### **Uppgift 2: Model-View-Controller**

Användargränssnittet ni utgick från i del A var en ansats till implementation av Model-View-Controller Pattern, men där gränsdragningen mellan model, view, controller och applikation inte var något vidare genomtänkt (for att inte saga usel).

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

Hela applikationen startades från controllern, det var även där som listan med bilar fanns (bör finnas i model). Det fanns även en update-timer som höll i modell-logik som inte bör finnas i en controller. Controllern ansvarade även för att rita om vyn vilket den inte bör göra. I viewn lades event listeners till, vilket rimligtvis är controllerns uppdrag.

Modellen skulle ha behövt bli smartare, viewn dummare och controllern tunnare.

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

Vi åtgärdade inte så många av dessa brister, mycket av modell-logiken finns kvar i controllern. En liten förbättring vi gjorde var att flytta ut main-metoden till en egen klass (applikation) och även lägga till bilarna därifrån.

* **Rita ett nytt UML-diagram som beskriver en förbättrad design med avseende pa MVC.**

### **Uppgift 3: Fler designmonster**

* Observer, Factory Method, State, Composite. För vart och ett av dessa fyra designmönster, svara pa följande frågor:
  + Finns det något ställe i er design dar ni redan använder detta pattern, avsiktligt eller oavsiktligt? Vilka designproblem löste ni genom att använda det?
  + Finns det något ställe dar 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**

I vår nya design för MVC har vi lagt till ett observer pattern för att kommunicera ändringar i modellen till vår applikation, som i sin tur kan uppdatera view:n för att reflektera dessa. Detta gör att vi slipper bero på View i vår Model, vilket skulle bryta mot MVC.

**Factory Method**

Vi har inte implementerat någon Factory / Factory Method men kommer att göra det. Genom att använda en Factory för att skapa bilar istället för direkta konstruktoranrop slipper vi ett beroende på de konkreta subklasserna av fordon.

**State**

~~Vi kommer implementera State genom att ändra engine-klassen till ett interface och lägga till flera olika klasser som representerar olika tillstånd hos motorn, till exempel på och av, som implementerar interfacet. Detta förbättrar designen eftersom det gör det svårare att hamna i ett ogiltigt tillstånd.~~

Vi kom fram till att det inte var en förbättring att använda detta designmönster här. Det skapade mer komplex logik och vi fick mer beroenden som inte fanns tidigare. Ett “TurboOn” state skulle kräva mer logik för att kolla så att vi inte hamnar i ett ogiltigt tillstånd (inga andra bilar än Turboable ska kunna ha en motor med TurboOn state).

(Man skulle hypotetiskt kunna kalla Ramp/LoadingPlatform för “states” av Hingeable, de representerar specifika tillstånd och ansvarar själv för att hålla tillståndet intakt.)

**Composite**

I koden använder vi ett dåligt Composite pattern för att behandla flera bilar som en entitet genom en Ramp. Vi ska göra fixa till detta yttligare. När rampen är fylld med entities agerar rampen, tillsammans med alla entities på rampen, som ett enda objekt. Alla objekt delar position och agerar lika utifrån de metoder som körs, fram tills att de separeras.

~~TODO!! ta bort teleportEntities i ramp och flytta logik till setPosition och setDirection~~

Uppdatera er design med de förbättringar ni identifierat.