- **Vilka avvikelser från MVC-idealet kan ni identifiera i det <u>ursprungliga</u> användargränssnittet?**

Från det ursprungliga användargränssnittet var CarView och CarController cirkulärt beroende av varandra (man använde sig av komposition på varandras objekt) och inom CarController hade man en main-class som startade programmet/applikationen. Där i hade man också en TimerListener-klass som skötte knappar och dyligt. Detta strider mot MVC-idealet eftersom vi vill ha (Car-) Controller som *bara* tar upp användarinput, om det finns då. Men från början hade man logik inblandat i denna klass också, samt själva main metoden för programmet. TimerListener inuti CarController sköter logik för hur knapparna ska visas med hjälp av drawPanel, så här har vi blandat MVC i samma klass.
Move-it funktionen var inne i DrawPanel. Den har inte med vyn att göra. Borde lägga den till modellen.

- **Vad borde ha gjorts smartare, dummare eller tunnare?**

Vi borde ha en **A**pplikationsklass och flytta klassen TimeListener bort från Car Controller, CarView hade controller och vy metoder inom sig, så dessa behöver vi också flytta. Vi vill också gärna flytta alla Actions Listeners och knapparna vi lägger till, så att de inte hamnar i samma klass. Då blir det extra tydligt vart och hur vi använder knapparna samt initialiserar/skapar dem.

- **Vilka av dessa brister åtgärdade ni med er nya design från del 3? Hur då? Vilka brister åtgärdade ni inte?**

Vi åtgärdade:
- En klass A (applikation)
- Göra CarView, DrawPanel och CarController icke-beroende av varandra.
- Flytta all logik från CC, så att den blev mer “tunn”, och inget annat.

  Vi gjorde 2 factory klasser där vi kunde skapa objekt för “Graphics" och “Cars”. Vi har här möjlighet att kunna skapa drawPanel, CarView, CarController och alla bilar som nya objekt. På detta vis blir klasserna mindre (“svagare”) beroende av varandra, istället för direkt komposition. Vi använder dessa factory-metoder för att kunna skapa objekt till main-programmet i Start.

Vi åtgärdade inte:
- Move-it-metoden i drawPanel. Vi märkte efteråt när vi blev klara att move-it inte borde varit i drawPanel utan istället borde vara i modellen eftersom den sköter rörelse-logik för bilbilderna.

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

    - 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 , avsiktligt eller oavsiktligt? Vilka designproblem löste ni genom att använda det?

- Vi har inte använt oss av varken Observer Pattern, State eller Composite Pattern i vår nuvarande kod. Men vi har delvis använt oss av Factory method (avsiktligt), på det viset att vi skapar en car factory som man kan använda för att skapa nya objekt av Volvo, Saab, Biltransport och Scania. Detta använder vi bland annat i main-applikationen/metoden. Vi skapade också en Graphis Factory för att kunna göra nya objekt av DrawPanel, CarController och CarView. Dessa använder vi också i main-applikationen/Start-klassen. Vi gör detta för att beroendet av varandra blir svagare. CarController och CarView är inte längre cirkulärt beroende av varandra från det ursprungliga programmet som exempel.

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?

- Vi skulle kunna använda **observer pattern** för att komma åt positionen för en bil som rör sig. Detta behöver vi bland annat när vi lägger till bilen i en verkstad, eftersom vi då vill berätta för programmet att bilen just då står still i verkstaden och kan inte kommas åt av applikationen, det vill säga knapparna vi trycker på ska inte göra någon effekt. Om bilen befinner sig i en verkstad så vill vi inte göra något, men så fort vi lämnar så ska man återuppta funktionaliteten för bilen (gasa, bromsa, höja flak…). (Vi skulle också kunna använda oss av ett State Pattern här, men vi får se hur vi gör).

- Vi kan förbättra våran ramp som används av Scania och BilTransport. Genom att använda ett **State Pattern**. Det gör att det blir lättare att lägga till fler saker, undvika många if- och else -satser om vi implementerar mer funktioner till rampen. Det blir lättare att återanvända också då.

    - Uppdatera er design med de förbättringar ni identifierat.