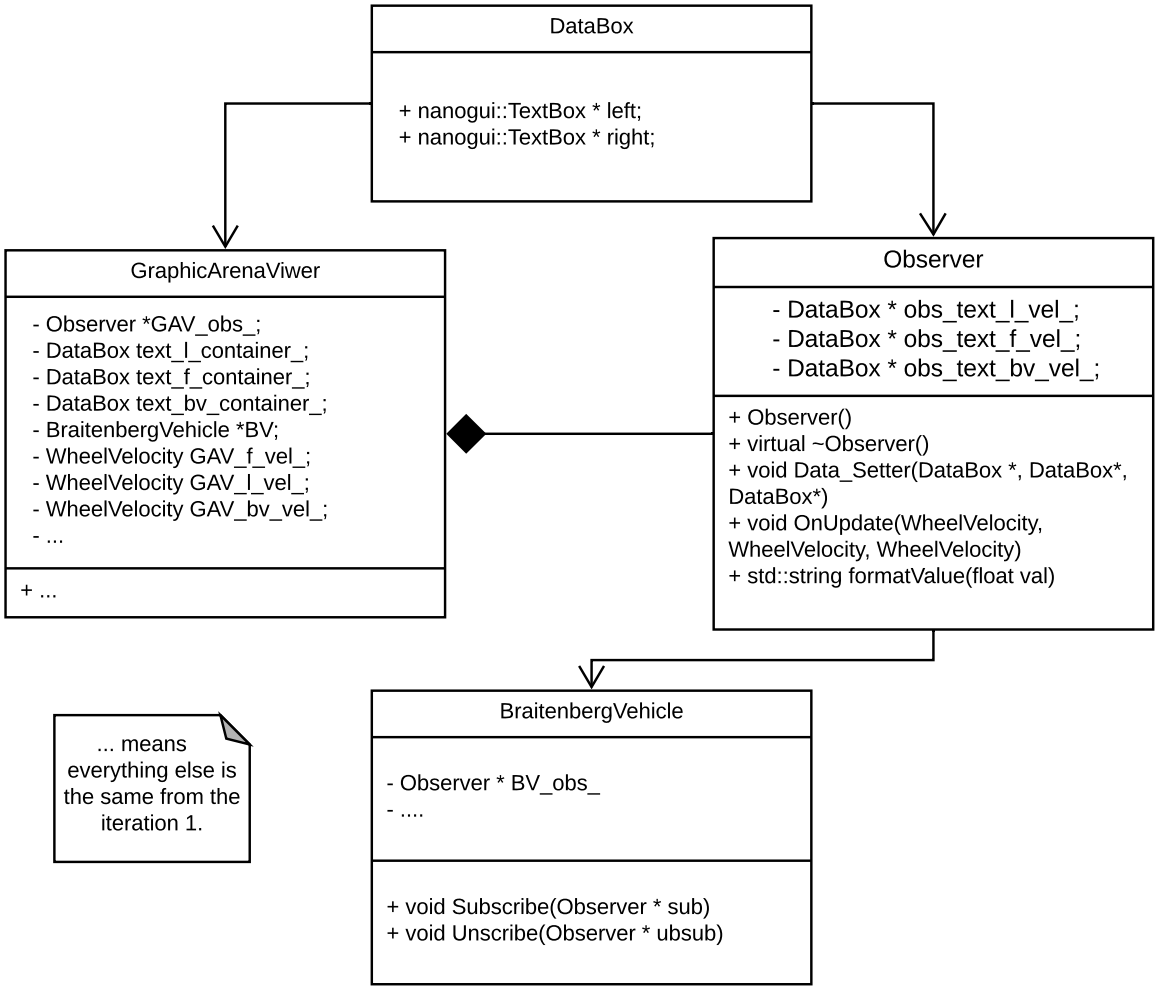
DataBox is a struct class that I added which contains the left and the right of the TextBox. GraphicArenaViwer has a composition of Observer class in it. It calls Subscribe and Unsubscribe for each observer.

With all those classes I named it above, when the WheelVelocity in BraitenbergVehicle class changed, it will call notify to the current observer. This will lead to updating all left and right of DataBox for Light, Food, and BraitenbergVehicle in the Observer class. Because the Observer class is a composition of GraphicsArenaViewer, the update of DataBox in the Observer will lead to the update of all Wheelvelocity TextBox in GraphicArenaViwer.

By using the Observer pattern, we gained some pros and also got some cons from this pattern. For pros, this pattern supports for loosely coupled design between objects that interact within the observer pattern. One example of this pro can be when we want to modify the Observer class, both BraintenbergVehicle and GraphicsArenaViewer do not need to change anything with help to lower the chance of errors and bugs.

The other pros that we can gain are the observer pattern allows us to send data to many other objects in a very efficient manner. This is proved by BraitenbergVehicle only send data to the Observer when Wheelvelocity updated. If we don't use the observer pattern, the GraphicsArenaViewer will need to constantly pull data from BraitenbergVehicle every iteration which is a waste of computer resources. Additionally, we can gain the advantage of no modification is needed to be done to the subject to add new observers or to remove observers at anytime. In this project, GraphicsArenaViewer can add and remove observer by reference the Subscribe and Unsubscribe methods of BraitenbergVehicle. In this way, BraintenbergVehicle doesn't need to modify anything for the new observer.

Beside of the pros, we still have one con that I face in the process of coding for this observer pattern, and that is if not correctly implemented, the observer can add complexity and lead to inadvertent performance issue. There are many versions to code for the observer pattern and each version has its own advantages and disadvantages. Because of these many versions, it got me confused from one version to another version and sometimes added unnecessary complexity to the code. It leads to the slowness of my old Observer pattern version performance.



Getter Methods

The other alternative that allows GraphicsArenaViewer can get wheel velocites from Braitenberg Vehicle is used getters method in BraitenbergVehicle. To implement this approach, all we need is create three getter methods which can get the Wheel velocity every time-step of BraitenbergVehicle.

```
// getter method to get the current food WheelVelocity
WheelVelocity food_velocity_getter () {
    return food_wheel_velocity;
}

// getter method to get the current light WheelVelocity
WheelVelocity light_velocity_getter () {
    return light_wheel_velocity;
}

// getter method to get the current BV WheelVelocity
WheelVelocity bv_velocity_getter () {
    return bv_wheel_velocity;
}
```

One advantage of this approach is so easy to implement and the amount of modify of code is small. All I need to add is three getter methods and a line of code in GraphisArenaViewer to call these method. One big disadvantage of this implement will be that the GraphicsArenaViewer need to call this method constantly no matter the BraitenbergVehicle has any changing of its wheel velocities or not. It will waste a lot of computer resources and not ideal way to implement for a big project.

III. Iteration 1 Design Document to Compare and Contrast Three Versions of The Factory Pattern

The instantiation of entities in the provided code is the simple way to code and that is the way I have been coded for the past two years. The first advantages that we can see from this strategy are easy to read and understand the code. Everything is built inside of one file which is easy to find and grab a function or a variable when needed. A good code flow is presented by combing all variables and classes in one constructor class instead of breadth out into many small classes like the factory pattern. Despite this advantage, there is a big disadvantage is the data protection and code reuse issue. A good example of this is in Arena's constructor (Code1 below). The arena can see and modify anything in the process of creating new entities for food, light, and Braitenberg's vehicle. This is not ideal for good software or may potentially create bugs in the future. If we use the factory pattern to create food, light, and Braitenberg's vehicle instead, Arena won't be able to directly manipulate any other classes; furthermore, it improves the loose coupling of the code. It is best to use this direct object construction when objects need to create is used only in one single place or the create logic is really simple.

```
Arena::Arena(json_object& arena_object): x_dim_(X_DIM),
    y_dim_(Y_DIM),
    entities_(),
    mobile_entities_() {
 x_dim_ = arena_object["width"].get<double>();
 y_dim_ = arena_object["height"].get<double>();
 json_array& entities = arena_object["entities"].get<json_array>();
 for (unsigned int f = 0; f < entities.size(); f++) {
  json_object& entity_config = entities[f].get<json_object>();
  EntityType etype = get_entity_type(
   entity_config["type"].get<std::string>());

  ArenaEntity* entity = NULL;

  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);
  }

  if (entity) {
   entity->LoadFromObject(entity_config);
   AddEntity(entity);
  }
 }
}
```

Code 1: Directly instantiate of Entities

The use of an abstract Factory class and derived factory classes has many advantages compared to the other two strategies. The first advantage is easy to add more features and allow the loose-coupling in the abstract class and derived classes. This pro is so important for a coder because any application or software always need to upgrade, improve, or add new features in the future. For instance of figure 1, what if we want to add a new feature for the factory to create a traffic stop sign so that a robot will stop if the distance between the robot and stop sign is ten iterations. It will be super easy if we use this strategy of abstract factory and derived factories. This is because all we need to do is add another derived class and override the Create function from Factory class to create the stop sign. Are the other two strategies can do it? Sure they can do it, but the amount of code that needs to be modified is way larger compared with the abstract Factory strategy. Another important advantage of this method can be unit-testability. By implement each class separate from the entity_factory, we have the ability to write an unit-test class for each class and test for functionality, bugs, limitation, or test for meeting our expectations for each class. Next advantage will be the ability to reduce the number of dependencies per class. With the separation of the big factory into many small factories to do a specific job, each class now only required to include the necessary header files. Besides all these advantages, there are some disadvantages that we need to consider. When we use this method to code, it adds more complexity to the code. Instead of showing data and classes in one code, we create a new class to hide all data and the implementation which can cause the readers confused about the connection and usage of the code flow. A problem that I had in this robot project is that it's so hard to find where is a function definition is and why it got to call inside of this function or class. Then I have to look at all the header files and find that function which sometimes annoys and confuse me. This issue can be solved by a good detail UML graph and adding a good document on where this function from and what is that function will do.

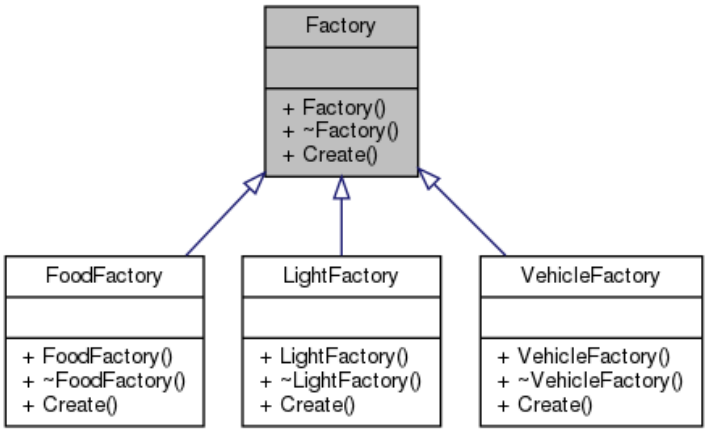


Figure 7: An Abstract Factory class and Derived Factories Class

Another strategy to create entities can be the use of a concrete Factory class that is responsible for the instantiation of all Entity types. This method is better when compared with the instantiation of entities in the provided code but still worse when compared with the derived class from the factory. It still gains some of the level protection of the private data but not as much as building a big abstract factory and derive it into many small factory classes. For example (figure 2), if Arena wants to create any objects using factory class, it can call the factory inside of its code to create objects. In this way, Arena can't directly manipulate any private data inside of the factory but the Factory class can. In contrast with the advantage, It is not loose-coupling because changing one part of the factory code may cause of changing the other parts of this class.

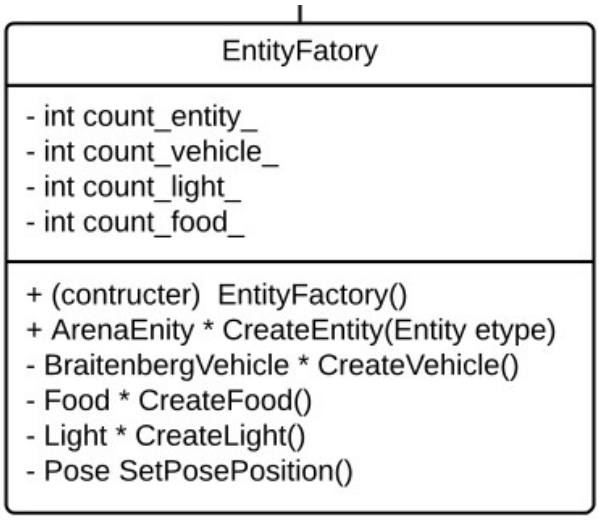


Figure 8: Concrete Factory Class

In conclusion, factory pattern design is a great way to create and control on certain criteria what objects should be created, so it is easy to maintain the factory in one place, instead of breath out throughout the whole project. Both abstract and concrete Factory design both have the ability to hide information, and keep classes loosely coupled; however, the extensibility is only provided by the abstract version of Factory design. With all the advantages of the abstract version of the Factory pattern design, I decided to choose this design for my Factory.