Uppgift 2:

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

- Vilka beroenden är nödvändiga?
    - Scania, Volvo240 och Saab95 ärver från Car
    - Att Car implementerar Movable för att separera rörelse-logik och bilens implementation
- Vilka klasser är beroende av varandra som inte borde vara det?
    - CarController har en direkt koppling till både transport och verkstad. Detta gör att den hanterar flera olika ansvarsområden, vilket kan bryta Single Responsibility Principle
    - Transport<T> verkar ha en stark koppling till Car. Om Transport<T> behöver känna till detaljer om Car, kan det vara bättre att använda ett mer generiskt interface istället för en direkt koppling.
- Finns det starkare beroenden än nödvändigt?
    - **CarController och CarView är beroende av varandra när de inte borde vara det då de har olika uppgifter.**
- Kan ni identifiera några brott mot övriga designprinciper vi pratat om i kursen?
    - Om CarController direkt manipulerar CarView, bryter det mot separationsprincipen (separera logik från UI, SoC).
    - Law of Demeter: Avstå från att kalla interna metoder av andra objekt eller klasser. Ex. CarController tillkallar frame.drawPanel.repaint(), detta bryter mot Law of Demeter då det kallar till något som inte tillhör CarController.
    - Single Responsibility Principle, Open/Closed Principle (uppgift 3 and 4).

Uppgift 3:

Ansvarsområden samt anledningar

- ➔ **Car**: ansvarar för egenskaper för alla gemensamma bilar
  - ◆ Anledning: ändra egenskaperna för alla bilar
- ➔ **Transport**: ansvarar för alla gemensamma transport (med ramp)
  - ◆ Anledning: ändra egenskaperna för alla transport
- ➔ **Verkstad**: ansvarar för egenskaper för alla gemensamma verkstad
  - ◆ Anledning: ändra egenskaperna för alla verkstad
- ➔ **CarController**: binder ihop bilen och CarView med varandra samt kollisionshantering. Den ansvarar även för tidsbaserade event och kallar olika metoder samt uppdaterar frame.
  - ◆ Anledning: ändras då bilar, metoder och kollision hanteringen behöver uppdateras. Men även vid hantering av verkstad och laddning av bilar
- ➔ **CarView**: ansvarar för gränssnittet, såsom knappar, storlek och events.
  - ◆ Anledning: ändra eller lägga till element till gränssnittet. Även för att hantera olika events och deras kopplingar.
- ➔ **DrawPanel**: Ansvarar för positionering, upphämtning av bilder, rendering och för verkstaden.
  - ◆ Anledning: ladda, lagra och positionera bilder. Ändra verkstadens position eller bild. Eller för att helt enkelt ändra bakgrundsfärg och lägga till fler bilar.

- ➔ **På vilka klasser skulle ni behöva tillämpa dekomposition för att bättre följa SoC och SRP?**
  - ◆ CarView: initComponents initializes ui elements as well as adds actionlisteners. The problem here is the fact that it does too much, such as, it initializes all GUI elements as well as sets their colors and size. A solution to this is refactoring it to smaller methods instead so one method doesn't do everything, breaking SRP and SoC.

  - ◆ CarController: This class contains the main method which should be in its own separate class. CarController also currently breaks SoC by correcting and adjusting collision mechanics. The class also contains a nested TimeListener class which breaks SRP (high coupling). Move repaint call from CarController as this responsibility isn't CarControllers.

  - ◆ DrawPanel: DrawPanel has multiple tasks, such as, retrieving and rendering images as well as positioning them. Dedicated classes for achieving these tasks will introduce modularity and reusability.

Uppgift 4

Motivering kring ny design

- SRP
    - Single Responsibility Principle means that a class should only be responsible for one thing and nothing else. This is the reason for why the DrawPanel class is moved to CarView as to only have a single reason to change files. The previous design needed to change two files instead of one. This is also the reason to separate main into its own class as it has the responsibility of starting the program which is its sole responsibility.
- SCP & OCP
    - Separation of Concern, focuses on decomposition and making methods distinct and easy to work with. This also increases readability and makes it easier to debug by separating a method which does multiple things into separate methods. Such a case would be initComponents, this also follows Open/Closed Principle as by refactoring into multiple methods the code can be reused and easily extended.
- MVC
    - The reasoning behind the new design is modularity and following the MVC-model which consists of Model View Controller, which was why the DrawPanel class was nested into the CarView. Another reason for this is SRP which means that a class should only have a single purpose to exist. The previous design was limiting as it introduced 2 different classes for one purpose.

Refaktoriseringsplan

1. Refactor main() into its own class
2. Refactor the code initComponents method in CarView into separate methods based on function
3. Refactor collision logic into an abstract method in Car-class as that's shared among all cars (even transports)
4. Move DrawPanel-class into CarView
5. Refactor eventhandler in CarController into separate class

6. Move repaint() from CarController to CarView to avoid hard dependance to DrawPanel
7. Refactor methods in DrawPanel into own method

Parallellisering kring arbete

The refactoring of initComponents and collision logic can be done independently as well as the refactoring of DrawPanel which can be handled by another developer. So to answer the question, yes, the plan can be executed in parallel.