# Uppgift 2: Beroenden

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

* *Vilka beroenden är nödvändiga?*

Finns subklasser till Vehicle (Car och Trucks). Dessa ÄR sin supertyp men med ytterligare funktionalitet så ett arvs-förhållande är rimligt. Barnen återanvänder koden som finns i superklassen och har konceptuellt och stabilt ”Is-a”-förhållande. De är en utökning av superklassens och har ett publikt beteende för subklassen som är en utökning av superklassens

Parametrisk polymorfism i RepairShop till Car. Detta är nödvändigt för att kunna begränsa shopen till olika Car modeller samt kunna ha shops med alla typer av Cars.

Loadable har också parametrisk polymorfism till Vehicle. Detta för att kunna ladda med enbart subklasser till Vehicle.

CarController måste kunna skapa bilar. Beroendet som föreligger just nu är usage dependency, vilket är mer svårläst än om man skulle använda komposition, men svagare.

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

CarView och CarController beror för tillfället på varandra vilket inte är nödvändigt och leder till hög coupling.

Just nu beror CarController på subklasser (använder volvo, saab och scania) av Vehicle, det hade kanske varit mer rimligt att beroendet går till superklassen Vehicle istället. Genom att använda supertyper istället för subtyper kan vi minska beroendet av en specifik klass (DIP).

CarController bryter mot LOD i och med att klassen inte bara pratar med sin ”närmsta granne”. Detta kan illustreras med exemplet ” () cc.frame.drawPanel.repaint()” då CarController objektet kallar på CarView-objectets attribut drawPanel (:DrawPanel), för att slutligen använda dess metod repaint.

* *Finns det starkare beroenden än nödvändigt?*

Få bort Carview referenerna från CarController. Att minimera dessa referenser skulle minska coupling och göra systemet mer modulärt.

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

HCLC innebär att vi ska sträva efter att externa beroenden är få och svaga och att interna beroenden är få och svaga. Denna princip följs inte vilket påverkar OCP. Likaså med DIP och LOD som också förhindrar OCP i och med att de inte följs på ett bra sätt. Detta påverkar systemets möjlighet till utvidgning och underhåll.

# Uppgift 3: Ansvarsområden

*Analysera era klasser med avseende på Separation of Concern (SoC) och Single Responsibility Principle (SRP). Vilka ansvarsområden har era klasser? Vilka anledningar har de att förändras? På vilka klasser skulle ni behöva tillämpa dekomposition för att bättre följa SoC och SRP?*

**Vehicle**: abstrakt klass som agerar som ett gränssnitt för fordon

Car: abstrakt klass som är en subklass till Vehicle, grunden för en bil. Finns för att kunna separera bilar från andra fordon. Kan addera ytterligare funktionalitet framöver

**Volvo240**: skapar en Volvo240

**Saab95**: skapar en Saab95

**Truck**: abstrakt klass som är en subklass till Vehicle, grunden för en lastbil.

**Scania**: skapar en Scania

**Lorry**: skapar en biltransport

**TruckBed**: abstrakt klass som agerar som ett gränssnitt för lastbilsflak

**AdvancedTruckBed**: skapar ett dynamiskt lastbilsflak

**StandardTruckBed**: skapar ett statiskt lastbilsflak

**NullTruckBed**: Null-lastbilsflak. Används för lastbilar med flak utan funktionalitet – framtida utökning.

**RepairShop**: skapar en bilverkstad

**CarController**: skapar upp objekt och sköter kommunikationen med dem. Innehåller mainfunktion och TimerListener.

**CarView**: skapar och hanterar knapparna

**DrawPanel**: ansvarar för det grafiska utritandet av knapparna och objekten

Att flytta ut TimerListener från CarController skulle förbättra projektets struktur och modularitet. Detta steg gör det tydligare vilken komponent som hanterar tidbaserade uppdateringar av bilarnas positioner, vilket är i linje med SoC-principen.

CarView ansvarar för att skapa det grafiska gränssnittet, inklusive knappar. Att separera skapandet av komponenterna till en annan klass skulle göra att CarView endast fokuserar på presentationen, vilket skulle stärka SRP.

För tillfället är CarController beroende av CarView för att uppdatera den grafiska representationen av bilarna. Modellen bör dock kunna existera oberoende av viewn och controllern, vilket innebär att CarController inte bör känna till CarView. Att eliminera beroendet från CarController till CarView skulle kunna bidra till en mer flexibel och återanvändbar kod.

För att ytterligare följa SoC och SRP bör CarController dekomposeras för att särskilja olika ansvarsområden som tidshantering och interaktioner med modellen från den grafiska representationen.

CarView bör också dekomposeras, exempelvis genom att separera ansvaret gällande det grafiska utritandet och hantering av knappinteraktionerna.

# 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?*

**Refaktoriseringsplan**

1. Funktionellt bryta ner metoder som bryter mot The Command-Query Seperation Principle.  
   Ex. checkCollisionRepairShop & changeDirection. En metod som enbart ändrar direction och en metod som anropar på denna när den slår i väggen.
   1. För att tydliggöra metodernas ansvarsområden och förbättra kodens läsbarhet.
2. Deklarerar bilder (imageStrings) som attribut i Vehicle-klassen och skickar in de som argument i subklassernas konstruktorer samt RepairShop-klassen.  
   Ta bort Point
   1. Eliminerar beroendena från den grafiska modellen.
3. Skapa ny klass CarViewGraphics – hantera all grafisk utritning
   1. Detta för att CarView bara ska hantera en aspekt av systemet, SRP. CarView byter namn till GameEngine eftersom denna har main funktionen.
4. Eliminera kompositionsberoendet från CarController till GameEngine(frame). Detta eliminerar även usage dependency mellan CarController och DrawPanel.
   1. Law of Demeter
5. Flytta ut TimerListener till en egen klass.
   1. SRP. Mer modulär design med tydligare struktur.
6. Använd subtypning för parametrisk polymorfism för att eliminera beroende från CarController till Volvo240. Just nu invariant, så kan ej använda subtypspolymorfism. Använd explicit covarians. Istället för att deklarera RepairShop till att bara ta in Volvo240 (invariant) så använder vi wildcard. Nu kan elementen antingen ha statisk typ Vehicle eller annan statisk typ som är subtyp till Vehicle. Denna kod kan återanvändas i större utsträckning.

Steg 1, 2, 5, 6 samt (3 & 4) kan utföras parallellt av fyra olika utvecklare.