Uppgift 2:
Vilka avvikelser från MVC-idealet kan ni identifiera i det ursprungliga användargränssnittet? Vad borde ha gjorts smartare, dummare eller tunnare?

- Viewn från den gamla koden borde ha gjorts dummare eftersom att den hanterar användar input som egentligen VehicleController ska hantera.

- VehicleController borde ha gjorts tunnare då den i nuläget hanterar olika beräkningar vilket den ej bör göra.

- Det borde ha funnits en “smart” modelklass som hanterar systemets logik.

- DrawPanel borde ha gjorts dummare eftersom den hanterar logik vilket den inte bör göra eftersom det är modellens jobb.

- Timer ligger fortfarande i VehicleController, men bör ligga i modellen.

Vilka av dessa brister åtgärdade ni med er nya design från del 2A? Hur då? Vilka brister åtgärdade ni inte?

Åtgärdade:
Det finns nu en “smart” modelklass “VehicleModel” som hanterar logik. All logik som låg i andra klasser har flyttats hit.

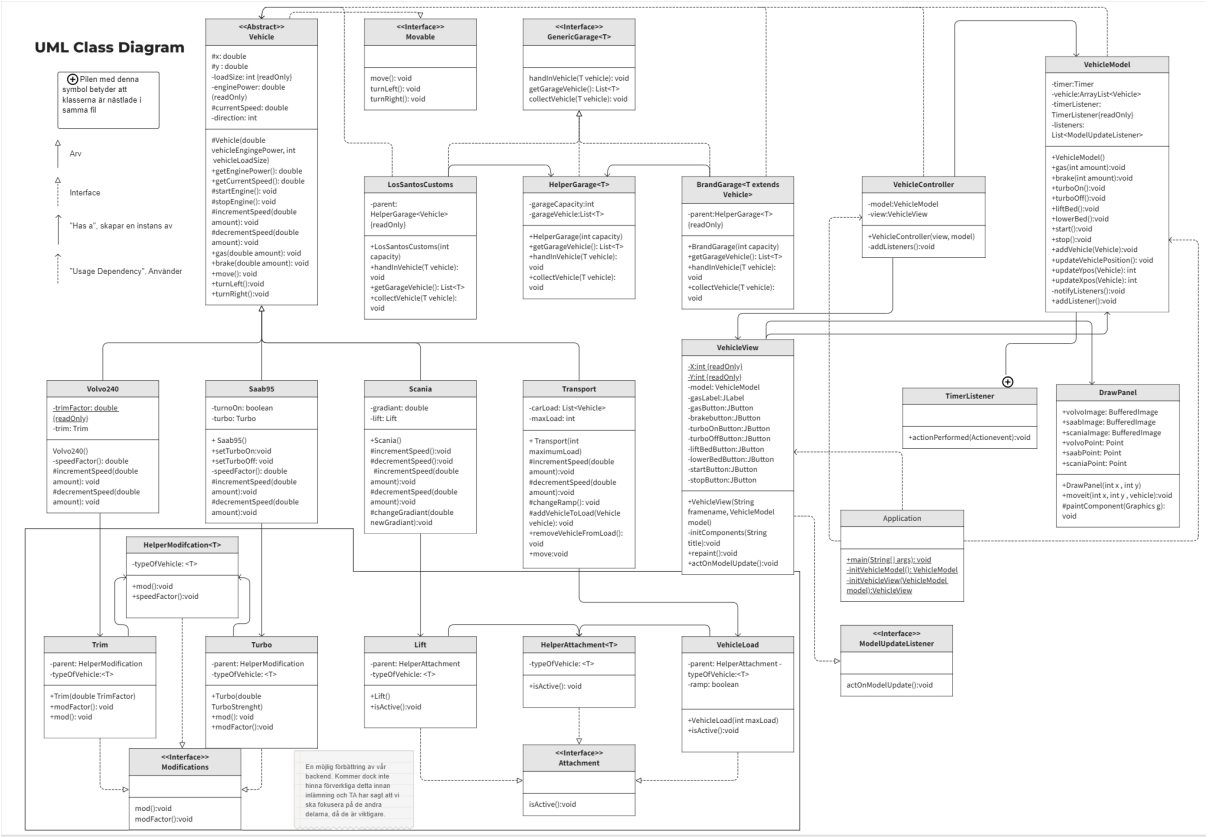
Viewn har blivit dummare då inputs har flyttats till VehicleController.

VehicleController har blivit tunnare då beräkningarna som fanns där nu ligger i modellen.

Timer ligger nu i VehicleModel med hjälp av observer pattern.

Inte åtgärdade:
DrawPanel innehåller fortfarande logiska beräkningar.

UML diagram:



Uppgift 3:

Finns det något ställe i er design där ni redan använder detta pattern, avsiktligt eller oavsiktligt? Vilka designproblem löste ni genom att använda det?

Observer:
Vi har tidigare använt oss av ActionListenners för att känna av knapptryckningar samt att meddela View:n att den ska uppdateras när en action sker. Dessa lyssnare liknar vad en Observer:s funktionellt är. Nu har koden refaktorisera för att använda observer pattern med hjälp av ett interface “ModelUpdateListener” som implementeras av vehicleView. Med hjälp av detta kan klassen TimerListener ligga i vehicleModel där alla lyssnare “vehicleView” notifieras av ändringar som sker i model.

Löser problemet att View:n blev för smart.

Factory Method:
Finns ej!

State:
Följande är inte state pattern, men det är en form av att använda olika tillstånd för att få saker att ske i programmet:
Om flaket är i uppfällt läge på de två lastbilarna (Scania och Transport) så kommer de inte kunna köra och därav har funktionaliteten förändrats på grund av deras tillstånd.

I DrawPanel så uppdaterar vi positionen av alla fordon i funktionen moveIt. Då används fordonens tillstånd i en Arraylist<Vehicle> för att bestämma vilken Point som är tillhörande fordonet i en annan Arraylist<Point>. Sedan kan uträkningen utföras som innan. Det finns möjliga problem med detta, men det var den bästa lösningen vi kom på.

Composite:
Finns ej!

Finns det något ställe där ni kan förbättra er design genom att använda detta design pattern? Vilka designproblem skulle ni lösa genom att använda det? Om inte, varför skulle er design inte förbättras av att använda det?

Observer:
Finns redan!

Factory Method:
Factory method kan vara bra men i detta fallet är factory method onödigt. Det kan vara bra att implementera en factory method för framtida implementationer av Vehicles som flygplan eller båtar men i nuläget finns det bara bilar och om det skulle implementeras flygplan skulle våran design fortfarande fungera eftersom superklassen Vehicle är så pass generell.

Factory Method skulle göra koden bättre för framtiden men i nuläget skulle Factory Method överkomplicera våran kod.

State:
State pattern skulle överkomplicera våran design. I nuläget har Vehicle ett enkelt sätt för att kolla om bilen ska kunna röra sig eller inte. Implementationen av ett state pattern skulle göra programmet onödigt komplicerat bara för att lägga till ett designmönster. Om state mönstret skulle implementeras skulle ett interface skapas där de olika funktionerna för olika states finns. Sedan skulle en klass för varje state skapas där respektive funktion implementeras och sedan skulle dessa klasser användas i koden där beräkningarna sker i nuläget.

Composite:
Composite kan användas för att flytta arrayListen som innehåller alla vehicles från modellen till en composite klass som hanterar en samling av Vehicles.

Uppgift 5:
Kan något designmönster vara relevant att använda för denna utökning?
Vi använder oss redan av en observer för att få signalerna från knapparna i view:n att kunna aktivera logikberäkningarna i modellen så utökningen som krävdes för detta steg var ytterst liten. En förändring var funktionen "removeVehicle" vilket vi inte hade tidigare och är ganska självförklarande vad den gör, den tar bort fordonsobjektet och fordons-pointen ur sina listor.

Det hade troligen gått att skriva om med hjälp av State Pattern hur vi lägger till och tar bort fordonen i de olika listorna vilket hade eventuellt kunnat bli bättre ur ett designperspektiv, men vi ser inte vilket problem det hade löst då det bara hade lett till mer beroenden än vad vi redan har.