Uppgift 2: Beroenden

-
    - Analysera de beroenden som finns med avseende på cohesion och coupling, och Dependency Inversion Principle.
        - Alla delar av koden som implementar eller extendar något är beroende på klassen eller interfacet den extendar eller implementar. I vår kod så är till exempel superklassen “Vehicle” något som många av våra klasser är beroende av för att kunna ha all information som behövs för att skapa till exempel en volvo eller en truck.
        - Vi upplever att vi har ganska många beroenden (hög coupling) eftersom att en ändring i Vehicle kan leda till att samtliga bil-klasser får ändringar. Däremot så är det inte några cykler där Modul1 är beroende av Modul2 som i sin tur också är beroende av Modul1. Det är inte heller så att om du ändrar något i Vehicle så måste du inte ändra något i samtliga bilar, om det inte är något man vill overrida, eftersom att dom ärver från en superklass och får med den nya ändringen “automatiskt”.
        - Vi tycker att vi har relativt hög cohesion i våra klasser. Varje klass har sina metoder som behövs för att skapa sin bil/truck/workshop med rätt attribut och inte så mycket mer.
    - Vilka beroenden är nödvändiga?
        - Vehicles
            - Alla våra bilar/trucks ärver från vehicle och därför är dom beroende på vehicle och behöver vara det så att vi lätt kan skapa nya typer av bilar från vehicles
        - Loadable
            - Loadable behövs för att kunna hantera dom saker som kan ta emot en load. dvs CarWorkshop och CarTransport. Eftersom att dom är två olika typer som inte ärver från samma superklass behöver vi ha ett interface som kopplar ihop dom gemensamma metoder.
        - Moveable
            - Moveable behövs för att kunna ge metoderna för att förflyttas till dom objekt som ska kunna flyttas. Just nu skulle man kunna ha det i Vehicle, men vi har det i ett eget interface så att man lätt ska kunna skapa nya klasser som kan förflytta sig, utan att vara vehicles, t.ex en cykel, om man skulle vilja extenda programmet mer.
        - Truck
            - Truck är både en superklass och en subklass i vårt program, och den relationen används för att särskilja personbilar från trucks. En truck ska vara för stor för att kunna lastas på en annan truck, och har också gemensamma attribut och metoder
        - HasTruckBed
            - Man skulle kunna ha metoderna i Truck istället för att ha ett interface för om något har en truckbed eller inte, men liknande

Moveable så är det bra att ha som ett interface ifall vi skulle vilja lägga till en truck som inte har en truckbed, eller en personbil som har en truckbed.

- Vilka klasser är beroende av varandra som inte borde vara det?
  - IsPersonalVehicle, IsVehicle är två interfaces som gör väldigt lite, rent funktionellt, och kanske hade gått att göra på ett snyggare sätt. Just nu behöver vi relationen eftersom att vi måste kunna avgöra om det är en vehicle (allt utom CarWorkshop) och om det är en personbil (Volvo och Saab). Vi tycker det känns onödigt att ha två interfaces som gör så pass lite och om vi kan bli av med dom relationerna så hade det varit bra.
  - I CarTransport har vi att T extends Vehicle medans T extends IsVehicle i CarWorkshop. Enligt Dependency Inversion Principle ska de bero på abstraktioner (inte klasser), så vi kanske ska ha den som en implementation av IsVehicle istället.
- Finns det starkare beroenden än nödvändigt?
- Kan ni identifiera några brott mot övriga designprinciper vi pratat om i kursen?

Uppgift 3: Ansvarsområden

- Analysera era klasser med avseende på Separation of Concern (SoC) och Single Responsibility Principle (SRP).
  - Vilka ansvarsområden har era klasser?
    - CarWorkshop kan t.ex. ta olika sorters bilar beroende på hur de klassificeras (generaliserade klasser)
  - Vilka anledningar har de att förändras?
    - Om det skulle behövas lägga till flera bilar: i så fall finns det olika interfaces för att bestämma bilens roll
  - På vilka klasser skulle ni behöva tillämpa dekomposition för att bättre följa SoC och SRP?
    - DrawPanel - koden skulle kunna bli mer flexibel om vi justerade den eftersom den nuvarande moveit metoden inte är särskilt extensible (den blir klumpig och är svår att ändra om det skulle finnas fler biltyper osv).

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.
  - 1. I Truck så har vi metoderna lower och raise, som vi även har i vårt interface HasTruckBed, och det overridas i samtliga klasser som har en truckbed. Vi borde alltså kunna ta bort lower och raise metoderna i Truck, och bara ha kvar dom i HasTruckBed och implementationerna i dom klasserna som har en truckbed. Det borde göra koden snyggare och inte så mycket duplicering av kod.

    - 2. Vi vill ändra i koden så att vi inte längre har ett interface som bara ska ge modelname av bilarna (IsVehicle) och inte heller har ett interface som är tomt och bara används för att se om en bil är en personal vehicle eller inte (IsPersonalVehicle). Vi vill ha längre coupling och tror att detta hade gett oss det. Vi vet inte ännu hur det ska ändras och implementeras dock.
    - 3. I DrawPanel vill vi ändra främst på Moveit-metoden eftersom att den just nu inte är så extensible, och det går just nu inte att ha flera bilar av samma sort som rör sig olika. Det är bättre att göra Moveit mer generell så att man inte behöver ändra lika mycket i koden varje gång man vill lägga till en ny typ av bil. Vi tror också att vi då kan behöva ändra på hur vi tar in bilderna också, men vi är inte helt säkra på vilka problem vi kan stöta på innan vi testar.
    - Några av dessa ändringar vi vill göra går emot Dependency Inversion Principle, genom att inte flytta ut beroenden till interfaces, men vi tycker inte att vårt program som det ser ut just nu behöver så mycket DIP som vi har. Vi ser inte poängen med att ha flera interfaces som mer eller mindre bara används för en check som hade kunnat lösas i superklassen. Programmet skulle potentiellt bli mer extendable om vi har kvar dom interfaces, men då kan man också argumentera för att göra en superklass för och ett interface för Buildings ifall det skulle komma fler byggnader, eller en superklass för olika cyklar som använder sig av moveable ifall vi skulle vilja lägga till cyklar. Vi argumenterar därför att tills vi behöver interfacesen för mer funktionalitet, så har vi inte kvar dom.
- Skriv en refaktoriseringsplan. Planen bör bestå av en sekvens refaktoriseringssteg som tar er från det nuvarande programmet till ett som implementerar er nya design. Planen behöver inte vara enormt detaljerad. Se Övning 3: UML, static vs dynamic för ett exempel på en refaktoriseringsplan.

    1. Ta bort metoderna raise() och lower() i truck, kolla så att allt fortfarande fungerar.

    2. En lösning vi funderade på är om vi skulle kunna använda modelname i vehicle och att carworkshop då skulle vara att modelnamet för en workshop är “model - workshop” eller om man hade kunnat lägga till ett nytt attribut i vehicle helt. I detta steget ska vi kolla upp och testa om det fungerar och om det är en bättre lösning än den vi har just nu med vårt interface.

    2,5. Vi vill ändra så att vi tar bort interfacet IsPersonalVehicle då den inte innehåller något, och det då känns onödigt att ha kvar den. Istället vill vi hitta ett sätt att avgöra om en bil är en personalVehicle i Vehicle klassen.

    3. Kolla upp hur bufferedImage fungerar i DrawPanel och se om vi kan få den att t.ex hämta bilarna ur en lista på bilar vi har istället för att manuellt behöva skriva in varje bil vi har. Detsamma gäller deras points.

- 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?
    - Det går absolut att arbeta oberoende av varandra, däremot borde man inte dela på steg 2 och 2,5 eftersom dom handlar ganska mycket om samma sak och det troligtvis är lättare om en person gör båda dom två stegen av refaktoreringsplanen.

Länk till lucidshart:
https://lucid.app/lucidchart/0ba2b31f-0b58-4bef-a31c-8d6d40afee91/edit?viewport_loc=-691%2C-128%2C2992%2C1466%2CHWEp-vi-RSFO&invitationId=inv_5af7331e-8142-493e-9d2f-277d8e3e5c5a