En bild som visar text, kvitto, dokument, Parallell

Automatiskt genererad beskrivning

UML-diagram innan refaktorisering.

Analys av UML

* Klasser som Saab95, Volvo240, Scania och CarTransporter är exempel på subklasser till vår abstrakta klass – Car. Genom att använda generalisering kan vi minimera duplicerad kod och i stället låta subklasser ärva gemensam kod. Alla klasser är en typ av ”Car” – men innehar också unika egenskaper, till exempel Scania, med sitt tippbara flak. Det finns alltså ett beroende – usage dependency – där subklasserna måste ha gemensam, vital kod från Car för att fungera. Allt detta bidrar till hög cohesion – alla är en ”Car”, delar på stor mängd kod i klassen Car och unika funktioner hålls lokalt i klasserna. Låg coupling, eftersom unik kod i en subklass inte påverkar en annan subklass under Car. Således följs DIP väl och de beroenden som finns får anses nödvändiga.
* Klassen Garage är en också subklass till Car, men ej ett fordon. Garage tar in en generisk klass ”T”, som måste vara en subklass av Car. Metoderna leaveCar och pickUpCar är anpassade efter subklasser till Car, så ett beroende finns. Frågan är om Garage verkligen ska vara kopplat till Car. Starkare beroende än nödvändigt till Car möjligtvis. Någorlunda hög cohesion/låg coupling då det ändå är relaterat till de andra subklasserna.
* Movable är ett interface med metoder associerat med rörlighet. Är starkt beroende av Car då denna tillhandhåller faktisk kod för metoderna i Movable.
* CarController har flera starka beroenden. CarController är direkt beroende av Car-klassen eftersom denna tillhandahåller ArrayList<Car> som sedermera används av CarController. Ett annat (onödigt, sannolikt) indirekt beroende är Scania-klassen genom ArrayList<Scania>. Utöver detta har CarController flera indirekta beroenden av bilar som används i simuleringen (Volvo240, Saab95 etc.). Föreligger starkt beroende till CarView för att möjliggöra simuleringen.
* CarView är direkt och starkt beroende till DrawPanel för animeringen. CarView förmedlar information till DrawPanel för att möjliggöra animeringen. DrawPanel är dock inte direkt kopplad till CarController. CarController sköter styrning av bilar medan DrawPanel sköter det grafiska – låg coupling, hög cohesion. I nuläget är CarController, CarView och DrawPanel relativt bra upplagda där de har specifika uppgifter som sedan sammanflätas till en fungerande animering. Men ett starkt beroende föreligger som är nödvändigt.

Analys av ansvarsområden – SoC och SRP

* Volvo240, Saab95, Scania och CarTransporter: Tillhandahåller unika egenskaper utöver det gemensamma som parent-klassen Car innehåller. Följer SRP väl då klasserna endast har ansvar för en viss typ av bilmodell och dess unika egenskaper. Ändrar man något i en klass påverkas endast klassen i fråga. Både SRP och SoC uppfylls väl.
* Garage hanterar bilar som är inne på verkstaden. Håller koll på aktuella bilar i verkstaden samt in- och utlämning. Garage är en separat klass dedikerad till detta och således uppfylls SRP och SoC.
* Movable är ett interface vars ansvar är förflyttning av objekten (bilarna) genom sina metoder. Hela klassen är dedikerat till ett enskilt ansvarsområde. Därmed uppfylls SoC och SRP.
* CarController har flera ansvarsområden relaterade till styrning av bilarna. Däribland funktioner som gas, brake etcetera. Hanterar även input från användare och har koll på bilens aktuella position. SRP och SoC uppfylls i viss mån då ansvarsområdena är relaterade – inputs från användare som översätts till styrning och positionering. Att separera dessa funktioner genom dekomposition vore lämpligt för att bättre uppnå SoC och SRP.
* CarView ansvarar för grafiska delar och tar emot inputs från användare. DrawPanel är en del av CarView och båda hanterar grafiska områden. DrawPanel är mer rent grafisk, medan CarView också hanterar inputs från användare. DrawPanel uppfyller SoC och SRP bättre än CarView då DrawPanel endast hanterar grafiska uppgifter. CarView har ett bredare ansvar som sträcker sig från grafik till inputs från användare. CarView bör ses över och möjligtvis delas upp i en grafisk del och en del för kontakt med användare.

Sammantaget bör CarController, CarView och DrawPanel ses över för att uppnå SRP och SoC på ett bättre sätt.

**Refaktoriseringsplan**

* Genom att skapa en ny klass i arvshierkan som ligger över Scania och CarTransporter kan vi minimera duplicerad kod. Scania och CarTransporter är båda lastbilar som har gemensamma funktioner som tippbart flak och identisk Speedfactor. Vi döper klassen till “Truck” och uppnår på så vis förbättring av SRP och SoC.
* Move är väldigt klumpig och stor (+ kodduplicering för carTransporter). Därför ska SoC implementeras. Detta görs genom att skriva flera mindre metoder som har specifika uppgifter, (i move kommer metoderna moveForward och checkWall skapas + carTransporters unika move ska även inkludera en metod för att flytta bilarna på flaket).

**MVC – plan:**

Vi vill simplifiera så att vyer enbart arbetar med vyer, controller med input och module med alla uträkningar.

**Controller:**

CarController ska enbart innehålla ett sätt för programmet att ta in input. Därav ska den innehålla actionlistener och skicka vidare vad som sker till modulen.

**View:**

CarView ska enbart innehålla info om vad som ska visas d.v.s. alla vyer som vilka knappar som ska ritas o.s.v. Här kommer det även innehålla “.pack()” som är för att skapa programfönstret. Annat som actionlistener ska bort.

**Model:**

CarModule ska skapas och ska vara steget mellan input och vyer, input till kod som gör uträkningar (med hjälp av andra filer som car) till vyer som blir uppdaterade med nya positioner.

CarModule kommer innehålla main-koden och därav även timerbitarna som tas från nuvarande CarController.

**Implementerade vi…**

**Observer?**

Vi använder en observer som vi namngav ”ScaniaObserver” som håller koll på Scanias aktuella fart och visar det på skärmen. ScaniaObserver är ett interface som implementeras av vår Scania-klass. Observer är ett bra sätt att övervaka en specifik funktion i koden.

**Factory Method?**

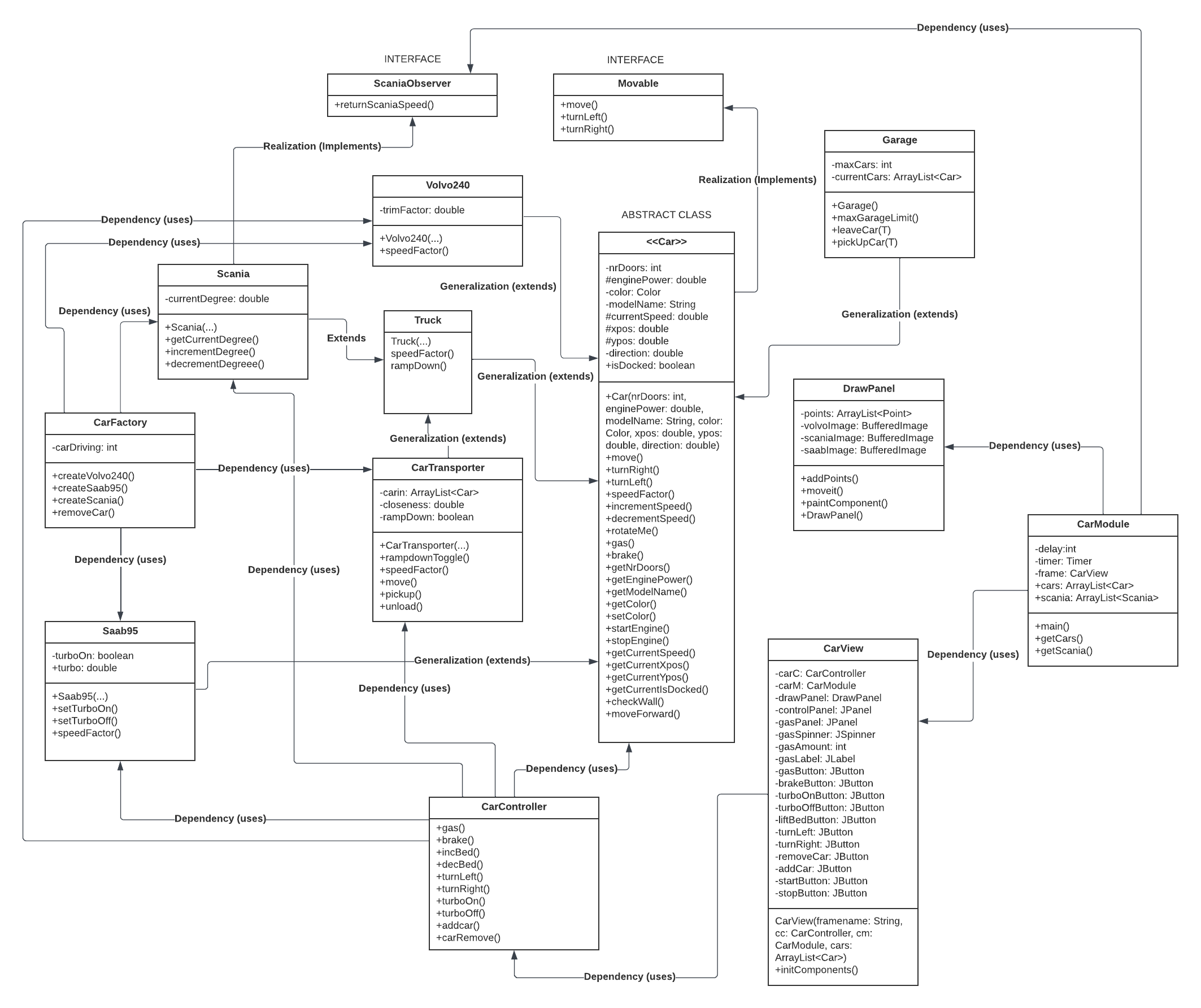
Nej, vi använder oss inte av factory method. Vi skulle kunna använda det för att förbättra hur bilderna väljs och skapar variabler kopplade till specifik bild. Det hade gjort att koden kan köra vilka bilar som än läggs till utan att behöva hårdkoda något extra utöver val av vilka som ska skapas. Factory method hade varit lämpligt om man skulle vidareutveckla spelet.

**State?**

Nej, vi använder oss inte av state. Vi hade kunnat använda oss av state när vi exempelvis har en bil som är “docked”, och därav ska vara i stadiet att bara följa efter CarTransporter utan att kunna göra något annat. Sedan kan även subklasser till Truck använda ett state pattern ifall de har rampen nere eller ej. Dessutom skulle det uppnå mer SRP genom att implementera dessa. Vår kod är i nuläget fullt fungerande och uppnår sina mål, så vi valde att ej implementera state.

**Composite?**

Nej, vi använde oss inte av det. Composite ger inget som vår kod inte redan uppfyller. Det kanske hade funkat att implementera för att se totala bränsleförbrukningen för alla bilar men det var inte uppgiften så vi lämnar det så.



UML-diagram efter refaktorisering.