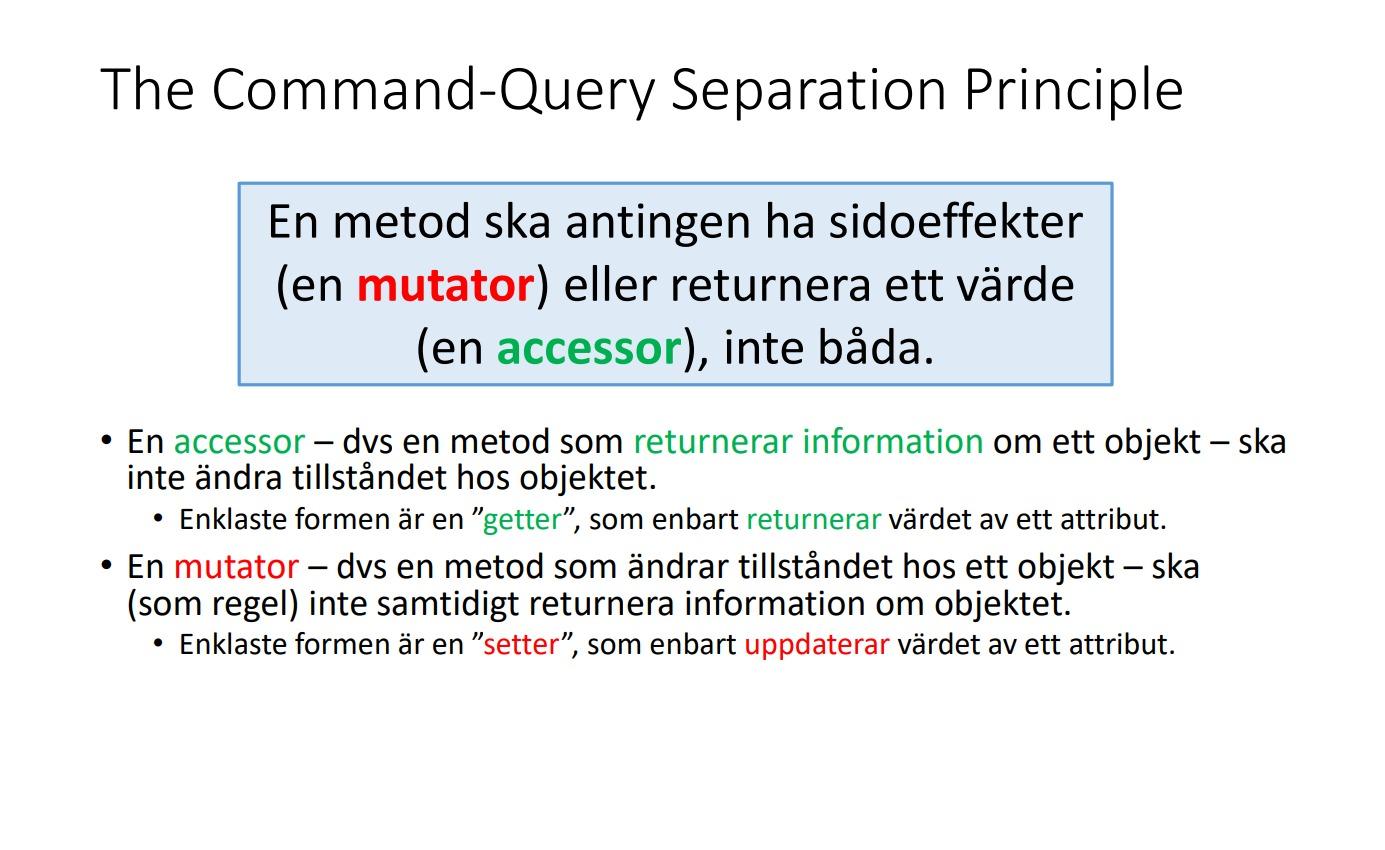
## 

## 



## **Beroenden**

* **Vilka beroenden är nödvändiga?**

Den grundläggande byggstenen i vår datamodell är klassen “Car” tillsammans med subklasserna till “Car”, t.ex. “Truck”, “Saab95” och “Volvo240”. Alla subklasserna har ett arvsheriarktiskt beroende till dess superklass och dessa beroenden är nödvändiga.

Både Scania och Lorry klasserna har båda ett flak; de använder en respektive en LoadingPlatform och Ramp. Dessa beroenden är nödvändiga.

DrawPanel måste veta vilka objekt den ska rita ut och därmed måste den ha en lista med RenderObject.

Att Mechanic beror på Car är inget problem då det är rimligt då det centrala syftet med en Mechanic är att arbeta med bilar/Car.

Det är nödvändigt att DrawPanel beror på BufferedImage för att kunna mellanlagra inladdade bilder för att slippa onödig minnesanvändning.

I många fall har vi ökade beroenden just för att vi skapat utbruten funktionalitet i nya klasser, som vi nu måste bero på.

Det är nödvändigt att CarView beror på (har en) CarController då det är UI:t som håller i knapparna för user input och den datan förs vidare till controllern.

Värt att notera är ObjectStorage vilket inte har någora beroenden (förutom bl.a. List) inga interna eller externa beroenden. Detta gör den mycket återanvändbar!

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

Mycket av koden som ligger i CarController/CarView har beroenden som inte är nödvändiga. För att exempelvis kunna lyfta flaket behöver man inte konkret veta att det är en Scania. Den behöver heller inte veta att det är konkret Saab för att sätta på turbon. Det vore bra med ett “Turboable” interface.

Att Ramp beror på Car är inte optimalt. Detta då det förhindrar en att använda Rampen till annat än bara object av typen Car.

CarController och CarView har ett dubbelt beroende på varandra vilket inte behövs. CarController ska inte direkt behöva känna till CarView.

Lorry har ett beroende på att den måste veta att Car existerar. Detta är inte bra då det igenligen är Rampen som förvarar objekt. Rampen i sin tur skall inte heller behöva veta att Car finns då syftet för den är att förvara objekt.

Car borde inte bero på Direction. Den borde endast ha ett beroende på Vector, som i sin tur modifieras för att implicit uttrycka bilens Direction.

* **Finns det starkare beroenden än nödvändigt?**

De starkaste beroendena vi har är subklassningen mot Car och Lorry vilket är ett designval vi gjort. I detta fallet är det mycket tydligt att Volvo/Saab ÄR en Car men visst skulle viss flexibilitet kunna utvinnas från att använda komposition istället.

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

Mycket av koden som tilldelades till laboration 3 och som vi snabbt utökat utan vidare eftertanke bryter mot flera designprinciper. Den följer inte OCP överhuvudtaget då konkreta subklasser används överallt istället för mer abstraherade typer, vilket t.ex. gör det otroligt svårt att lägga till fler lastbilar med ramper/flak som ska lyftas utan att behöva ändra i källkoden.

Vi bryter även mot Law of Demeter i CarController då vi anropar frame.drawPanel vilket inte är den närmsta grannen.

I många fall skulle även namngivningen kunnat bli bättre då det finns en del tvetydiga namn på vissa attribut, metoder osv.

## **Ansvarsområden**

Analysera era klasser med avseende pa Separation of Concern (SoC) och Single

Responsibility Principle (SRP).

* **Vilka ansvarsområden har era klasser?**

Klassen “Car” innehåller funktionalitet som är allmän för alla bilmodeller. “Truck”, “Saab95”, “Volvo240”, “Scania” och “Lorry” innehåller attribut och metoder som bara används av specifika modeller. Den representerar vad det är att vara en bil samt ansvarar för att förflytta sig och liknande.

“ObjectStorage” innehåller allmän funktionalitet för att lagra objekt, medan “Ramp” och “Mechanic” har specifika sätt att använda de funktionerna och ytterligare funktionalitet för deras respektive användningsområden.

“CarController”, “CarView” och “DrawPanel” används för att skapa det grafiska gränssnittet. “CarController” används för att styra bilarna och innehåller “main”-metoden som programmet körs genom. Den innehåller även klassen “TimerListener” som används för att regelbundet uppdatera gränssnittet. “CarView” skapar gränssnittets layout och skapar knapparna. “DrawPanel” sköter grafiken.

Vi har även enum-klasserna “WeightClass”, som används av bilmodeller för att bestämma vikt och av “Ramp” för begränsa hur tunga bilar den kan förvara, och “Direction”, som används för att spara och hantera en bils riktning. Slutligen finns en “Vector”-klass som framför allt används för hantering av bilars position.

Lorry ansvarar för att för att lägga till bilar på rampen och samtidigt beräkna om de är nära nog för att få åka på. Det är rampen som själv borde säga till om bilarna är för långt borta; rampen ska vara en egen entity (med en egen position) som ska kunna göra avståndsbedömningen oberoende av lorry:n.

* **Vilka anledningar har de att förändras?**

Om Car skulle implementera mer funktionalitet som ska kunna styras av användaren måste fler knappar läggas till på kontrollpanelen. För att göra detta behöver utvecklaren gå in och modifiera den nuvarande koden (vilket inte följer OCP).

Car skulle behöva förändras om man behöver uppdatera det tillåtna intervallet för gasning/bromsning, kanske för ett specifikt fordon.

Direction skulle behöva förändras avsevärt om man vill lägga till nya riktningar som bilen kan röra sig åt, mycket av klient koden skulle också bli ogiltig då.

Om lorry:n blir en egen entity med egen position begrännsas den inte i funktionalitet beroende på om den tillhör en lorry eller inte, dvs. rampen kan själv lägga till bilar på sig själv utan hjälp av en lorry.

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

“CarController” borde förändras eftersom det inte är nödvändigt att ha metoder för att kontrollera bilarna i samma klass som programmet startas och uppdateras från. Att göra det på det sättet innebär att klassen måste innehålla viss funktionallitet för att hantera grafik, vilket inte är relaterat till dess huvudsakliga ansvarsområde och därför bryter mot SRP. Det skulle kunna lösas med dekomposition genom att man skapar en ny klass t.ex. “Main” som innehåller “main”-metoden och “TimerListener”-klassen. “Main”-klassen skulle då använda instanser av “CarController” och “CarView” genom composition.

“Lorry” har delvist ansvar för att lägga till och ta bort bilar; detta ansvaret delas med Ramp. Optimalt hade varit om allt ansvar för att last på och av bilar hade lagts på Ramp, detta då rampens huvudsyfte är att hantera objekt (eller bilar som den nuvarande versionen endast kan hantera).

Koden i CarView är väldigt svårläslig då det nästan inte finns någon dekomposition alls. Exempelvis skulle “initComponents” kunna brytas ned avsevärt i de olika grupperna av komponenter som används.

“Car” gör alldeles för många saker samtidigt, den representerar att vara en bil, men även att förflytta sig och gasa/bromsa. Detta skulle kunna brytas upp i ännu mindre beståndsdelar såsom Engine och liknande.

“CarController” skulle behöva mycket mer funktionell dekomposition i t.ex. “TimerListener” som är väldigt komplex att förstå.

* **Övriga saker som måste fixas**

Flera av klasserna beror på klasser som är mer specifika implementationer än vad de egentligen behöver vara. Man kan alltså abstrahera vissa av de statiska typerna ytterligare, t.ex. i Ramp och Lorry där båda använder sig av Car objekt men egentligen endast behöver en position.

## **Refaktoriseringsplan**

* Ändra “Car”-klassens “direction”-attribut så att det är en instans av “Vector” istället för “Direction”. *Detta följer OCP då vi lättare kan utöka med flera riktningar i framtiden, vi behöver inte ett starkt beroende på fyra hårdkodade riktningar som är svåra att manipulera.*
* Ändra Ramp så att den tar objekt av typen Entity istället för endast bilar.
* Bryta ned Car i mer återanvända beståndsdelar
  + Skapa en Engine klass
* För att följa SoC delar vi upp klassen CarView. Vi gör även funktionell nedbrytning på metoderna i klassen.
  + Dela upp i klasser: ControlPanel, GasPanel, GasSpinner
    - En metod som skapar fönstret.
* Skicka in X och Y till CarView för att specificera storlek istället för hårdkodade dimensioner
* Använda funktionell nedbrytning på update/TimerListener-koden i Main
* Skapa interface “Platformable” och “Turboable” som kan användas i CarController istället för konkreta subklasser - *Detta följer både OCP då vi lättare kan utöka med flera fordon med LoadingPlatform samt följer vi DIP bättre eftersom vi inte behöver high coupling till konkreta subklasser.*
* Skapa en “Main” klass där programmet startas, anropa addCar på CarController utanför CarController
* Skicka in en array med supportedModels till CarController istället för att ha en statisk array - *Följer OCP då det blir mycket lättare att utöka med fler modeller utan att behöva ändra ett statiskt attribut i en intern klass.*
* Gör alla package-private attribut i CarView till privata - *Detta följer encapsulation då dessa interna detaljer inte behöver exponeras utåt i paketet.*
* Skapa ett Entity interface som representerar en entitet i världen, skapa en “getPosition” metod så att vi t.ex. kan jämföra positionen med en lorrys position för att se till att den är tillräckligt nära.
* Då lorry ej *endast* ska kunna hantera bilar, utan ska kunna hantera allt som dess tillhörande ramp kan hantera skall metodnamnen döpas om samt justeras så att alla stödja typer (t.ex. en Entity).
* Skapa loadImage och getHeight/getWidth i CarView som anropar sin **privata** frame så att vi slipper göra frame.drawPanel.loadImage vilket bryter mot law of demeter.
* Dela upp koden mer i paket, *följer SoC då det blir tydligare vad de olika klasserna tillhör för kategori.*
* Uppdatera konkreta typer som ArrayList i CarController till mer breda typer som List som endast möjliggör det gränssnitt vi vill använda. (Dependency Inversion Principle - vi beror på en abstraktion, inte konkreta implementationer)
* Byt namn på Car till Vehicle för att tydliggöra dess bredare ansvarsområde, en Truck är inte en “Car” utan mer en “Vehicle”.
  + **NOTERING: Innan denna punkt har den nu omdöpta VehicleController, Vehicle o.s.v. refererats till som CarController, Car o.s.v.**
* Ändra Lorry så att den inte behöver ansvara för logiken att lägga till bilar och beräkna avstånd, låt Ramp göra det istället. Följer bättre SRP och vi får högre cohesion eftersom det tekniskt sett inte är lorryns ansvarsområde.

# Kan refaktoriseringen utföras parallellt av flera utvecklare?

*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?*

För att kunna arbeta samtidigt behöver refactoriseringsplanen göras stegvis i en specifik ordning. Då många klasser och metoder ska döpas om måste de fixas först. Det är även fördelaktigt att “jobba uppifrån och ner”, att börja arbeta med de största paketen/modulerna och sen börja jobba längre och längre ner. Desto längre ner i koden man kommer, desto enklare är det att arbeta parallellt.

* Man kan arbeta med CarController/CarView/DrawPanel relativt isolerat från de utvecklare som arbetar med datamodellen då det inte finns så starka beroenden däremellan.