Uppgift 2:

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

En avvikelse som vi kan identifiera är att vår controller, 'CarController', var för smart och borde gjorts mycket tunnare. 'CarController' har för mycket logikkod med exempelvis metoder för vad som ska hända efter varje knapptryck, samt metoder som 'checkBoundaries' och 'checkWorkshop' som kollar ifall en bil krockar med kanterna eller workshop. Dessa metoder borde istället tillhöra modellen som då inte var tillräckligt smart. När dessa metoder inte tillhörde modellen kunde den alltså inte fungera helt av sig själv vilket bryter mot MVC-idealet. Även vår view, 'CarView', var för smart då den innehöll 'actionListener'-metoder som anropades efter knapptryck.

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

Med MVC-idealet i åtanke gjordes en förbättring gällande CarView. Knapplogiken flyttades fullständigt till CarController vilket innebar att CarView blev dummare, vilket man strävar efter. Däremot innebar det att CarController blev smartare vilket går emot konceptet. Därför måste vi i nästa steg flytta logikkod från CarController så att den blir tunnare.

Uppgift 3:

- **Observer, Factory Method, State, Composite. För vart och ett av dessa fyra designmönster, svara på följande frågor:**
    - *Finns det något ställe i er design där ni redan använder detta pattern, avsiktligt eller oavsiktligt? Vilka designproblem löste ni genom att använda det?*
    - *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 Pattern Observer Pattern har använts mellan Model och DrawPanel. Ett interface Observer skapades med en metod update() som DrawPanel implementerar. Metoden är till för att skapa en vehicle visuellt när den läggs till i modellen. Model är obeservable medan DrawPanel är en observer.

Composite pattern har vi delvis använt i TimerListener klassen i Model, där olika fordonstyper kan komma åt move(). Detta beror på trädstrukturer där Vehicle är superklassen och subklasserna kan komma åt superklassens funktioner. Även metoden checkBoundaries utnyttjar superklassens funktioner (dvs Vehicles funktioner).

För att följa **Factory Pattern** är tanken att implementera en VehicleGenerator som kan användas. Då kan vi skapa fordon utan att exponera den interna implementationen.

State Pattern har inte implementerats men att det möjligtvis hade kunnat göras på Truck-klasserna Scania och CarTransport. Dessa har en flatBed som kan höjas och sänkas och därmed kan ge objekt av klasserna olika states. I programmet finns ett TruckInterface som innehåller en metod för att kolla vilket state en flatBed har.