Design Document

Prepared by Moti Begna

University of Minnesota

CSCI 3081W

April 13th, 2019

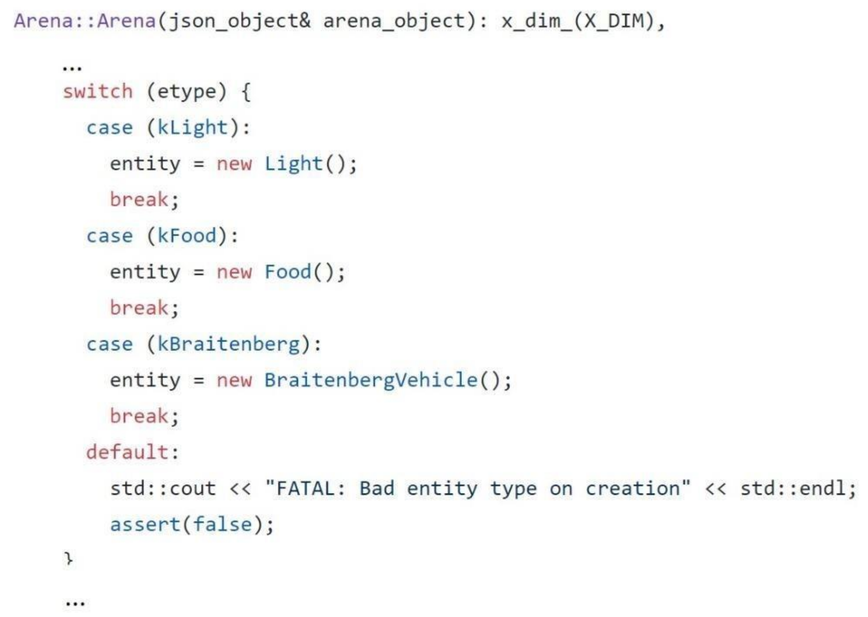
Table of Contents

1. Factory Pattern

1.1 The Instantiation of Entities Provided in the Original Code

Description: In this approach, entities will be created as the are in the original code. Specifically, the Arena class will have access to the Light, Food, and BraitenbergVehicle classes, instantiating them based on the configuration file passed in as a parameter.

Within arena.cc



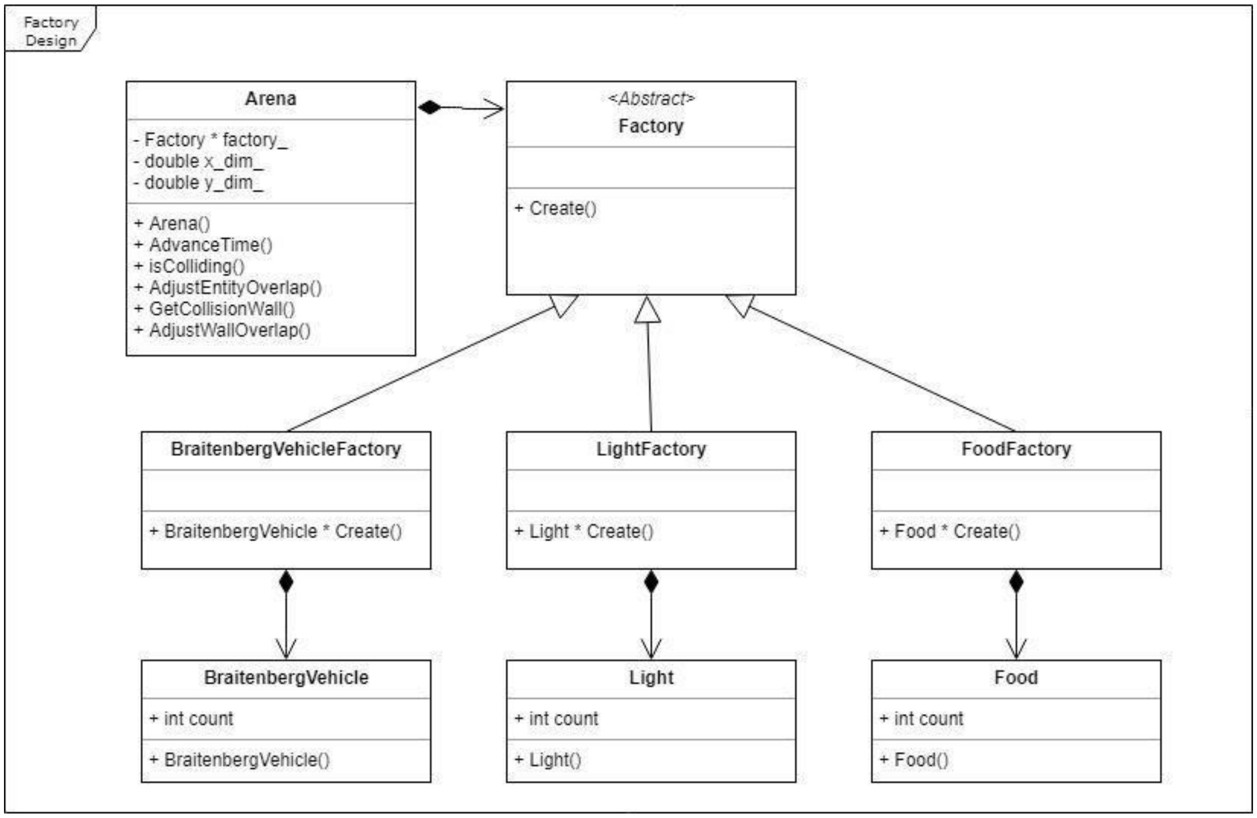
```
Arena::Arena(json_object& arena_object): x_dim_(X_DIM),

    ...
    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);
    }
    ...
```

Advantage: The instantiation of entities within the constructor of arena.cc allows for simplicity throughout the entirety of the entity creation routine. This ensures that any other developers that may want to understand how the system works are allowed to do so without information being hidden through various levels of abstraction that may have otherwise complicated the entire process.

Disadvantage: This approach creates many disadvantages to the entity creation routine. Firstly, this process chases simplicity but falls behind to flexibility. While the routine is easier to understand when laid out within a single constructor, the process doesn't allow for any feature extensions/enhancements for how each individual arena entity may be instantiated later in the development process. Secondly, encapsulation and information hiding is non-existent seeing as how the entire creation process for all of the various entities are visible within a single constructor.

1.2 The Use of an Abstract Factory Class and Derived Factories

Description: In this approach, an abstract Factory class is created, while the arena class contains the derived factory classes BraitenbergVehicleFactory, LightFactory, and FoodFactory. Each derived factory instantiates its respective entity based on a configuration file that is passed in as a parameter to each factory classes' Create() method.

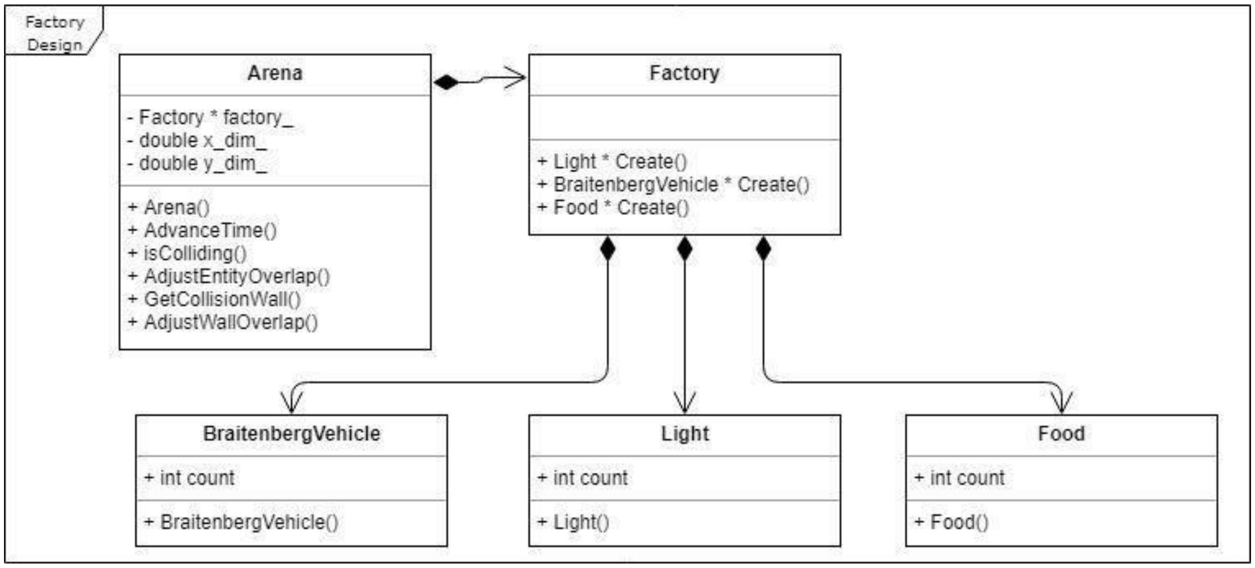


Advantage: By utilizing this factory strategy, the arena class itself becomes unaware of the implementation of various entities within the arena, which not only keeps information hidden and data protected, but also allows the entities themselves to be open to future extension. This pattern also ensures that classes are loosely coupled meaning that by separating the creation of arena entities through factories, the 'entity creation' routine allows for small, direct, visible, and flexible class relationships. In addition, by creating separate factories for the various arena entities that inherit from a base abstract factory, classes are kept strongly cohesive in that all of the different routines (such as the creation of a Braitenberg object, a Light object, and a Food object) are situated within a respective factory class where all members of that class support the same purpose.

Disadvantage: By utilizing this factory strategy, it can become harder for developers to follow the subroutine of the creation of arena objects seeing as how the code is hidden behind different levels of abstraction. In essence, simplicity is being given up in exchange for flexibility.

1.3 The Use of One Factory Class That Instantiates All Entities

Description: In this approach, only one Factory class exists which is contained within the Arena class. This Factory is in charge of instantiating all of the entities by contain three separate Create() methods that return each of the entities.



Advantages: While this approach is fairly similar to the previous factory strategy in which a factory class is responsible for the instantiation of entities, this version helps the main issue of its predecessor: the routine is kept simpler. By having a single factory class in charge of the instantiation of all Entity types, the process removes a level of abstraction that might have otherwise complicated the overall routine.

Disadvantage: Because a level of abstraction is lost in this implementation, we lose the advantage of encapsulation in that expectations to how we instantiate one entity type may cause issues with the entirety of the factory routine. In addition, we also give away how the various subroutines of each entity type work meaning we lose the ability to utilize good design through information hiding.

1.4 Version Implemented

The factory pattern implemented for this project was the one detailed in approach 2, seeing as how the advantages of utilizing this implementation outweighed the disadvantages that came with it. While increasing the levels of abstraction between interacting classes can lessen the simplicity of the overall structure, encapsulation and data protection is an important essence of software design that must always be sought after.

1.5 CSV Functionality Implementation

1.5.1 Conversion of a .csv Configuration File to a Json Configuration

Description: To allow a .csv configuration file to be used in the creation of arena entities by factories, this approach simply converts the file to a json configuration within the Controller constructor. The command-line argument representing the configuration file is read and checked to see if it is of filetype .json or .csv. If it is the latter, the configuration is used as it has previously been used. If the configuration file is of type .csv however, a json configuration is populated by the keys and values representing the various entities within the file.

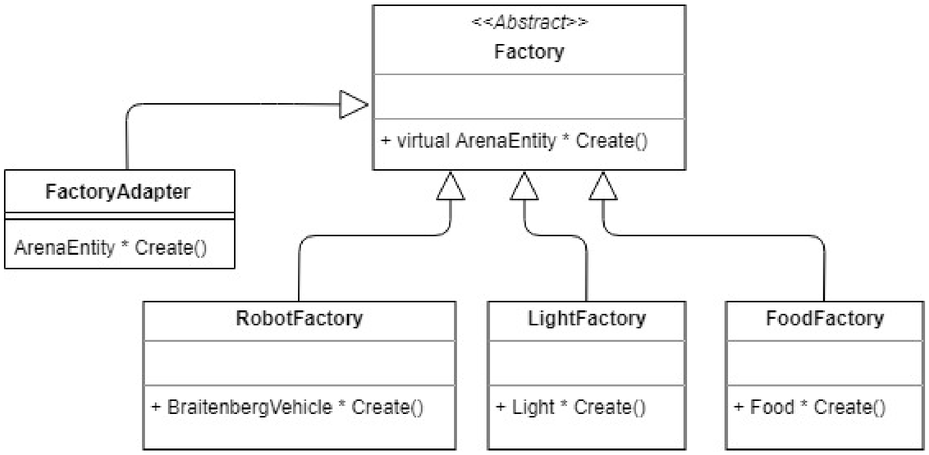
```
Controller::Controller(int argc, char **argv) :
  last_dt(0), viewers_(), config_(NULL), xdim_(600), ydim_(600) {
    …
    std::string file(argv[3]);
    if (file[file.length() - 4] == 'j') {
      std::ifstream t(std::string(argv[3]).c_str());
      std::string str((std::istreambuf_iterator<char>(t)),
                      std::istreambuf_iterator<char>());
      json = str;
    } else {
      /* Conversion takes place */ }
    config_ = new json_value();
    std::string err = parse_json(*config_, json);
    …
```

Advantage: This implementation allows for the integration of a new configuration filetype to be kept simple. Here, all of the code for conversion is not only kept in a single class, but within the Controller constructor, maintaining that simplicity.

Disadvantage: The primary issue with this approach is that the Controller classes has lower cohesion overall. An argument can be made that it should not be the job of the Controller to deal with how various configuration filetypes are read, and that another class should instead be created to adapt to various input scenarios.

1.5.2 The Use of the Adapter Pattern

Description: In this approach, and adapter class FactoryAdapter is created which inherits from the adaptee class Factory. This allows a .csv file to be used as a configuration file by converting the FactoryAdapter class's interface into the interface of the original Factory class which uses a json configuration to instantiate arena entities.



Advantage: The primary advantage of using this approach is that the Adapter Pattern employs the use of loose coupling and high cohesion in its implementation. Instead of having the Controller in charge of the subroutine of adapting a .csv file to be used by the Factory class, a separate class is used which is not visible to the Controller, ensuring that the connections between the classes are kept small, direct, and flexible.

Disadvantage: The most glaring issue when using the adapter pattern in this manner is that the adapter itself may have access to the internal implementations of entity creation within the Factory class. In addition, because we are only using two configuration files of different types, having this amount of abstraction can be deemed unnecessary since conversion of a .csv file to a json configuration can be implemented simply as with the previous approach.

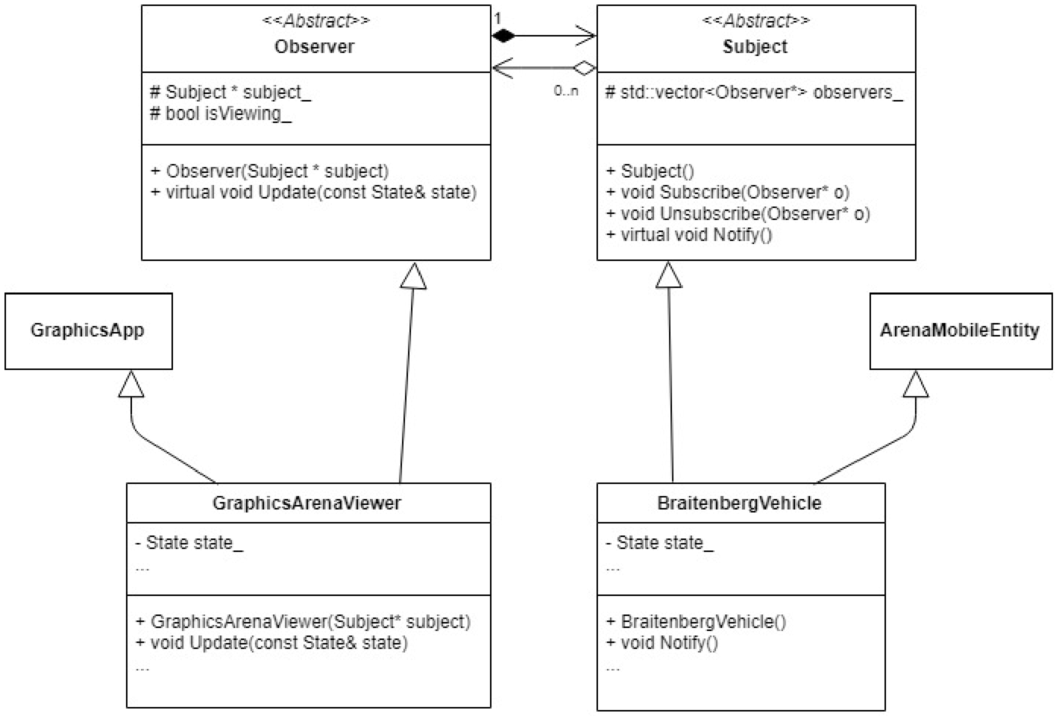
1.5.3 Version Implemented

For this project, the first approach was utilized based solely on its simplicity. As previously stated, the conversion of a .csv file to a json configuration would not require much in the way of changes/additions, thus having the Controller be in charge of the conversion allows ease of understanding in the process.

2. Observer Pattern

2.1 The Use of Abstract Observer and Subject Classes

Description: In this approach, an abstract Observer and Subject class is created in which the GraphicsArenaViewer and BraitenbergVehicle classes inherit from respectively. This also entails that double inheritance would occur in the aforementioned classes. In any case, the abstract Subject class contains a vector of Observer objects, which is a collection of all of the observers that the subject will notify whenever the subject updates it's contained State member variable. When notify is called, the State variable is passed to the Observer's Update method, which updates its own State member variable. In addition, Observers can subscribe an unsubscribe from a subject.

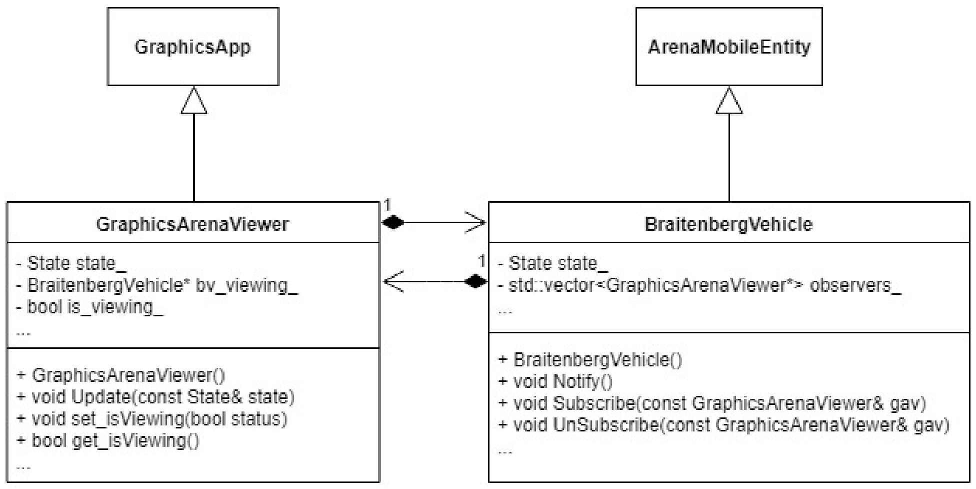


Advantage: One of the key advantages of using this approach is the increased levels of abstraction, which allow for information hiding and data encapsulation of the derived Observer and Subject classes. The GraphicArenaViewer doesn't have direct access to the BraitenbergVehicle class, which ensures that the classes are loosely coupled which is an important consideration in design. This approach also allows for easier extension in the future. Specifically, if we ever want to add more subjects and/or observers, they would simply have to inherit from the abstract Observer and Subject classes.

Disadvantage: The most glaring disadvantage of this approach is the fact that double inheritance cannot be avoided. While double inheritance is not inherently unfavorable, function overriding can play a disastrous role in future extensions. For example, if we ever decided to add similarly named methods and/or variables to either parent classes of the GraphicsArenaViewer and the BraitenbergVehicle classes, and those additions have not been overridden by the children, they compiler will have issues discerning which methods/variables to use if they are called elsewhere.

2.2 The Use of Observer Pattern Methods in Established Code

Description: In this approach, double inheritance is avoided by simply implementing the Observer Pattern methods into the established GraphicsArenaViewer and BraitenbergVehicle classes. Here, a BraitenbergVehicle contains a vector of GraphicsArenaViewer objects, whose Update methods are called whenever the BraitenbergVehicle calls its Notify method after changing its own State member variable. In a similar fashion as before, the State variable is passed to the Update method of the GraphicsArenaViewer, which is used to change the class's own State member variable.


GraphicsApp
ArenaMobileEntity
GraphicsArenaViewer
- State state_
- BraitenbergVehicle* bv_viewing_
- bool is_viewing_
...
+ GraphicsArenaViewer()
+ void Update(const State& state)
+ void set_isViewing(bool status)
+ bool get_isViewing()
...
1
1
BraitenbergVehicle
- State state_
- std::vector<GraphicsArenaViewer*> observers_
...
+ BraitenbergVehicle()
+ void Notify()
+ void Subscribe(const GraphicsArenaViewer& gav)
+ void UnSubscribe(const GraphicsArenaViewer& gav)
...

Advantage: Since double inheritance is no longer occurring, there is no need to consider the problem of function overriding in later extensions. Any additions to the GraphicsApp and ArenaMobileEntity classes ensure that their children will only inherit a single instance of the new additions. Another advantage is the simplicity of the approach. In this project, we know that the GraphicsArenaViewer is the only observer and the BraitenbergVehicle is the only subject. Thus, it is simpler to integrate what would have the methods and variables of an abstract Observer and Subject class into the GraphicsArenaViewer and BraitenbergVehicle classes respectively.

Disadvantage: The major disadvantage of this approach is the fact that we loose levels of abstraction that would have otherwise insured that data encapsulation would be maintained. Specifically, because the GraphicsArenaViewer contains instances of a BraitenbergVehicle object—which also means that the classes are strongly coupled—it also has access to all of the methods of the BraitenbergVehicle class. This can cause issues with how BraitenbergVehicle objects are used if we are not careful about how we use the objects in the GraphicsArenaViewer.

2.3 Version Implemented

The observer pattern implemented for this project was the one outlined in approach 2, considering the simplicity of it. As mentioned before, we know that the GraphicsArenaViewer and BraitenbergVehicle classes are the only observers and subjects of the simulation. And while abstraction is lost and data usage is less secure, careful considerations of how this approach is implemented can combat these issues.

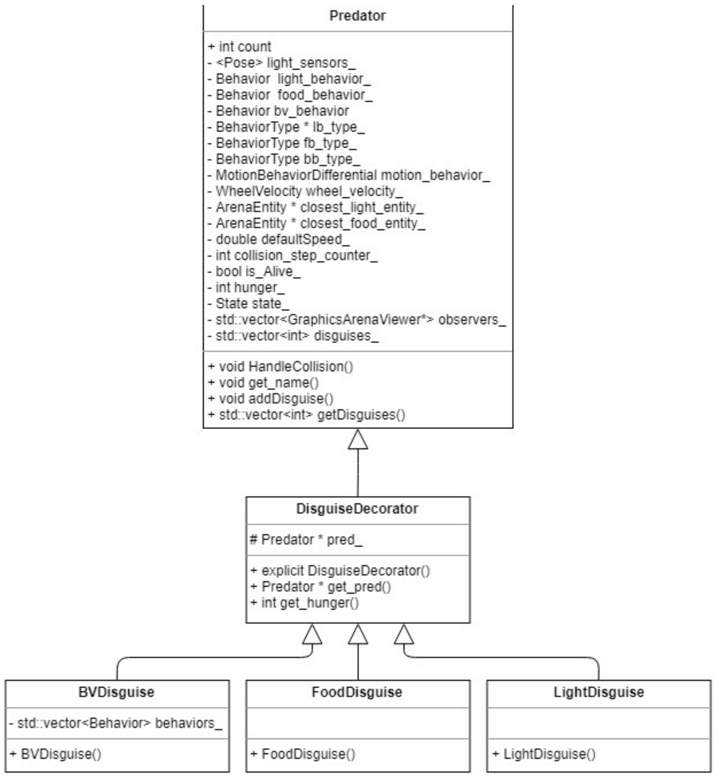
2.4 Final Version Differences

My final implementation of the Observer Pattern included no changes from my original design. Implementation still occurs within the GraphicsArenaViewer and BraitenbergVehicle classes since functionality still performs correctly.

3. Decorator Pattern

3.1 The Use of a Parent DisguiseDecorator class and Derived Classes

Description: In this first implementation, the decorator pattern is implemented by having a parent DisguiseDecorator class inherit from the Predator class. Different disguise classes (BVDisguise, FoodDisguise, LightDisguise) inherit from this parent class, and are instantiated within Arena::UpdateEntitiesTimestep to disguise the Predator entity at random. This occurs by simply passing in the entity, which is then contained within the parent DisguiseDecorator class. While the predator is disguised, it’s interactions within the Arena ensures that the entity is viewed as it’s respective disguised entity, either as a Light, Food, or Braitenberg Vehicle.

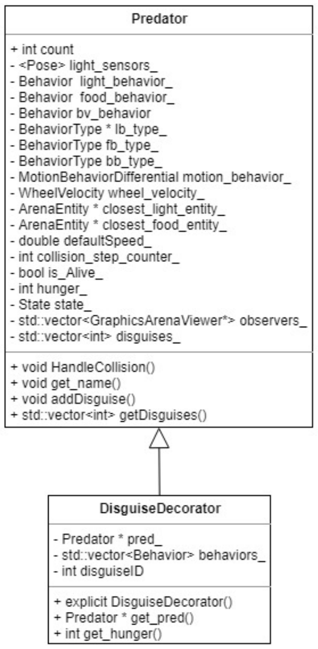

Predator
+ int count
- <Pose> light_sensors_
- Behavior light_behavior_
- Behavior food_behavior_
- Behavior bv_behavior
- BehaviorType * lb_type_
- BehaviorType fb_type_
- BehaviorType bb_type_
- MotionBehaviorDifferential motion_behavior_
- WheelVelocity wheel_velocity_
- ArenaEntity * closest_light_entity_
- ArenaEntity * closest_food_entity_
- double defaultSpeed_
- int collision_step_counter_
- bool is_Alive_
- int hunger_
- State state_
- std::vector<GraphicsArenaViewer*> observers_
- std::vector<int> disguises_
+ void HandleCollision()
+ void get_name()
+ void addDisguise()
+ std::vector<int> getDisguises()
DisguiseDecorator
# Predator * pred_
+ explicit DisguiseDecorator()
+ Predator * get_pred()
+ int get_hunger()
BVDisguise
- std::vector<Behavior> behaviors_
+ BVDisguise()
FoodDisguise
+ FoodDisguise()
LightDisguise
+ LightDisguise()

Advantage: The main advantages to this implementation is that the pattern ensures that the code can be closed to change, but open to extension. Essentially, if a new entity is introduced within the Arena, a new disguise class which would inherit from DisguiseDecorator can simply be created without the need to drastically change the old implementation. Overall, this implementation lends itself to flexibility which is important for design.

Disadvantage: While flexibility is maintained in this approach, ease of understanding can be lost when trying to follow the subroutine of disguising a Predator. As we have seen time an time again, increased levels of abstraction would also entail that any developers that may want to view the implementation may find it hard to navigate, whereas they might have preferred simplicity.

3.2 The Use of One DisguiseDecorator class

Description: This implementation is similar to the previous, however disguises for a Predator entity are maintained within a single class rather than deriving various other disguise classes. A predator entity would still be passed as a parameter in the instantiation of a DisguiseDecorator, however a disguiseID would also be sent which identifies the specific disguise that the predator would maintain. This id would be used to decide whether the disguised predator would behave as a Light, Food, or Braitenberg Vehicle entity.


Predator
+ int count
- <Pose> light_sensors_
- Behavior light_behavior_
- Behavior food_behavior_
- Behavior bv_behavior
- BehaviorType * lb_type_
- BehaviorType fb_type_
- BehaviorType bb_type_
- MotionBehaviorDifferential motion_behavior_
- WheelVelocity wheel_velocity_
- ArenaEntity * closest_light_entity_
- ArenaEntity * closest_food_entity_
- double defaultSpeed_
- int collision_step_counter_
- bool is_Alive_
- int hunger_
- State state_
- std::vector<GraphicsArenaViewer*> observers_
- std::vector<int> disguises_
+ void HandleCollision()
+ void get_name()
+ void addDisguise()
+ std::vector<int> getDisguises()
DisguiseDecorator
- Predator * pred_
- std::vector<Behavior> behaviors_
- int disguiseID
+ explicit DisguiseDecorator()
+ Predator * get_pred()
+ int get_hunger()

Advantage: While the previous approach focuses on flexibility, this one fixes the issue of simplicity. Rather than piling levels of abstraction to implement a disguise functionality, a single class that is in charge of changing how a predator interacts within the Arena helps in understanding how the subroutine works.

Disadvantage: Because abstraction is lost with this implementation, future extensions would prove to be more tedious as the DisguiseDecorator class itself would have to change in order to adapt the new additions. This process has the possibility of cascading further errors within the entirety of the code if not done properly, and so more time will have to be spent to ensure that this does not happen.

1.4 Version Implemented

The Decorator Pattern implemented for this project was the one outlined by the first approach. While simplicity may be a disadvantage of this implementation, the possibility of extension is a crucial idea that must be taken into consideration during the development process. Thus, abstraction in the hopes of overall flexibility must not be avoided simply because ease of understanding may be difficult upon future inspections.