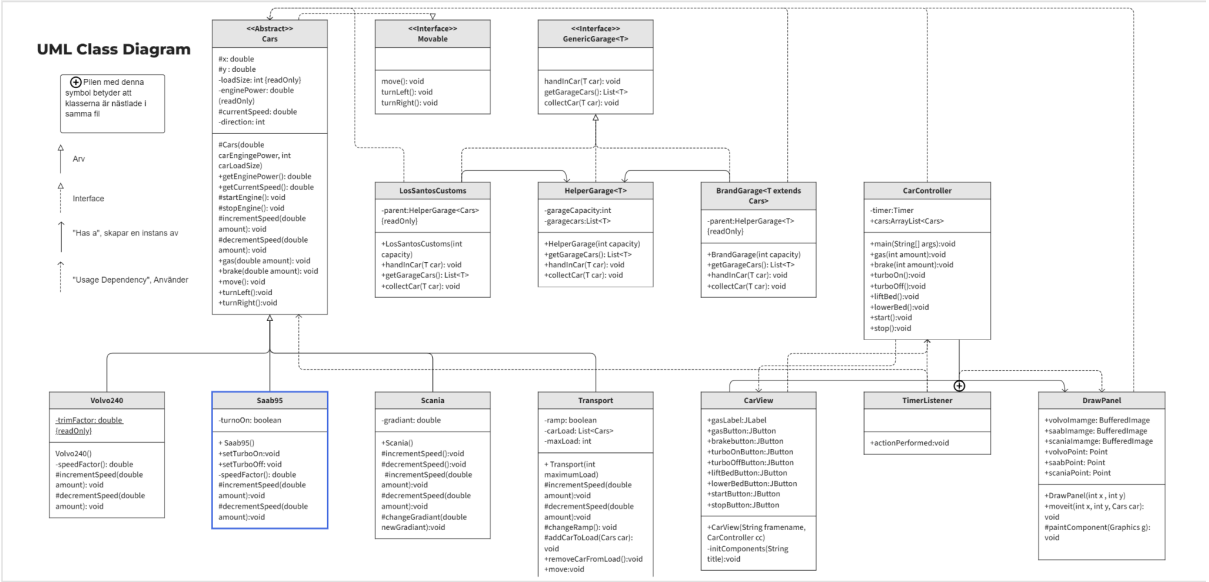
Uppgift 2: (av Oskar Andersson, Alex Tang och Tom Rex)



Vilka beroenden är nödvändiga?

- Beroenden mellan subklasser till Cars är nödvändiga.
    - Alla subklasser till Cars representerar bilar och därför och därav kan de allmänna funktionerna bryta ut och placeras i Cars, detta följer Polymorfism och Single Responsibility Principle (SRP).

- Beroendet mellan de olika verkstäderna och HelperGarage är nödvändigt.
    - Vissa funktioner i både LosSantosCustoms och BrandGarage går att skriva på ett allmänt sätt för att minimera kodduplicering. Detta uppnås genom komposition och delegering vilket HelperGarage och GenericGarage skapar.

Vilka klasser är beroende av varandra som inte borde vara det?

- DrawPanel borde inte ha ett beroende till Cars
    - Eftersom DrawPanel är beroende av CarController som också är beroende av Cars. Det räcker med att bara CarController har ett beroende
- CarView är beroende av CarController och CarController är beroende av CarView
    - Det bör räcka med en väg istället för att det både är beroende av varandra.

Finns det starkare beroenden än nödvändigt?

- Klassen DrawPanel har ett för starkt beroende av Cars superklassen.
    - DrawPanel är beroende av CarController och Cars. Beroendet som DrawPanel har på Cars är för starkt eftersom det räcker med att DrawPanel är beroende på CarController då CarController redan är beroende av Cars.

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

- Single Responsibility Principle, SRP:
    - Många olika klasser har många olika uppdrag när de egentligen bör ha ett specifikt uppdrag. (Single Responsibility Principle, SRP).
- Separation of Concern , SoC:
    - Koden kan också bli mer förståelig om den görs läsbar med hjälp av att bryta större funktioner i mindre delar. Då hade komplexiteten minskat.
    - Koden består också av “dåliga” namn som kan förbättras så att de förklarar mer. Exempelvis turboOn som visar om Saabens turbo är på eller av, medan namnet låter som om att funktionen endast behandlar turbon som på.
    - De olika fordonen har funktioner och modifikation som bör vara mer allmänna (turbo, trimfactor och lyftfunktioner). Dessa bör istället implementeras i interface som bilarna använder. Med hjälp av komposition och delegering kan även rätt bilar få tillgång till rätt funktioner så att exempelvis en Volvo inte kan få en turbo.

Uppgift 3:
Ansvarsområden?

- Cars
    - Det är en superklass som allmänt delar ut hur en bil ska vara uppbyggd och innehåller allt gemensamt mellan de olika bilarna.
- Volvo 240
    - Skapar objektet Volvo240.
    - Den använder superklassen Cars för att få den allmänna strukturen.
    - Introducerar en trimfactor vilket är unikt för Volvo240.
- Saab95
    - Skapar objektet Saab95.
    - Den använder superklassen Cars för att få den allmänna strukturen.
    - Introducerar en turbo vilket är unikt för Saab95.
- Scania
    - Skapar objektet Scania.
    - Den använder superklassen Cars för att få den allmänna strukturen.
    - Introducerar en lyftfunktion vilket är unikt för Scania.
- Transport
    - Skapar objektet Transport.
    - Den använder superklassen Cars för att få den allmänna strukturen.
    - Introducerar en lista över de bilar som finns på flaket, denna lista samt dess tillhörande funktioner är unika för Transport.
- BrandGarage
    - Ser till att bilarna som sorteras i rätt listor, så att de inte blandas. Resten av funktionerna delegeras till HelperGarage.
- HelperGarage
    - Ger de tomma funktionerna i interfacet GenericGarage något innehåll. Samt så ser den till att BrandGarage och LosSantosCustoms har tillgång till dessa funktioner.
- LosSantosCustoms
    - Ser till att alla bilar hamnar i samma lista, så att de blandas. Resten av funktionerna delegeras till HelperGarage.
- CarController
    - Skapar en delay.
    - Skapar en lista över bilar vilket sedan används för att uppdatera bilarnas position.
    - Ber att CarView ska rita ut en frame.
    - Sist har Carcontroller en ActionListener vilket lyssnar på de knappar som Carview har ritat upp på skärmen, CarController ser även till att rätt sak händer när de har blivit tryckta.
- CarView
    - Skapar fönstret och alla knappar i fönstret.
- DrawPanel
    - Räknar om bilarnas position, ser till att bilarna håller sig på skärmen samt att den laddar om skärmen så fort något har ändrats.
- TimerListener
    - För varje fordon som är aktiv (I cars listan) så ser TimerListener till att de omräknade för deras position. TimerListener ser också till att rePaint-funktionen i DrawPanel uppdateras.

Vilka anledningar har de att förändras?

- Trim/turbo på Volvo240 och Saab95 borde istället implementeras i ett interface som heter Modifications och en mods-klass som är en komposinerad klass av interfacet bör skapas som blir en faderklass till de bilar som ska implementera olika modifikationer. Anledningen till detta är att om det skulle skapas en ny klass/bil som till exempel Peugeot så skulle Peugeot enkelt kunna implementera Turbo, Trim eller både Turbo och Trim med hjälp av det nya interfacet.

- Superklassen Cars kan döpas om till Vehicle och sedan användas som en superklass till alla möjliga fordon. Cars består enbart av funktioner som är allmänna för alla fordon och därför bör den döpas om. De klass -och funktionsnamn som består av ordet Car ska bytas till Vehicle.

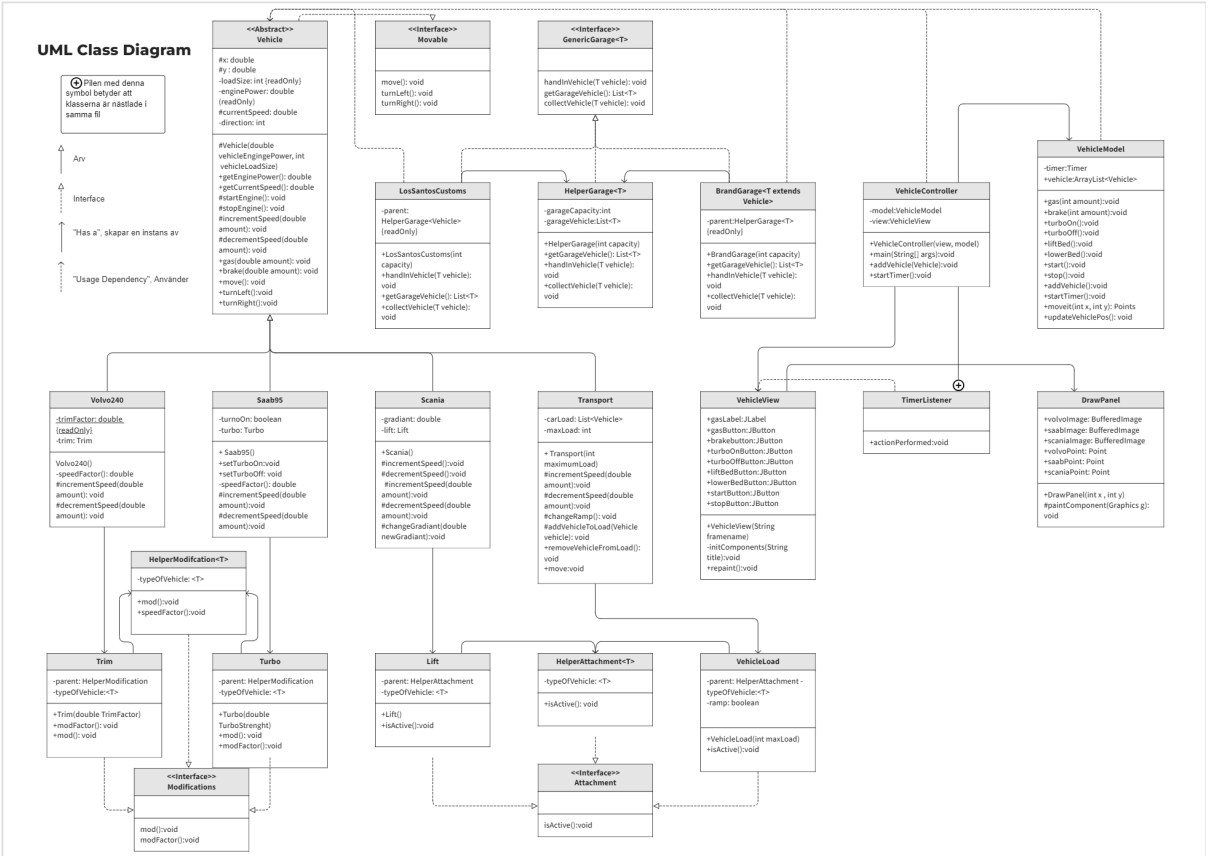
- Likt turbo och trimfactor i saab och volvo bör även de olika lyftfunktionerna i scania och transport inte definieras i deras respektive klasser, utan de bör istället flyttas ut till ett interface som innehåller olika lyftfunktioner så att andra fordon som behöver en lyftfunktion också kan implementera den utan kodduplicering.

- Koden bör ändras i CarController och CarView eftersom de är kopplade åt båda hållen. Det innebär att i nuläget krävs förändringar i båda klasserna för att vidareutveckla systemet vilket är onödigt.

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

- Det finns många klasser som har fler ansvarsområden än de borde. Exempelvis har klassen CarController fler ansvarsområden än den borde och därför bör den förändras så att den ansvarar över en specifik uppgift. Om klasserna enbart ansvarar för en specifik uppgift kommer koden att bli mer lättläst samt enklare att navigera igenom (SRP).
- Klasserna Volvo240 och Saab95 bör implementera sina modifikationer istället för att skapa dem enskilt för varje bil.
- Klasserna Scania och Transport bör implementera sina lyftfunktioner istället för att skapa dem enskilt för varje bil.

UPPGIFT 4



Motivera varför era förbättringar verkligen är förbättringar.
MVC faktoriseringen är en förbättring eftersom den medför att inga klasser är kopplade åt båda hållen, vilket var ett tydligt problem med tidigare versioner.

Genom komposition och delegering kan modifications och attachments samt deras tillhörande klasser skapas.

Refaktoriseringsplan:

Steg 1 (Byta namn):

- Byt alla Cars och car till Vehicle och vehicle.

Steg 2 (MVC):

- Skapa klassen CarModel
    - Placera om kod i CarController, CarView, CarModel och CarPanel.
    - Ex:

```
private class TimerListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        for (Cars car : cars) {
            car.move();
            int x = (int) Math.round(car.getX());
            int y = (int) Math.round(car.getY());
            frame.drawPanel.moveit(x, y, car);
            frame.drawPanel.repaint();
        }
    }
}
```

```
private class TimerListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        model.updateCarPos();
        CarView.drawPanel.repaint();

    }
}
```

Steg 3 (Interface):

- Skapa interface modifications.
- Skapa en helper-klass (HelperModifications) till modifications som implementerar modifications.
- Skapa en Turbo-klass och en Trim-klass som ska innehålla varsitt objekt av HelperModifications samt de funktioner ur modifications som dessa klasser ska använda.
- Gör detsamma med interfacet attachments.

Finns det några delar av planen som går att utföra parallellt?
Steg 1, 2 och 3 går att utföra parallellt då de förändrar olika delar av systemet som inte påverkar varandra.

Inom steg 1 kan allt utföras parallelt då detta steg endast innebär att byta alla namnen. Cntrl F, replace all.

Inom steg 2 går det inte att jobba parallellt för att det spelar roll vilken ordning som de olika klasserna färdigställs i. CarController måste känna till CarView och CarModel för att fungera och därför kan exempelvis den inte implementeras samtidigt som CarView och CarModel implementeras.

Inom steg 3 går det inte att jobba parallellt eftersom de olika klasserna bygger på varandra, exempelvis måste modifications existera innan andra klasser som ska använda sig av det skapas.