Uppgift 2:

* Analysera de beroenden som finns med avseende pa cohesion och coupling, och Dependency Inversion Principle.
  + Vilka beroenden är nödvändiga?

CarController -> Vehicle

CarController -> Saab95

CarController -> Scania

CarView -> DrawPanel

CarView -> CarController

Antingen (Timelistener -> CarController -> CarView -> DrawPanel) eller (Timelistner -> CarView)

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

CarController -> CarView

Timerlistener -> Collision

Collision -> Vehicle

-Beroende på om Collision bakas in i CarController

TimerListener -> CarController

* + Finns det starkare beroenden än nödvändigt?

Timelistner -> CarController -> CarView -> DrawPanel

CarController -> ArrayList

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

Open-Closed Principle och Liskov substitution principle: Mycket hårdkodat, finns metoder och klasser som tar emot specifika objekt där parametrisering hade tillåtit större användningsområden (tex checkCollision(Vehicle)) och TimerListener. Modularitet: Collision klassen hade kunnat vara oberoende och i så fall vara tillgänglig för andra användningsmöjligheter.

Uppgift 3:

* Analysera era klasser med avseende pa Separation of Concern (SoC) och Single  
  Responsibility Principle (SRP).
  + Vilka ansvarsområden har era klasser?

CarController: Har main() så den ansvarar bland annat för att programmet körs. Exekverar metoder utifrån användarens input. Skapar listan med Vehicles som används ofta. Skapar en timer för att “hålla koll” på tiden.

Carview: Styr det grafiska gränssnittet som användaren ser.

DrawPanel: Uppdaterar det grafiska gränssnittet visuellt med vad som händer i CarController.

Collision: Håller koll om objekten (bilarna) stöter i kanterna i den visuella representationen. Inverterar bilarnas riktning.

TimeListener: Kollar alltid var bilarna befinner sig och informerar drawpanel vart den ska måla ut dom nya bilarna.

* + Vilka anledningar har de att förändras?

För att bättre följa ett “High cohesion, low coupling” mönster.

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

Drawpanel borde delegera till timelistener för att få en uppdaterad position. Carcontroller borde delegera till collision för att veta om en krock sker. Collision borde delegera till timelistener för att kolla om bilarnas x och y värde är “out-of-bounds”.

Uppgift 4:

* **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.

Klasserna får ännu tydligare ansvarsfördelning och beroende-struktur som följer MVC närmre genom att separera M, V, C och Applikationen. Vi ökar modulariteten i programmets övergripande struktur enligt OCP, sänker coupling och ökar cohesion.

* **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.

Skapa en ny class **Application** som får som ansvarsområden att sköta uppstart och uppdatering av programmet. ***TimeListener*** och ***main*** hamnar inuti **Application**.

Arraylistan med cars som finns i **Application** ska skrivas om till en parametriserad lista med statisk typ Collection för att göra ett svagare beroende som extendar vehicles. För att undvika castingen som vi har för subklass-specifika metoder just nu.

**Collision** ansvarar för villkoren för en krock. **Application** uppdaterar hela tiden och delegerar ansvaret för en krock till **Collision**. Villkoren för en krock skall ändras då dom nu är hårdkodade.

**CarController** ska bara ha inputs för styrning. (likt en kontroller)

* 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?

Man kan arbeta med de följande två delarna separat, så länge man vet vad man har för specifikationer på vardera del så bör man kunna arbeta parallellt och sedan koppla ihop klasserna tsm.

* Del 1:
  + **Application**:
    - Flytta **TimeListner** och ***main*** funktionen från **CarController** till en ny klass **Application.**
    - Använd **Collision** och **CarView** genom komposition
    - Kontrollera och utför de ändringar som krävs för att **Application** skall bibehålla funktionaliteten som förväntas med **TimeListner** och ***main*** funktionen genom den nya strukturen.
* Del 2:.
  + **CarController:**
    - Flytta ut den nestade klassen **Collision** till en egen klass
    - Använd en lista med vehicles från **Application,** och **Collision:s** funktionalitetgenom komposition för att utföra **CarController**:s metoder med bibehållen funktionalitet
    - **CarController** skall inte längre ha en instans av **CarView** klassen