Uppgift 2: Beroenden

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

Cohesion

- Exempel på high cohesion: Movable – allt är relaterat till rörelse
- CarController har ansvar över att vara en brygga mellan modellen (fordonen) och användargränssnittet (CarView).
- CarController hanterar dock vissa grejer relaterade till modellen, såsom position och riktning. Detta kan separeras.
- TimeListener i CarController gör flera olika saker (low cohesion) – rör på fordon, kontrollerar och korrigerar deras positioner och uppdaterar vyn

Coupling

- Våra interfaces definierar abstraktioner som andra klasser kan implementera vilket möjliggör low coupling
- Stark coupling mellan CarView och CarController eftersom de har mutual dependencies
- CarController har ett beroende till drawPanels implementation (frame.drawpanel.repaint()) som är tre steg bort, vilket inte är bra pga Law of Demeter och high coupling mellan olika moduler
- Type casting i CarController skapar beroenden till många andra klasser. Kan dock vara svårt att ändra i nuläget.
- Arvshierarkin mellan vehicle, car, truck etc är nödvändig
- Måste finnas beroenden mellan CarController och Carview samt mellan CarController och Vehicle
- CarController och DrawPanel är klasser som är beroende av varandra men som inte borde vara det.
- CarController och CarView bör inte ha mutual dependencies. Detta är ett starkare beroende än nödvändigt.

Brott mot övriga designprinciper

- Command-Query Separation Principle – En metod ska antingen ha sidoeffekter eller returnera ett värde, inte både och. Unload() funktionen gör både och.
- Law of Demeter. Vid flera platser i koden använder vi metoder av “grannar längre bort”. Ex frame.drawPanel.repaint(); i timerlistener.
- Single Responsibility Principle, Separation of Concern Principle, Dependency Inversion Principle, Open/Closed Principle – mer om dessa i nästa uppgift

Uppgift 3: Ansvarsområden

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

Klassers ansvarsområden och anledningar att förändras

- Vehicle: ansvarar för beteenden och egenskaper som är gemensamma för alla fordon oavsett typ.
    - Anledningar att förändras: ändra beteende eller egenskaper för alla fordon
- Truck: ansvarar för beteeenden/egenskaper gemensamt för lastbilar
    - Anledningar att förändras: samma som vehicle
- Car: ansvarar för beteeenden/egenskaper gemensamt för bilar
    - Anledningar att förändras: samma som vehicle
- ServiceShop: ansvarar för beteeenden/egenskaper gemensamt för serviceshops
    - Anledningar att förändras: samma som vehicle
- CarController: dess ansvar är att vara en brygga mellan modellen (fordonen) och användargränssnittet (CarView). Också att fordonet inte ska gå utanför och då ändra position och riktning på fordonet. Timelistener (hantera tidsstyrda händelser). Hantera kollisioner.
    - Anledningar att förändras: lägga till funktion som klienten kan använda, t.ex. turn left knapp
- CarView: ansvarar för gränssnittet utåt mot klienten. Initialiserar GUI, hanterar knappar, eventhantering.
- DrawPanel: ansvarar för grafiska gränssnittet; bilder och bakgrund samt måla ut det.
    - Anledningar att förändras: vill lägga till en bil, ändra färg på bakgrunden, samt andra grafiska åtgärder.

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

- CarController (checkCollision och frame size). Carcontroller utför i nuläget beteenden som inte bör höra till carcontroller, tex hantera kollisioner och om fordonen åker utanför bild. Carcontroller har alltså mer än ett ansvarsområde och följer därmed inte SoC. Vi planerar att lägga in beteendet för det som sker under systemets gång(hantera kollisioner och fordon inom bild) i modell delen av koden, det vill säga vehicle/car/truck. Dessutom bör main funktionen brytas ut till egen klass, eftersom denna har beroenden till nästan samtliga klasser och startar programmet, och detta bör inte carcontroller ha.
- InitComponents()-metoden i CarView kan delas upp i flera metoder för att bättre följa SRP och SoC. I nuläget gör den flera olika saker.

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. Motivera, i termer av de principer vi gått igenom, varför era förbättringar verkligen är förbättringar.

Refaktoriseringsplan

1. Ta ut TimeListener från CarController så den inte är nästlad
2. Ta bort det direkta beroendet till DrawPanel från CarController genom att lägga frame.drawpanel.repaint() i CarView
3. Flytta checkAndCorrectPosition() till Vehicle (model-modulen)
4. Flytta checkCollision() till Vehicle som en abstrakt metod där varje bilmodell har sin egen implementering
5. Skapa en egen Main()-klass
6. Bryt upp intiComponents till flera metoder med separata ansvarsområden:
    - setupFrame(title);
    - initDrawPanel();
    - initGasPanel();
    - initControlPanel();
    - initActionButtons();
    - finalizeFrame();
7. Lägga DrawPanel som en nästlad klass i CarView och lägga alla referenser till CarController i yttre klassen CarView

Motivering till förändringar

Förändringarna som kommer göras grundar sig framförallt i att göra programmet mer läsbart, återanvändbart och utbyggbart genom att eftersträva high cohesion och low coupling. Förbättringarna baseras på principer som Single Responsibility Principle (SPR), Separation of Concern Principle (SCP), Dependency Inversion Principle (DIP), Open/Closed Principle (OCP) och Liskov Substitution Principle (LSP).

SRP, som innebär att en klass bara ska ha en anledning att förändras, är anledningen till att DrawPanel ska vara en nästlad klass inuti CarView. Båda klasserna har att göra med användargränssnittet och om något av det visuella skulle ändras i den gamla designen behövde båda klasser förändras. Att ta ut TimeListener från CarController följer också SRP.

SCP, som bland annat handlar om att göra programmet modulärt med hjälp av dekomposition, tillämpas på förändringen att dela upp initComponents()-metoden till sex mindre metoder som är mer specifika för sin uppgift. Detta tillvägagångssätt underlättar även för framtida ändringar och utvidgningar enligt OCP.

Genom att ta bort det direkta beroendet till DrawPanel från CarController, genom att bland annat repainta direkt i CarView, minskas kopplingar mellan CarController och den specifika implementeringen i DrawPanel, vilket är i linje med DIP.

Att ha main()-funktionen i en egen klass är en förändring som görs för att få en tydligare separation mellan programmets startfunktion och dess modellogik, vilket är i linje med SRP. Detta underlättar dessutom underhåll då systemet blir mer modulärt och minskar onödiga beroenden.

Förbättringarna görs dessutom för att bättre följa ett design pattern kallat Model–View–Controller (MVC). Vi har då tre övergripande paket, ett för Model (vehicle och dess subklasser och interfaces), ett för View (CarView och DrawPanel) och ett för Controller (CarController).

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?
För att utföra delar av refaktoriseringsplanen parallellt, kan arbetet delas in i två huvudgrupper:

1. **Ändringar av gränssnitt:** Stegen som involverar att flytta repaint() till CarView, bryta upp initComponents och integrera DrawPanel i CarView kan göras parallellt av en grupp som fokuserar på gränssnittsändringar.
2. **Ändringar av modell:** Flytta checkAndCorrectPosition och checkCollision till Vehicle-klassen, där varje bilmodell har sin egen implementering, kan skötas av en annan grupp som arbetar med modelländringar.

Skapandet av en Main()-klass kan hanteras oberoende och parallellt med de andra uppgifterna. Genom att dela upp arbetet på detta sätt kan olika delar av refaktoriseringsplanen utföras samtidigt av olika utvecklare, vilket effektiviserar processen och minskar den totala utvecklingstiden.