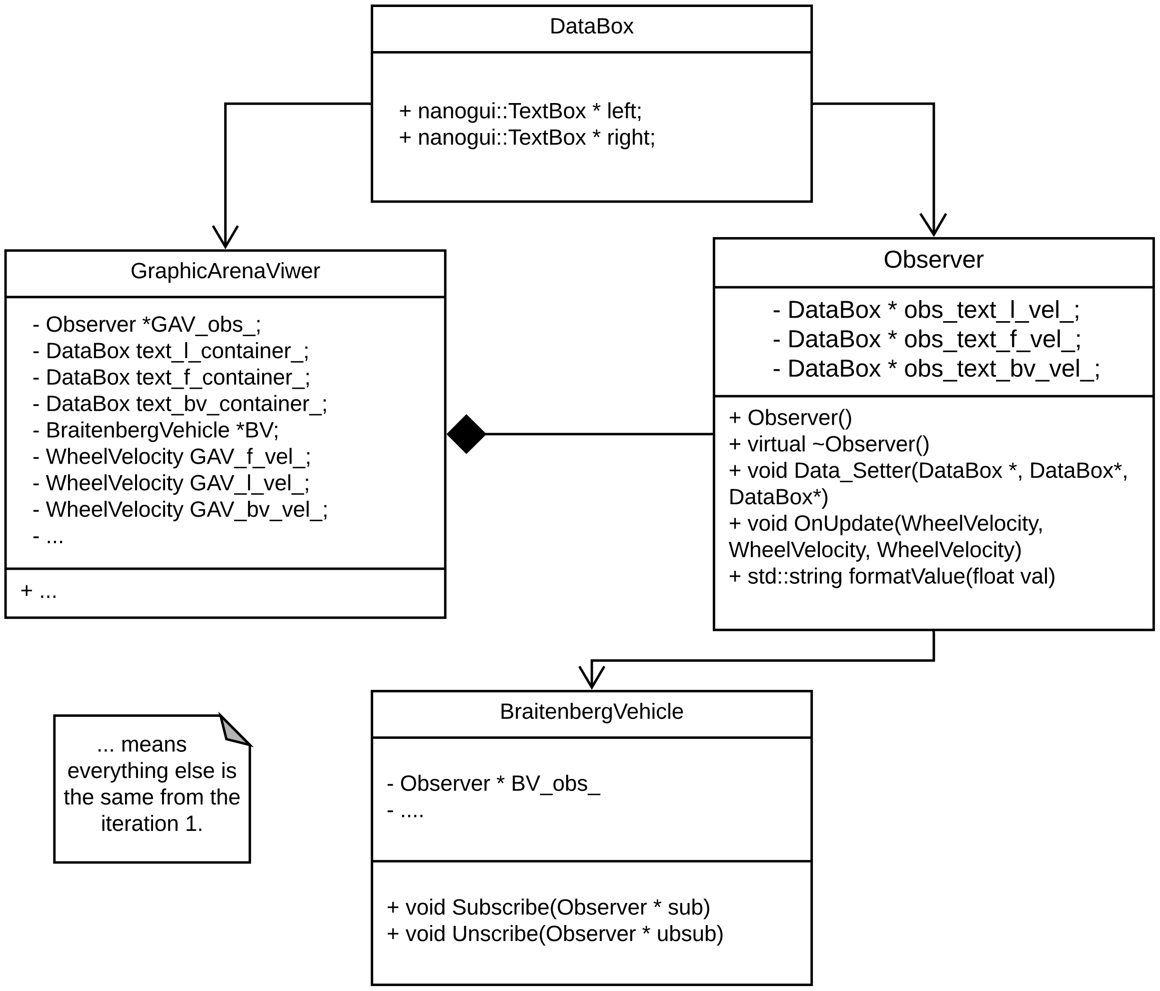
Name: Khoa Tran
x500: tran0707
Class: Csci 3081W

Observer Pattern Vs Getter Methods

Observer Pattern

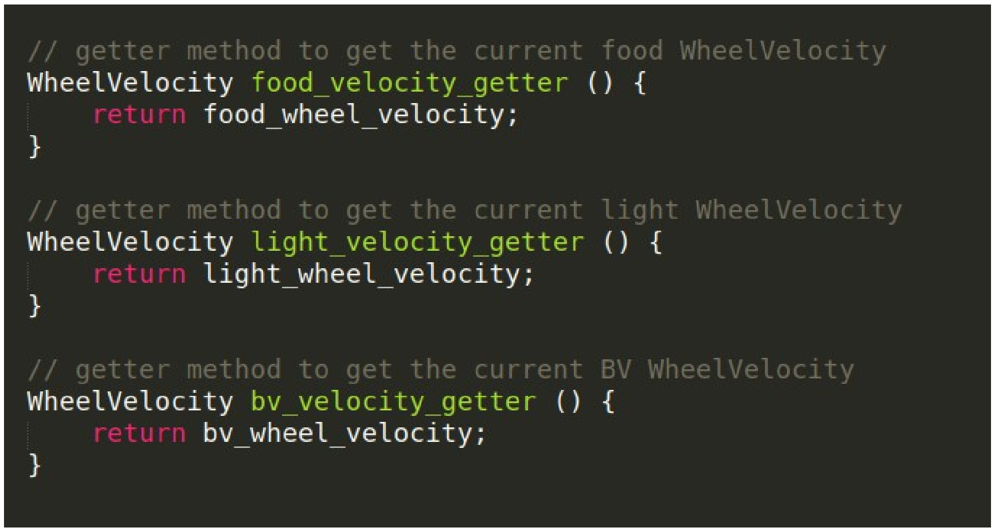
My approach to implement of the viewing of the wheel velocities from GraphicsArenaViewer to Braitenberg Vehicle is used the Observer pattern. In general, The Observer pattern defines as a one-to-many dependency between objects. When the subject changes state, then the subject notifies all its observers 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, BraitenbergVehicle 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.


DataBox
+ nanogui::TextBox * left;
+ nanogui::TextBox * right;
GraphicArenaViwer
- Observer *GAV_obs_;
- DataBox text_l_container_;
- DataBox text_f_container_;
- DataBox text_bv_container_;
- BraitenbergVehicle *BV;
- WheelVelocity GAV_f_vel_;
- WheelVelocity GAV_l_vel_;
- WheelVelocity GAV_bv_vel_;
- ...
+ ...
Observer
- DataBox * obs_text_l_vel_;
- DataBox * obs_text_f_vel_;
- DataBox * obs_text_bv_vel_;
+ Observer()
+ virtual ~Observer()
+ void Data_Setter(DataBox *, DataBox*, DataBox*)
+ void OnUpdate(WheelVelocity, WheelVelocity, WheelVelocity)
+ std::string formatValue(float val)
... means everything else is the same from the iteration 1.
BraitenbergVehicle
- Observer * BV_obs_
- ....
+ void Subscribe(Observer * sub)
+ void Unscribe(Observer * ubsub)

<u>Getter Methods</u>

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.