Uppgift 2 - Beroenden

Analysera de beroenden som finns med avseende på cohesion och coupling, och Dependency Inversion Principle.

Vilka beroenden är nödvändiga?

- Vehicle och dess subklasser
- CarTransport och CarWorkshop “has a” storage. (komposition)
- Scania och CarTransport “has a” truckbed. (komposition)
- CarController är i behov av en CarView som i sin tur är i behov av DrawPanel. Dessa beroenden (komposition) anses nödvändiga.
- Vi har beroenden till externa ramverk, som t.ex JFrame, JPanel - dessa är nödvändiga för att vi grafiskt kan realisera vårt program och använda bilarna.

Vilka klasser är beroende av varandra som inte borde vara det?

- CarController är beroende av CarView, samtidigt som CarView är beroende av CarController (mutual dependency). Mutual dependency är generellt inte önskvärt.

Finns det starkare beroenden än nödvändigt?

- Vi har ganska hög coupling (starka beroenden) till ett flertal externa ramverk.
- Vi har starka beroende internt, mellan t.ex CarView och CarController (mutual dependency) - detta bidrar till hög coupling

Kan ni identifiera några brott mot övriga designprinciper vi pratat om i kursen?

Om vi ser till Separation of Concern:
Detta är något vi försökt jobba med kring våra metoder men det finns en del fall där man möjligtvis hade kunnat använda sig av mer functional decomposition (funktionell nedbrytning). I t.ex CarTransporter - för att göra metode lättare att förstå. Detta är viktigt för att uppnå en bra abstraktion - som leder oss till mindre komplexitet. Det hänger även ihop med OCP för att vi ska kunna återanvända och utöka utan förändring i framtida fall.

OCP hänger ofta ihop med DIP. Om vi uppnår ett av dem så får vi ofta med de andra. DIP - säger att vi ska använda abstraktioner före konkreta implementationer. Vi hade kunna implementera fler interfaces för att uppnå detta och öka reusability. Detta hade kunnat göras med t.ex interface Loadable - om fordonet kan lastas på CarTransporter eller Workshop, eller Truck och Car som interfaces för att särskilja fordon och enkelt kunna lägga till flera. Detta bidrar med ökad abstraktion och gör modulen öppen för extension precis som OCP hänvisar till.
För att verkligen öka reusability och extensibility hade vi även kunnat implementera interface Drawable som “CarGame” tar in -> fordonen som skapas i CarGame ska vara drawable - alltså gå att rita ut i vår view.

Single-Responsibility Principle:

- Vi har en del klasser som har flera olika ansvarsområden, när man egentligen vill sträva efter att varje klass har ett specifikt område. Detta bryter mot SRP och medför till low cohesion i våran modul och förmodligen kan ha gett oss high coupling, då risk för onödiga beroenden tillkommit.

Uppgift 3: Ansvarsområden

Analysera era klasser med avseende pa Separation of Concern (SoC) och Single Responsibility Principle (SRP).

Vilka ansvarsområden har era klasser?

Vehicle har ansvar för:
- Att definera hur objekt av typen Vehicle ska instansieras
- Definerar många variabler och metoder som delas för alla subtyper av Vehicle (Definierar hur ett fordon fungerar / gör).

Movable har ansvar för:
- Interface som definerar hur fordon ska kunna förflytta sig.

Volvo240 har ansvar för:
- Att definera hur objekt av typen Volvo240 ska instansieras
- Hur just Volvo accelererar (overridear increment/decrement - speed)

Saab95 har ansvar för:
- Att definera hur objekt av typen Saab95 ska instansieras
- Hur just Saab accelererar (overridear increment/decrement - speed)

Scania har ansvar för:
- Att definera hur objekt av typen Scania ska instansieras
- Att Scania kan startas om truckbed är nere (overridar startEngine)
- Skapar genom komposition ett objekt av typen Truckbed
- Kallar metoder från Truckbed som påverkar Truckbed-objektet

CarTransporter har ansvar för:
- Att definera hur objekt av typen CarTransporter ska instansieras
- Skapar genom komposition ett objekt av typen Truckbed
- Kallar metoder från Truckbed som påverkar Truckbed-objektet
- Skapar genom komposition ett objekt av typen Storage
- Lasta CarTransportern och förflytta bilarna som lastats

Truckbed har ansvar för:
- Definiera metoder för Truckbed-objekt
- Definierar en variabel truckbedAngle

Storage har ansvar för:
- Definiera metoder för Storage-objekt
- Definierar en variabel maxLoad

- Definierar en arrayList som håller Cars-objekt

CarWorkshop har ansvar för:
- Skapar genom komposition ett objekt av typen Storage
- “Lasta” / “Avlasta” bilar från verkstaden.

CarController har ansvar för:
- Att skapa ett timer-objekt
- Skapar Vehicle - objekt
- Att skapa ett CarController-objekt
- Att lägga till bilar i en lista som tillhör CarController.
- Att skapa ett CarView-objekt.
- Att med hjälp av timerobjektet uppdatera bilarnas positioner.
- Att implementera metoderna (från Vehicle) gas, brake, startaAllCars, stopAllCars, turboOn, turboOff, LiftBed, lowerBed, efter instruktion från CarView.
- Att säkerställa att bilarna inte rör sig utanför fönstret.
- Att kalla metoden paintComponent i DrawPanel genom repaint()

CarView har ansvar för:
- Att skapa det fönster där bilarna ritas (genom att skapa ett DrawPanel-objekt)
- Att skapa en kontrollpanel med knappar för att styra bilarna
- Att skapa actionListeners som lyssnar efter input på knapparna och kallar på CarController för att implementera metoderna gas, break, etc..

DrawPanel har ansvar för: (Animera en del av CarView med bilder på bilarna)
- Att skapa en HashMap (ungefär ett slags dictionary) som kopplar Vehicle-objekt till deras motsvarande bild-filer.
- Att definera storlek och utseende för fönstret där bilarna ritas.
- Att skapa en String för varje bils bildfil utifrån dess modellnamn för att sedan kunna läsa in bilarnas bilder
- Overridar JComponents metod paintComponent, som uppdaterar bilden varje gång den refreshas mha timern (kallas av CarController)
- Läser HashMap-objektet och ritar för varje bil en bild av bilen på dess koordinater.

Vilka anledningar har de att förändras?

Vehicle
Vehicle kan behöva ändras om man skulle vilja lägga till nya attribut eller beteenden för samtliga objekt av typen Vehicle.

Volvo240
Kan behöva ändras om man vill lägga till nya attribut eller beteenden för Volvo240

Saab95
Kan behöva ändras om man vill lägga till nya attribut eller beteenden för Saab95

Scania

Kan behöva ändras om man vill lägga till nya attribut eller beteenden för Scania

CarTransporter
Kan behöva ändras om man vill lägga till nya attribut eller beteenden för CarTransporter.

Truckbed
Kan behöva ändras om man vill lägga till nya attribut eller beteenden för alla objekt av typen Truckbed (Scania och CarTransporter i nuläget, men även om man skulle skapa ett nytt objekt som har ett Truckbed-objekt)

Storage
Kan behöva ändras om man vill lägga till nya attribut eller beteenden för alla objekt av typen Storage (CarTransporter och CarWorkshop i nuläget, men skulle kunna tillkomma nya klasser som kräver en ny implementation)

CarWorkshop
Kan behöva ändras om man vill lägga till nya attribut eller beteenden för CarWorkshop

CarController
Kan behöva ändras om
- vi vill lägga till fler Vehicles som ska ritas i vyn.
- fler möjligheter att styra bilar läggs till i CarView (kalla fler metoder)
- om vi lägger till nya actionListeners som lyssnar efter knapptryckningar, till exempel för att svänga.

CarView
Kan behöva ändras om man önskar lägga till nya sätt att styra Vehicles, till exempel att låta användaren svänga, eller kontroller för att lasta Vehicles på Truckbeds eller i en CarWorkshop.

DrawPanel
Kan behöva ändras om man skulle vilja rita fler slags objekt i fönstret, till exempel rörliga Truckbeds som tillhör olika vehicles (kanske där man ritar att andra objekt lastas på), eller fasta objekt som en CarWorkshop.

På vilka klasser skulle ni behöva tillämpa dekompositionen för att bättre följa SoC och SRP?

CarController och CarView skulle behöva delas upp i flera klasser, då de i nuläget har många olika ansvarsområden. Detta bryter mot både SoC och SRP, där varje klass ska ha ett ansvarsområde.

Uppgift 4: Ny design

- **Rita ett UML-diagram** över en ny design som åtgärdar de brister ni identifierat med avseende både på beroenden och ansvarsfördelning.
- Motivera, i termer av de principer vi gått igenom, varför era förbättringar verkligen är förbättringar.
- **Skriv en refaktoriseringsplan.** Planen bör bestå av en sekvens refaktoriseringssteg som tar er från det nuvarande programmet till ett som implementerar er nya design. Planen behöver inte vara enormt detaljerad.
- Finns det några delar av planen som går att utföra parallellt, av olika utvecklare som arbetar oberoende av varandra? Om inte, finns det något sätt att omformulera planen så att en sådan arbetsdelning är möjlig?

Motiveringar till våra ändringar:

För att undvika direkt beroende mellan CarController och skapandet av fordonen så kan vi med hjälp av Factory Pattern använda oss av en bilfabrik (VehicleFactory), som skapar objekt av fordonen. Genom detta så skapa objekt samtidigt som döljer detaljerna och den interna komplexiteten för klienten. Det hjälper oss alltså att dölja den interna implementationen samtidigt som vi ger ett förenklat gränssnitt. Vi ökar alltså abstraktionen och gör komponenterna lättare att förstå och blir av med onödiga beroenden.

Flytta ActionListeners till CarController. CarView skapar knapparna, men ActionListeners bör ligga tillsammans med metoderna som kallas vid ”Actions” - händelserna. Eftersom det är del av “input” för vad bilarna gör vid knapptrycken.

Main funktionen bör tas bort från CarController - detta hör till “model” - alltså vår controller bör inte skapa bilar. Den bör istället ligga inom “Model - package” alltså vara skild från vår controller och vår view. Det är sista steget i att CarController endast skall ha ett ansvarsområde (SRP, SoP). Tanken är att “Main” eller själva skapandet av bilar till spelet flyttas till en klass “Game” som med hjälp av VehicleFactory skapar bilarna lägger till de i en lista med “Spelbara objekt” - (om inte lista funkar, kanske interface drawable?)

För att göra programmet mer extensible och lättare att utveckla så kan man göra en ny abstrakt klass som heter Truck (subklass till Vehicle). Detta i sin tur gör att man i framtiden kan enkelt lägga till nya Trucks till koden. På så sätt kan vi särskilja Trucks och Cars från Vehicles ifall framtida förändringar endast har med den ena typen av fordon att göra. Inte lika starka beroenden från just Vehicle

Refaktoriseringsplan

1. Indela klasser i korrekta packages (främst skilja CarController - Model - CarView/DrawPanel

   (package.View
   - CarView
   - DrawPanel

   package.Controller
   - CarController

   package.Model
   - CarGame
   - Vehicles
   - etc.)

2. Skapa en (abstrakt) klass VehicleFactory som har som ansvar att skapa instanser av olika Vehicles.
3. Refaktorisera main i CarController till en ny klass Game.
   a. Ta bort direkta anrop till att skapa Vehicles.
   b. Skapa Vehicles genom anrop av VehicleFactory i konstruktorn.
   c. kan lägga till objekt (Vehicles) till spelet.
   d. Instansiera CarController.
4. I CarController skapa konstruktor.
5. Flytta actionListeners från CarView till konstruktorn i CarController. (Nu bör vi ha fått bort mutual dependency CarView - CarController sinsemellan.
6. Skapa en abstrakt klass som heter Truck.
7. Möjliga Interfaces…

Det skulle vara möjligt att utföra skapandet av Truck, VehicleFactory och Game något parallellt med sammanfogandet av CarView och CarController. Dock kommer sedan CarController och Game behöva ha ett beroende mellan, men de första stegen, alltså själva skapandet och ändringarna går att göra separat.

Möjliga kodbaser att analysera:

Html parser i java:

https://github.com/HK-SHAO/DarkCalculator

Samling med små spel:
https://github.com/brianalcl/1609

https://github.com/jagrosh/MusicBot

https://docs.oracle.com/javase/8/docs/api/

10 klasser
alklepin/Robots: The project to learn OO design concepts and MDI application development in Java (github.com)

Vi valde denna:

14 klasser - Courier management system
James Alberta/Courier Management System: JAVA - Courier Management System (github.com)

19 klasser - Shopping Cart - denna känns bra
Iman/shopping-cart-java: This project was created as part of recruiting process in Java. (github.com)

12 klasser - Goose Game - relativt simpel kod.
goose-game/src/goosegame/GooseCell.java at master · shezadt/goose-game (github.com)