UML PRACTICAL NOTES:

(Choose to not include JFrame, JPanel, Graphics and all those things for sanity reasons)

ArrayList<Truck> randomInstanceAttribute = new ArrayList<>(); // is implemented in UML as have have two Has-A arrows, one to ArrayList and one to Truck.

For nested class (i.e TimeListener in CarController), we decided that if it uses instance attributes from the class it is defined in, this only converts to a usage-dependency for TimeListener to these classes (although still Has-A for CarController for example)

QUESTION FOR NIKLAS: the things we do in the "public static void main(String [] args)... " method in a class, how do they affect the beroenden of the class (i.e the class where the main function is within).

Learnings: If a class (A) has composition to another class (B), for which that other class uses or has komposition to other classes(C, D, E, …) , then that doesn't necessarily force a beroende mellan the first class (A) and C, D, E etc. It only does if A uses the gränssnitt (ex methods etc) (if it is aware about the existence of C, D, E etc), then A should have arrows to the ones it "knows if its existence" (and then not "Has-A, but usage-dependency).

EV ADD TIMER

LABB 3

Uppgift 2 Questions:

Analysera de beroenden som finns med avseende på cohesion och coupling, och Dependency Inversion Principle (bero på abstraktioner, inte implementationer).

Vilka beroenden är nödvändiga?
De nödvändiga beroendena är riktade mot abstraktioner, såsom interfaces, snarare än konkreta implementationer (arv).

Vilka klasser är beroende av varandra som inte borde vara det?
För närvarande identifieras ett oönskat beroende mellan CarController och DrawPanel, där Law of Demeter bryts. Vi kan lösa detta genom att använda CarView

som en mellanhand och ta bort den direkta kopplingen. Vidare finns ett ömsesidigt beroende (mutual dependency) mellan CarView och CarController, vilket är suboptimalt. Det rekommenderas att endast CarView ska ha ett beroende på CarController, och att logik ska centraliseras i CarController.

Finns det starkare beroenden än nödvändigt?
Det förekommer fall där subclassing används där en komposition tillsammans med interfaces skulle vara mer fördelaktigt, exempelvis mellan Vehicle och dess underklasser som Car, Truck, Saab, Volvo, och Scania. Trots att subclassing underlättar användning av polymorfism genom parametrisering med V extends Vehicle, skulle en omstrukturering till en modell baserad på komposition och interfaces bibehålla denna flexibilitet utan att skapa onödigt starka beroenden. Detta kan ske genom att V extends VeichleInterface. Dock anses inte fördelarna med en sådan omstrukturering vara tillräckligt övertygande för att motivera en refaktorisering.

Kan ni identifiera några brott mot övriga designprinciper vi pratat om i kursen?
En överträdelse av Dependency Inversion Principle (DIP) och Law of Demeter identifieras i CarController där direkt åtkomst till frame, en närliggande klass, leder till interaktioner med drawPanel, en mer avlägsen klass ("stranger"). Sådana direkta anrop till avlägsna klassers metoder strider mot rekommendationerna om att endast interagera med nära associerade klasser för att upprätthålla låg koppling och hög sammanhållning. Denna praxis bör revideras för att bättre följa principerna om objektorienterad design.

```java
frame.drawPanel.prePaint(tripleImagePointCar.getAllFirstElements(), tripleImagePointCar.getAllSecondElements());
frame.drawPanel.repaint(); // repaint() calls the paintComponent method of the panel
```

Sedan såg vi att Single Responsibility Principle (SRP) bröts mot i DrawPanel, som både ansvarade för att måla ut bilarna och hålla koll på deras positioner och bilder. Detta skedde genom moveIt() funktionen och att skapa instanser av bilderna som attribut. Att hålla koll på positioner och bilder borde vara en del av logiken i programmet, och därmed tillhöra modellen (CarController). Vi löste detta genom att sköta logiken med i CarController och endast skicka bilder och positioner till DrawPanel, som numera bara ansvarar för att måla ut bilderna på rätt plats. Sedan bryts fortsatt SRP när det kommer till dessa tre klasser då ansvar inte helt är fördelat i separata områden på ett tillräckligt gott sätt.

Uppgift 3:

Analysera era klasser med avseende på Separation of Concern (SoC) och Single Responsibility Principle (SRP).

Vilka ansvarsområden har era klasser?

Som sagt tidigare har ansvarsområdena inte varit optimala enligt SoC och SRP. Vi har sett att CarController har använts för att uppdatera DrawPanel vilket tämligen är underligt då dess syfte endast borde vara att arbeta med det tekniska underliggande, medan CarView och DrawPanel skall samverka för att framställa det visuella.

Vilka anledningar har de att förändras?

Uppdelningen av ansvar har stora anledningar att förändras så att SoC och SRP kan tillämpas. Vi behöver göra en tydligare uppdelning av ansvarsområdena så att alla klasser ansvarar för en sak (SRP). Därmed tycker vi att följande uppdelning är lämplig:

- **Model**: CarController (ansvarar för logiken)
- **Controller**: CarView (ansvarar för att koordinera logiken med grafiken, intermediary mellan CarController och DrawPanel.)
- **View**: DrawPanel (ansvarar för grafiken)

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

- **CarController**
    - För att bättre följa SoC och SRP bör CarController strikt fokusera på logiken bakom bilarnas beteende, såsom rörelse, acceleration och inbromsning. All direkt interaktion med GUI-komponenter, såsom DrawPanel, bör separeras ut.
- **DrawPanel**
    - DrawPanel bör enbart fokusera på grafikrepresentationen, d.v.s., att måla ut bilarna på skärmen baserat på information den får.
- **CarView**
    - CarView bör tydligare definieras som kontroller mellan modellen (CarController) och vyn (DrawPanel). Detta skulle innebära att CarView hanterar användarinteraktioner (exempelvis knapptryckningar) och vidarebefordrar dessa till CarController, som sedan uppdaterar modellen. Samt interagerar med DrawPanel för att måla ut dessa ändringar.

Uppgift 4

Refaktoreringsplan:

1. **Förbättra CarController**

    a. För att bättre följa SoC och SRP bör CarController strikt fokusera på logiken bakom bilarnas beteende, såsom rörelse, acceleration och inbromsning. All direkt interaktion med GUI-komponenter, såsom DrawPanel, bör separeras ut. T.ex. skall inte CarController initiera en instans av CarView utan endast tvärtom, och CarView kommer istället att fetcha information från CarController.
    b. Genom att ta ut TimerListener och lägga som en egen klass kan vi ha att t.ex. CarController har komposition till denna klass och delegerar och att t.ex. andra klasser använder TimerListener vid behov, om det följer MVC. Det svåra här blir att förutspå hur vi strategiskt lägger upp funktionalitet som nu bygger på att TimeListener har tillgång till alla CarControllers instansattribut för en given instans. Hur vi exakt gör med alla dessa beroenden (som implicit dagsläget följer från CarController, t.ex. usage dependency till TripleManager, GenericWorkshop, Volvo240 och Vehicle). Eventuellt kan vi ta oss runt och bli av med dessa beroenden mellan dessa som egentligen kommer från den ursprungliga/rena CarController delen i CarController. Förhoppningsvis kan beroenden som dessa undvikas och därav ge oss lägre coupling i slutändan, såväl som högre cohesion.
    c. Ta bort alla DrawPanel-instans calls från TimeListener således att detta endast görs i View möjligen, om det går på ett fungerade sätt. Det blir ett sätt att skilja på V och C, men vi måste fortfarande kunna kommunicera mellan C och V.

2. **Förbättra CarView**
    a. CarView bör tydligare definieras som kontroller mellan modellen (CarController) och vyn (DrawPanel). Detta skulle innebära att CarView hanterar användarinteraktioner (exempelvis knapptryckningar) och vidarebefordrar dessa till CarController, som sedan uppdaterar modellen.
    b. Samt interagerar CarView med DrawPanel för att måla ut dessa ändringar. D.v.s kan vi även använda en ActionListener override variant i CarView för att uppdatera vår instans av DrawPanel (drawpanel) efter att modellen i CarController har uppdateras. d.v.s att när CarView upptäcker en state-förändring i CarController så triggar det en listener som då gör ett call till drawpanel.prePaint och drawpanel.repaint osv …, snarare än att detta görs i CarController.
    c. En annan approach som fungerar är att CarView kan hämta “the state” av CarController vid varje frame av programmet.
3. **Ändra signaturer**
    a. Vi bör se över åtkomstnivåerna för att säkerställa att alla metoder är korrekt exponerade.

Förändringarna handlar om kopplingen mellan två moduler så det är inte så mycket parallellt arbete. Metodsignaturerna kan dock ske parallellt.Utöver detta finns det lite små aspekter som vi kan rensa upp i för att bättre uppfylla MVC-pattern, vilket förekom i kursen först efter ordinarie deadline för lab 3.

TODO:
I UML: *ta bort ArrayList, Stack och färgade grejer, lägg till lite usage dependencies, uppdatera UML utefter ändringar i BedComponent och BedAngleComponent grejer osv (skickar via Truck osv). Skriv upp att ta ut timerlistener från CarController som ett refaktoriseringssteg som en egen klass som alla kan nå, eller vissa komponenrar till en timerlistener osv. Lägg även till JFrame, JButton och alla sådana i UML och dess kopplingar.*

#Ev change relation/arrow between Scania and BedWithAngleComponent to usage only as well as the same for CarTransporter and BedComponent. Maybe add komposition arrow from Scania and CarTransporter to BedInterface also. Should BedInterface have composition arrow from Truck? That sort of makes sense

JFrame
JButton
JPanel
JSpinner
JLabel

LABB 4

Uppgift 2

Vilka avvikelser från MVC-idealet kan ni identifiera i det ursprungliga användargränssnittet? Vad borde ha gjorts smartare, dummare eller tunnare?

DrawPanel - som vi nu döpt om till CarView - innehåller logik, vilket inte överensstämmer med SRP. DrawPanel bör vara dumb (och thin) t.ex. skall den inte innehålla implementation kring gasSprinner vilket den i dagsläget gör - något som bör flyttas till controllern, som förvisso skall vara thin men ändå säga hur knapptrycket/styrsignalen skall konverteras och ges som signal till modellen.

CarView - nu omdöpt till CarController - istället för att skapa en instans av CarModel så borde den lyssna på den genom ett interface, vilket kommer att administreras i applikationen (A), CarApp där CarView läggs till som en observer (och därav också måste implementera ObserverInterface) till vår observable CarModel.

CarController - numera CarModel - skall inte skapa en instans av CarController, och bör inte kommunicera med DrawPanel eller direkt till CarView. Model i MVC skall inte riktigt behöva känna till existensen av V eller C (förutom att den har ett ObserverInterface i och med Observer Pattern där den skickar ut radiosignaler om dess state change). Detta kommer innebära att CarModel indirekt kommer att skicka signaler till CarView genom att CarView

läggs till i CarModels observer-lista i Applikationen (CarApp), men eftersom denna lista endast består av <ObserverInterface> objekt kommer CarModel inte behöva känna till CarView som klass, och förlitar sig bara på att den implementerar (åtminstone) det gränssnitt som ObserverInterface definerar.

Vilka av dessa brister åtgärdade ni med er nya design från del 2A? Hur då? Vilka brister åtgärdade ni inte?

Bristerna med DrawPanel är åtgärdade genom att den tar in bilder och punkter som den ritar ut. Vi har även separerat CarView och CarController på ett bättre sätt där CarController nu är mycket tunnare och vår CarView är mycket dummare, genom att flytta styrsignaler till CarController och låta den endast hantera detta, medan CarView framförallt endast ritar ut till en JFrame och sedan innehåll en del implementation kring hur den uppdaterar sin utritning när den får ett call från Modellen via dess observerlist.

Övriga är inte åtgärdade. Se även kommentarer ovan för en del justeringar.

Rita ett nytt UML-diagram som beskriver en förbättrad design med avseende på MVC.

Uppgift 3: Fler designmönster

Observer, Factory Method, State, Composite. För vart och ett av dessa fyra designmönster, svara på följande frågor:

Finns det något ställe i er design där ni redan använder detta pattern, avsiktligt eller oavsiktligt? Vilka designproblem löste ni genom att använda det?

Observer Pattern: I dagsläget använder vi ett observer pattern i vår MVC-pattern där vi i modellen (M) delvis har en klassisk modell (CarModel) tillsammans med ett interface ObserverInterface (M = {CarModel, ObserverInterface}) som ger ett gränssnitt till vad observers till modellen skall implementera. Genom att låta modellen ha en lista med observers av typen ObserverInterface och en funktionalitet för att lägga till observers till denna lista kan vi genom Applikationen lägga till CarView som observer till listan och låta vår modell (CarModel) uppdatera sina observers vid egenförändring enligt gränssnittet utmålat i ObserverInterface. Därav behöver CarModel inte känna till CarViews direkta existens och vi uppnår DIP samt modularitet i största allmänhet genom att fylla MVC:s krav. Vi hade tidigare mutual dependencies mellan flera M-V-C komponenter och följde inte MVC:s direktiv utan vår implementation av observer pattern.

Factory method: I dagsläget implementerar vi inte factory method i vår kod.

State: Vi implementerar inte state pattern i dagsläget.

Composite: Eventuellt använder vi oss av composite pattern när vi loopar igenom Vehicle och kallar på move på alla vehicles.

Finns det något ställe där ni kan förbättra er design genom att använda någon av dessa design patterns? Vilka designproblem skulle ni lösa genom att använda det? Om inte, varför skulle er design inte förbättras av att använda det?

Observer Pattern: Vi har redan implementerat detta.

Factory method: En factory method skulle kunna implementeras i CarApp. Vi kommer att åtgärda detta. (Åtgärdat inom ramen för uppgift 5).

State: State pattern skulle eventuellt kunna implementeras på vissa delar av Vehicle alternativt för TurboOn för saab, där vi vill switcha mellan olika beteenden, men det bedöms något överflödigt att implementera.

Composite: Efter noggrann övervägning ser vi ingen fördel med att ytterligare använda composite pattern i vår kodbas. Det finns ingen uppenbar nytta som skulle motivera en sådan ansträngning, vilket gör det till en mindre gynnsam investering av våra resurser.