Uppgift 2 (del 2):
Analysera de beroenden som finns med avseende på cohesion och coupling, och Dependency Inversion Principle.

- Vilka beroenden är nödvändiga?
- Vilka klasser är beroende av varandra som inte borde vara det?
- Finns det starkare beroenden än nödvändigt?
- Kan ni identifiera några brott mot övriga designprinciper vi pratat om i kursen?

CarView använder komposition för att komma åt **CarController** och DrawPanel. Åt andra hållet använder **CarController** komposition för att komma åt **CarView** och Workshop. Att CarView och CarController återfinns i bägge kompositionerna innebär att det finns en Mutual Dependency (starkt beroende) vilket inte känns rimligt vid eftertanke. Detta går även emot Open Closed Principle, om vi gör metod ändringar i ena klassen kan det innebära att den andra klassens funktionalitet påverkas och att man måste ändra den klassen också.

- CarView och CarController har hög coupling, vilket innebär att det finns mindre möjligheter att använda klasserna enskilt (går emot OCP). De har även ganska låg cohesion (går emot high cohesion, low coupling).

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?

CarView sköter allt som sker i fönstret och har även ActionListeners för knapparna som kollar om knapparna trycks på i programmet. De specifika implementationerna av knapparna återfinns i CarController. En möjlig “förbättring” enligt SoC skulle vara att skilja på knappar och view, så man betraktar en sak i taget för lättare förståelse (skulle även göra det mer modulärt).

DrawPanel målar fönstret och flyttar fordonen som åker i fönstret (även workshop). Man bör överväga att låta CarController ta ansvaret för att flytta fordonen.

CarController sköter bilarnas/lastbilarnas logik, bland annat hur de specifika implementationerna av knapptryckningarna fungerar. Dessutom kollar den så att objekten befinner sig innanför fönstret (checkar kollisioner med väggar+workshop).

Vehicle är en abstrakt klass som innehåller generella egenskaper för fordon (antal dörrar, modellnamn etc.).

Car är en abstrakt subklass till Vehicle som innehåller ytterligare information som krävs för bilar (längd och vidd).

Truck är en abstrakt subklass till Vehicle som innehåller ytterligare information om ex. flaket på lastbilen.

Vehicle, Car och Truck är en rimlig uppdelning enligt SoC-principen eftersom klasser inte borde representera flera modeller om de inte har något att göra med varandra.

Movable är ett interface som enbart innehåller förflyttningsegenskaper för fordonen. Detta känns rimligt enligt SoC-principen eftersom den är applicerbar på flera fordonstyper.

TruckInterface innehåller information som är specifik för lastbilstyperna men som har olika implementationer.

Saab95 och **Volvo240** är två subklasser till Car. Detta är en rimlig uppdelning eftersom typerna har specifika egenskaper (t.ex. Saab95 har turbo). Om något behöver ändras hos bilarna görs detta i deras specifika klasser, vilket följer SRP-principen.

CarsInOut används både i CarTransport och Workshop, där båda kräver lagring av bilar. Genom komposition används det i båda klasserna.

CarTransport och **Scania** är subklasser till Truck och har samma fördelar som de andra modellerna.

Workshop lagrar bilar på verkstaden med hjälp av CarsInOut.

Uppgift 4: Ny design

I uppgift 3 identifierade vi några förbättringsförslag som gör koden mer förutsägbar och strukturerad. Den ena handlar om att göra DrawPanel så att den enbart målar fönstret. Just nu sköter den styrningen av fordonen också vilket är mer lämpligt att CarController gör.

Alt. 1:
Dessutom för att minska beroendet mellan CarView och CarController kan man låta CarView ha de fullständiga implementationerna av vad knapparna ska göra. Då följer man Separation of Concern (SoC) och MVC-design praxis eftersom Controllers och Views inte båda bör bero på varandra samtidigt. Denna förändring innebär att komposition från CarView till CarController inte längre krävs (en pil försvinner i UML).

Alt. 2 (som även implementerar Alt. 1):
För att ytterligare följa principerna kan man helt göra om kopplingarna genom att införa en Mainklass som ansvarar för allting programmet ska göra. Detta innebär att allt som sker i CarView, DrawPanel, CarController, sker genom Mainklassen. Förändringen följer också en typisk MVC-design där all källkod tillhör modellen istället för i Controllers och Views. Därmed följer designen SRP och SoC principerna samtidigt som det blir högre cohesion och lägre coupling.

DrawPanel, CarController och CarView borde man kunna förändra parallellt. Däremot måste implementationen av Mainklassen göras enskilt, eftersom den beror av alla.