Uppgift 2: Beroenden

- **Rita upp ett UML-diagram**
    - Se draw.io
- **Analysera de beroenden som finns med avseende på cohesion och coupling, och Dependency Inversion Principle.**
    - Vilka beroenden är nödvändiga?
        - Det är nödvändigt att ha en superklass, Car som har grundläggande funktionalitet för alla möjliga sorters bilar. Den implementerar interfacet "Movable". Funktionaliteten i Car behövs i alla subklasser.
        - Vidare, finns även Truck som är en subklass till Car. Den tillagda funktionaliteten är att alla Trucks har ett attachment vilket är nödvändigt beroende för att inte behöva implementera sådan funktionalitet varje gång. Beroende på Truck-typ finns det olika "flak" som implementerar interfacet attachment beroende på vad som krävs för det specifika "flaket".
        - Loadable, CarShop och CarTransporter beror av Car för att dessa endast får lasta objekt av typ Car.
        - Det måste gå att skapa bilar i CarController, för att kunna köra applikationen.
    - Vilka klasser är beroende av varandra som inte borde vara det?
        - CarController och CarView är just nu beroende av varandra vilket anses vara felaktigt.
        - CarController bör inte bero på subklasserna till Car.
        - DrawPanel bör inte bero på CarShop.
    - Finns det starkare beroenden än nödvändigt?
        - CarController och CarView bör inte bero på varandra och använda varandras metoder i båda klasserna. Det är en violation mot High Cohesion, Low Coupling.
    - Kan ni identifiera några brott mot mot övriga designprinciper vi pratat om i kursen?
        - Single Responsibility Principle:
            - CarController skapar både bilar och ansvarar för logik i utför händelser för bilarnas beteende. Detta bör separeras så att exempelvis en Factory skapar bilar medan CarController ansvarar för att kontrollera dessa.
            - CarView både skapar knappar och implementerar logiken för vad knappar ska göras. Bör separeras.
        - Open Closed Principle:

            - Finns en del hårdkodad logik i programmet, kanske är det värst i CarController. Det bryter mot OCP.
        - Dependency Inversion Principle: Högnivåmoduler som CarController bör inte bero på Lågnivåmoduler såsom Bilklasserna.

Uppgift 3: Ansvarsområden

- **Analysera era klasser med avseende på Separation of Concern (SoC) och Single Responsibility Principle (SRP).**
    - Vilka ansvarsområden har era klasser?
        - Car: Har huvudfunktionaliteten som är gemensam för alla typer av fordon, exempelvis gas och brake.
        - Volvo240: Innehåller funktionalitet som är specifik för en Volvo240, exempelvis att den har en trim-modul och 4 dörrar.
        - TrimmedBehaviour: Innehåller funktionalitet som definierar hur en trim-modul fungerar.
        - Saab95: Innehåller funktionalitet som är specifik för en Saab 95, exempelvis att den har en turbo och 2 dörrar.
        - TurboBehaviour: Innehåller funktionalitet som definierar hur en turbo fungerar.
        - Truck: Är en subklass av Car som beskriver specifik funktionalitet för alla fordon som är lastbilar.
        - Loadable: Definierar funktionalitet som behövs för något som kan lasta en bil.
        - Flatbed: Skapar en “attachment” som kan höjas och sänkas för att exempelvis lasta bilar.
        - Platform: Skapar en “attachment” som kan höjas och sänkas för att exempelvis lasta gods.
        - Scania: En subklass av truck som har funktionalitet som är specifikt för Scania-lastbilar, exempelvis att de har platform.
        - CarTransporter: En subklass av truck som har funktionalitet specifikt för billastare, exempelvis att de har flatbed.
        - CarShop: Beskriver funktionalitet som krävs för att bygga en bilverkstad.
        - CarController: Den övervakar i vilket tillstånd de olika bilarna befinner sig i och flyttar på bilarna. Skapar även bilar som används i applikation.
        - DrawPanel: Hämtar in och ritar upp bilderna för respektive bil och verkstad.

        - CarView: Skapar alla knappar som används i applikationen. Implementar även logiken för vad knapptrycken ska göra.

    - Vilka anledningar har de att förändras?
        - Främst är det tydligt att både CarController och CarView har fler än ett ansvarsområde. CarController både skapar bilar och sköter tillståndet för bilarna. CarView både skapar knappar till applikationen och implementerar dess logik.
    - På vilka klasser skulle ni behöva tillämpa dekomposition för att bättre följa SoC och SRP?
        - De klasser som inte ingick i Lab 2, d.v.s. CarController, CarView och DrawPanel behöver dekomposeras för att förtydliga vilka ansvarsområden som hör till vilken klass. CarController och CarView bör delas upp i minst två klasser var för att dela på ansvaren. DrawPanel har det tydligaste ansvarsområdet, men i klassen skapas en CarShop vilket går att förbättra.

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.**
    - Se draw.io.
- **Motivera, i termer av de principer vi gått igenom, varför era förbättringar verkligen är förbättringar.**
    - Genom att dela upp klasserna CarController och CarView till flera klasser, har dessa nu ett ensamt ansvar. Det följer då Single Responsibility Principle.
    - Genom att lägga till en klass CarGame får vi bort att CarView och CarController behöver bero på varandra. CarGame körs istället för CarController. Detta följer High Cohesion, Low Coupling.
    - Genom att skapa en CarFactory, blir ansvarsområden för CarController och DrawPanel tydligare då dessa inte längre skapar instanser av bilar/verkstäder.
- **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. Se Övning 3: UML, static vs dynamic för ett exempel på en refaktoriseringsplan.**
    1. Skapa en ny klass CarFactory som gör det möjligt att skapa bilarna.
    2. Skapa en ny klass CarGame som initialiserar bilarna, verkstaden och har komposition med CarController och CarView. Flytta ut skapandet av bilar från CarController och verkstaden från DrawPanel.
    3. Bryta ut EventHandler från CarView för att förtydliga ansvarsområden.

4. Bryta ut TimerListener från CarController för att förtydliga ansvarsområden.

- **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?**
    - Steg 1 och steg 2 går dock inte att göra parallellt med varandra då CarGame behöver CarFactory för att skapa instanser av bilarna. Denna funktionalitet behövs först.

      Steg 3 och 4 går att göra parallellt med steg 1-2 och även med varandra då de endast påverkar CarView och CarController internt.