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 (för att inte säga usel).

- Vilka avvikelser från MVC-idealet kan ni identifiera i det ursprungliga användargränssnittet? Vad borde ha gjorts smartare, dummare eller tunnare?
- Vilka av dessa brister åtgärdade ni med er nya design från del 2? Hur då? Vilka brister åtgärdade ni inte?
- Rita ett nytt UML-diagram som beskriver en förbättrad design med avseende på MVC.

Uppgift 3: Lägg till ny vy

- Skriv ut "<Bil>: <Hastighet>" i något lämpligt hörn av panelen, eller kanske på en helt ny JLabel ni kan lägga under alla knappar. Detta bör göras inte som ett tillägg till den befintliga vyn, utan som en ny vy som kan samexistera med den gamla vyn, existera fristående från den gamla vyn, eller samexistera med helt andra vyer.
- Hur bör eran MVC-lösning vara utformad för att möjliggöra att ovanstående förändring blir en utökning snarare än en modifikation?
- Hur bör de olika komponenterna kommunicera med varandra?

Uppgift 4: Fler designmönster

- 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?
- Uppdatera er design med de förbättringar ni identifierat.

Avvikelser från MVC:

- Avsaknad av tydlig och smart model del där det mesta av logiken (tillsammans med Car + subklasser) ska finnas
- Avsaknad av applikation, den är just nu i Controller??
- View är lite väl smart för tillfället…
- Controllern är inte direkt tunn…

Vad åtgärdade vi?

- Vi gjorde Controller tunnare genom att flytta ut skapandet av bilar till CreatedCars.
- Vi gjorde View oberoende av Controller, med hjälp av en observer, vilket leder till en större uppdelning av ansvarsområden.

Vad åtgärdade vi inte?

- Flytta allt som har med uppritning i CarController till CarView. Separera logik och uppritning i TimerListener. Alltså gör CarController oberoende av CarView.
- Ingen Model, sköt på det till nästa uppgift.
- Ingen applikation skapades.

Uppgift 4

Används någon av följande i koden? Vilka designproblem löser dem?

Kan någon av dem implementeras någonstans i koden? Vilka designproblem löser dem? Annars, varför skulle ni inte tjäna något på dem?

Åtgärda!

- Observer

Vi har en observermönster för att eliminera ett beroende mellan carView och carController, samt carModel och carView. Då låter vi carView ta emot event från knapparna i gränssnittet för att sedan skicka en updatesignal till som sedan avgör vilka metoder som ska göras i en switch case i carController m.h.a. ett tillagt enumvärde.

Designproblemet som detta observerobjektet löser är att observern (CC) kan få uppdateringar om observable's (CVs) tillstånd (i detta fallet statusen på eventen) utan att behöva utveckla ett beroende. (High Cohesion, Low Coupling)

- Factory method

Vi har ett Factorymetodmönster som kallas CarFactory. Det den gör är att den skapar objekt av subklasserna till car. I vårt sammanhang har vi valt att använda en Factory method för att kunna skapa en lista av alla bilar i "världen", där samtliga bilar skapas i CarFactory och sedan skickas till CarModel innan listan sedan distribueras till de andra klasserna som behöver den informationen.
Designproblemet som denna fabrikmetoden löser är att vi kan skapa alla tre subklasser av Car utan att någon utav de andra klasserna ska behöva använda konkreta implementationer

av subklasserna till Car. Vi minimerar därmed beroenden av konkreta implementationer. (DIP)

- State

Vi har inte avsiktligt implementerat ett statemönster i vår kod.

State pattern används för att simplifiera processen att uppdatera ett objekts tillstånd. Istället för att skapa en massa booleans och ifsatser som håller koll på tillstånden utspridda i olika klasser så skapar vi istället olika tillstånd som vi sen byter mellan vid signal från programmet.

Finns det en del av koden som skulle tjäna på att använda detta mönstret?

Vi anser att det finns plats för statemönstret att implementeras i vår kod. IRaisables subklasser Mercedes och Scania delar booleans som avgör dels om de har släp eller ej och om de kan köra eller inte. Dessutom delar alla bilar på ett tillståndsfall som avgör om de transporteras eller ej. Problemet med implementeringen då är att den antingen kan gälla samtliga bilar men endast har två states: transporteras eller ej eller att den endast täcker Scania- och Mercedesklassen vilket leder till en inkomplett lösning på transportproblemet (alla bilar inkluderas ändå inte) och en alltför nischad lösning för ramp/move-tillstånden. Med andra ord blir implementationen bara krångliga än vår nuvarande lösning.

- Composite

Vi har inte avsiktligt implementerat ett compositemönster i vår kod. Men använder det i princip genom implementationen av ICar. (se nedan)

Composite pattern används för att alla objekt i arv hierarkin ska kunna funka med t.ex. en metod som om de vore instanser av samma objekt. Dvs. developers, managers och executives är olika klasser i ett företag, men är alla samlade under samma interface så därmed kan alla använda t.ex. en metod som getSalary(). En samling objekt ska kunna manipuleras på samma sätt som ett enskilt objekt gör. (Polymorfism)

Vi använder en abstraktion och en hierarki när vi skapar våra bilar och färjor. Det tydligaste exemplet på detta är våra bilar. Vi använder en abstrakt klass Car som implementerar (bland annat) ICar, Varav vi skapar bilar som utökar abstrakta klassen Car. Ett annat exempel på detta är Ferry. Ferry är en abstrakt klass som implementerar (bland annat) IMovable. Vi använder den abstrakta klassen Ferry för att skapa Stenaline.

Där är Car och Ferry en composite till respektive Bilmärkena (Saab95 osv) och Stenaline. Vi har dock inte några traditionella löv i vår kod eftersom att det ej går att skapa barn till till exempel Stenaline eller Saab95. Så på det viset använder vi oss inte av ett traditionellt Composite-pattern, och vi ser inte (med den strukturimplementation vi använder just nu) att det hade varit passande att använda detta.
Refaktorisering:

1. Sära på/dela upp modellen och applikationen
2.