Refaktoriseringsupplägg:

1. Applikation för att initiera MVC och Cirkulär dependecy I CarController och CarView

För att minska coupling så väljer vi att flytta ut koden som kör programmet från CarController till en separat "Applikation". Detta ökar även cohesion eftersom CarController nu är mer relevant till sitt uppdrag därav uppfyller vi även Separation of concern.

2. Kör TimerUpdate I applikationen

För att ytterligare minska coupling väljer vi att flytta TimerUpdate klassen till Applikationen eftersom att det skulle kunna kallas Game-loop och berör därför applikationen. Detta uppfyller separation of concern, Single responsibility principle.

LAB 4

- **Vilka avvikelser från MVC-idealet kan ni identifiera i det ursprungliga användargränssnittet? Vad borde ha gjorts smartare, dummare eller tunnare?**
  CarControllern var för smart, då den styrde hela programmet. Dessutom fanns en tendens till cirkulära beroenden, vilket orsakade hög coupling. Dessutom saknades en applikationsklass som har uppgiften att sy ihop programmet, och därmed såg vi en del olämpliga beroenden mellan delarna i MVC. En annan brist är att DrawPanel klassen hör till view, men i strid mot MVC idealen, innehåller mycket logik.

- **Vilka av dessa brister åtgärdade ni med er nya design från del 2A? Hur da? Vilka brister åtgärdade ni inte?**
  Vi åtgärdade alla brister vi hade identifierat vid förra labbtillfället (se refaktoriseringsplan ovan). Vi insåg även att om vi hade refaktoriserat enligt observer pattern skulle programstrukturen förbättras ytterligare då vi får ännu mindre coupling mellan klasserna. Genom att även flytta ansvaret att bevara en lista av alla skapade bilar från carcontroller till själva car modellen statiskt, minskar vi coupling ännu mer då klassar som berodde på carcontroller för att få en lista av cars, t.ex DrawPanel, inte längre behöver carcontroller utan använder klassen car som den redan beror på.

- **Rita ett nytt UML-diagram som beskriver en förbättrad design med avseende pa 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 pattern, avsiktligt eller oavsiktligt?**
    Observer: Nej
    Factory Method: Nej
    State: Nej
    Composite: Nej

- **Finns det något ställe dar ni kan förbättra er design genom att använda detta design pattern?**
  Observer pattern: Ja.
  Factory pattern: Ja

- **Vilka designproblem skulle ni lösa genom att använda det? Om inte, varför skulle er design inte förbättras av att använda det?**
  Enligt ovan insåg vi att genom att införa en observer tar vi bort onödiga arbiträra beroenden och istället abstraherar ett interface för alla klasser som vill lyssna på uppdateringar i modellen. Detta följer open closed principle då det är mycket lättare för eventuella nya klasser i framtiden att lyssna på uppdateringar av modellen än det var innan. Dessutom så kommer inte drawpanel nu uppdateras när bilen står still då ingen notifier skickas, vilket är bättre för prestandan på programmet.

  Vi insåg även att Factory Pattern skulle kunna ta över ansvaret för att skapa bilar av typen car istället för klasserna själva. I framtiden kanske vi vil ändra skapandet av till exempel en Saab där farten alltid är högre. Om vi inte hade en factory pattern skulle vi behöva uppdatera detta på varje ställe en ny Saab skapas i flera olika klasser. Nu behöver vi bara uppdatera koden på ett ställe.