Laboration 4: Designmönster och refactoring

1)

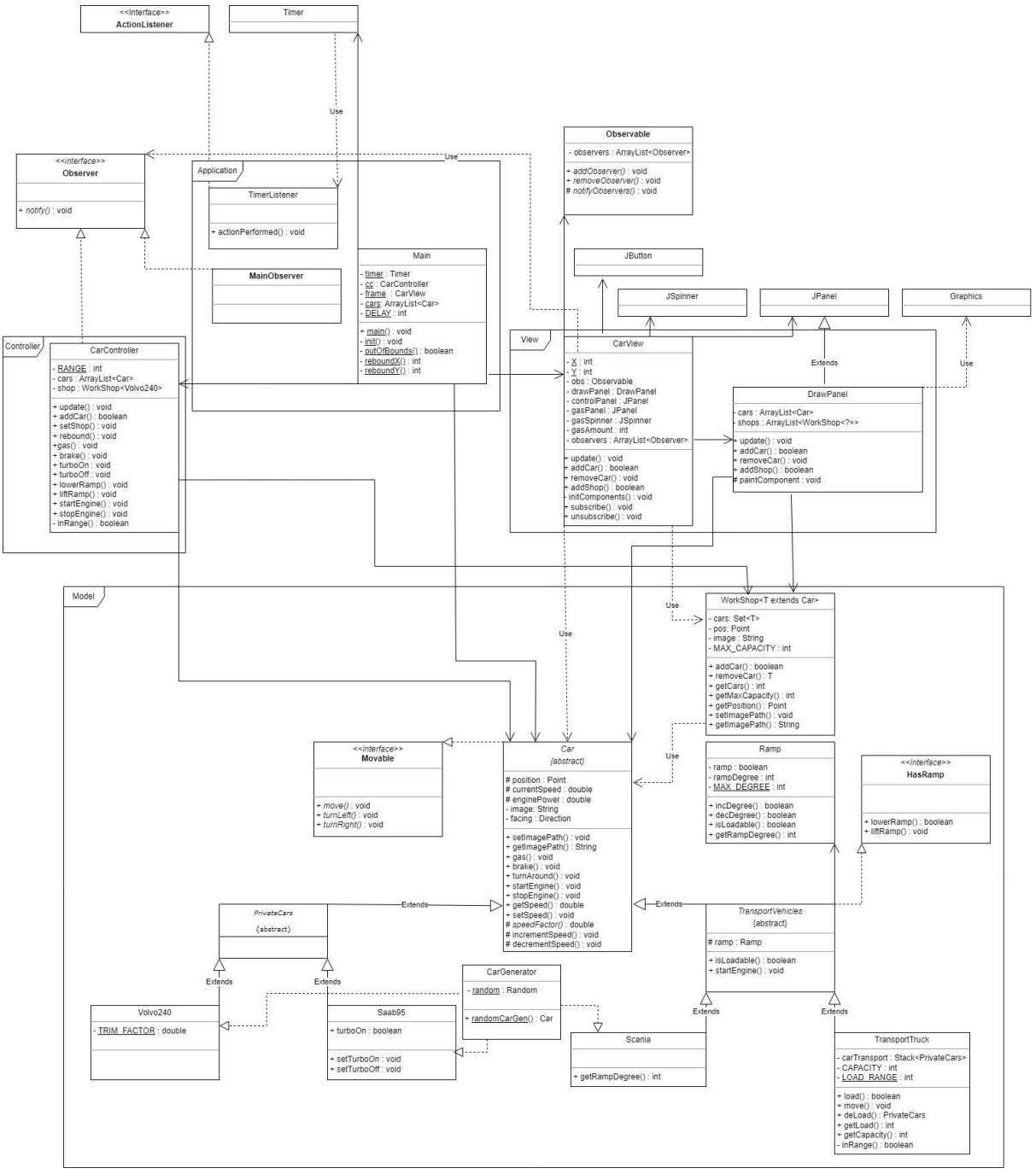
-

2)

The Controller was also handling communication with the View. These modules should really be decoupled as much as possible so that they can be exchanged without impacting the other.
The Model should have been "smarter", it should have offered an option for linking cars with images. We added a field and getter and setter methods to the Car class so that it can store a path to its image now.
The View should have been "dumber", instead of calculating where to move an image, it should have simply relied on the model to read where to put the image. We removed the moveit function, simply relying on repainting. To facilitate this, the DrawPanel now stores a list of cars. From Car it reads the image path - if one exists - and places the image at the position of the car.
The CarController should have been "thinner": Instead of handling running the entire Application and communication with the View, the Controller should only offer a "thin" layer between the Model and the Application. We split off the main function and the timer that makes the entire Application tick into a Main class, and controlled the interplay between Controller and View from there.

The View is still reliant on a Controller, that is to say, it has a Controller as a field. It uses this to call on appropriate functions in the Controller when buttons in the GUI are pressed. Optimally, this relationship would be decoupled, so that the view does not need to know about a specific Controller implementation. Instead, this could be done with the Observer pattern: The View would offer observability and "announce" events to any Observer. The CarController would register as an Observer and react with appropriate method calls to observed events.

LINK TO UML-DIAGRAM



3)

Observer: Planned to be implemented in the design. Serves to decouple Controller and View.

Factory Method: Planned with consideration of task 5. Will be useful for creating random cars (so that we don’t have to enter random arguments every time we create one).

State: Not planned, but could introduce a State interface for vehicles with Ramps. We could introduce a Driving and a Standing State, which would both have lowerRamp() methods, but the ramp would only be lowered in the Standing state. In the same vein, we could introduce RampUp and RampDown states that have accelerating methods, where only the one in RampUp leads to acceleration. However, for such basic functionality this is a lot of overhead. It also introduces a weird relationship between the Ramp states and the Moving states, where they should have similar behavior or you get weird behavior, like, it might be possible to accelerate when the ramp is down, but it might not be possible to lower or lift the ramp while moving.

Composite: Using it in TransportVehicles (Ramp is its own Class). Main, the application, is basically a composite, holding both a Controller and a CarView, so it can call on both. It also has a Timer object that is used to keep the application updating, and a MainObserver for notifying main of certain events. CarView has a DrawPanel, the part of the GUI that represents the workshops and moving cars. It passes on added cars, workshops, and update calls.
Purpose: Modularity. Different DrawPanels could be added to CarView and they would work the same from CarView's pov. A TransportVehicle could use different Ramps, and at the same time other Classes can make use of the Ramp Class, rather than having to implement their own Ramp.