Inför Laboration 3

Lite repetition:

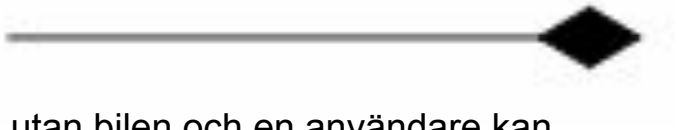
Komposition

- **Arv** är när man designar sina klasser med tanke på vad de **<u>är</u>**.
- **Komposition** är när man designar sina klasser runt vad de **<u>gör</u>**.

Två typer av Komposition:

- Aggregation *(Has-A)*
  - "Has-A"-relation
  - Kan existera oberoende av varandra.
  - Exempel: En skola har elever, båda kan existera oberoende av varan.

- Composotion *(Starkt Has-A / Part-Of)*
  - Kan enbart existera tillsammans med en annan klass.
  - "Gömmer" funktionalitet för användaren.
  - Exempel: En bil har en motor, motorn kan inte existera utan bilen och en användare kan köra bilen utan att veta om motorn. *(OBS! Beror på hur man designar)*



Delegering

- Använd ett annat objekt för att utföra en eller flera uppgifter.
- Det objekt som tar över upp-giften kallas för delegerare.
- En typ av composition.
- En delegerare ska inte ändra befintlig funktionalitet för klassen som den “hjälper”.
- Delegeraren är “gömd” i klassen som den hjälper.

- Exempel: En timmerlastbil har en lyftkran längst bak som hjälper till att lasta timmer på flaket. Men den ändrar inte själva lastbilens beteende, bara utökar.
- *En motor “hjälper” bilen att köra, men är ingen delegerare då den påverkar själva bilens beteende och krävs för att bilen ska fungera.*

Objekt-orienterad design

- Designen utgör underlaget för implementationen.
- Bra design minskar kraftigt tidsåtgången för implementationen.
- Brister i designen överförs till implementationen och blir mycket kostsamma att åtgärda.
- Vanligaste misstaget i utvecklingsprojekt är att inte lägga tillräckligt med tid på att ta fram en bra design.
- Bra design är svårt!
- I allmänhet bör mer tid avsättas för design än för implementation.

Hours of planning can save you days of coding

Symptom på dålig implementation

Synliga tecken i koden att designen är på väg att ruttna:

- Duplicerad kod, ”klipp-och-klistra”-programmering
- Långa metoder
- Långa parameterlistor
- Stora klasser
- Klasser med enbart data
- Publika instansvariabler
- Långa if- eller switch-satser
- Avsaknad av kommentarer
- Onödiga kommentarer
- Oläslig kod

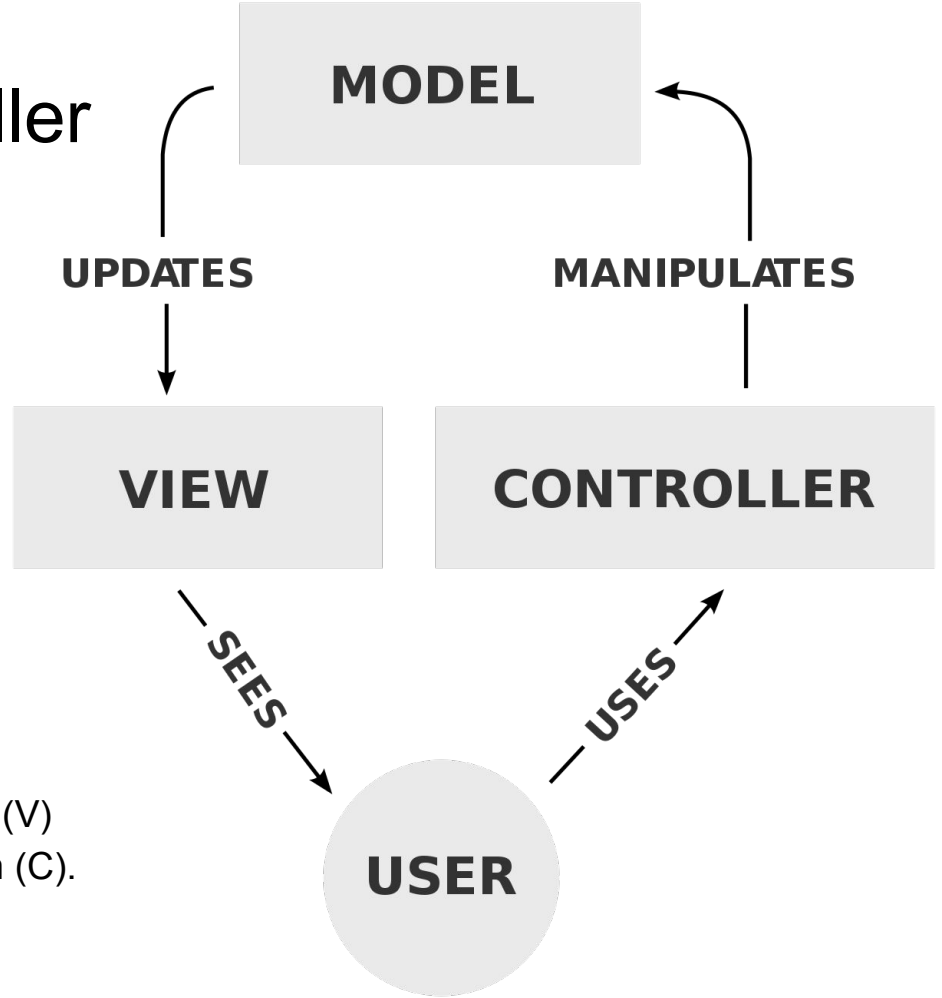
Nytt inför nästa uppgift:

MVC - Model View Controller

- Ett design pattern som är väldigt vanligt förekommande för alla typer av program som innehåller någon form av grafisk representation.

- Grunden i MVC är att vi separerar Koden för data-modellen (M) från användargränssnittet (V och C).
  Detta är den viktigaste uppdelningen.

- Inom användargränssnittet skiljer vi koden som visar upp modellen för användaren (V) från koden som hanterar input från användaren (C).



Model

- Modellen (Model) är en representation av den domän som programmet arbetar över – helt oberoende av användargränssnitt.
  - Data, tillstånd, logik.
  - Modellen i singular – vi har inte flera modeller för samma domän.
    Modellen kan dock internt bestå av flera olika delar som representerar olika aspekter av domänen (dvs ingen duplicering).

- Tumregel för avgränsning från V och C:
  - Ska kunna presenteras för och kontrolleras av användaren på olika sätt – ex. med 2D-grafik eller text – utan att valet i sig påverkar modellens tillstånd.
  - Ska innehålla allt det som vore detsamma oavsett hur modellen presenteras eller kontrolleras. Ingenting ska behöva dupliceras mellan olika vy- eller kontroll-komponenter.

View

- En vy (View) beskriver modellen för användaren.
  - Olika format: Text, grafik (2D, 3D), ljud, etc…
  - Vi pratar om ”vy”, oavsett vilket format presentationen görs på.
- Vi kan ha många olika vyer (ofta samtidigt) för en och samma modell.
  - Olika vyer kan visa upp olika aspekter av modellen.
    - Ex: karta, inventory, synfält för ett dataspel.
  - Olika vyer kan visa upp samma aspekt på olika format.
    - Ex: musik spelas upp eller visas grafiskt som frekvens-amplitud-diagram.
- Vi vill (oftast) att vyn uppdateras när modellen den presenterar uppdateras.

Controller

- En Controller styr modell och vy(er) utifrån input från användaren.
  - Vi kan ha många controllers som styr olika aspekter av både modell och vy.
  - Olika varianter av MVC använder controllers lite olika:
    - En enskild controller som styr alla aspekter av modell och vyer.
    - Eller många separata controllers för varje del-vy.

- De flesta moderna grafik-gränssnitt innehåller stöd för att förenkla för controllers att hantera användar-inputs genom events och event listeners.

Tumregel

- Smart, Dumb, Thin
  - SMART model
    - All logik ska ligga i modellen – därav SMART.
  - DUMB view
    - En view ska inte göra egna beräkningar, bara ”visa” utifrån direktiv – därav DUMB.
  - THIN controller
    - En controller ska enbart hantera input från användare.
      Den ska bara vara ett tunt lager mellan användare och program – därav THIN.

Hur bör M, V och C (och A) hänga ihop?

- Vilka beroenden bör vi ha?
  - M får inte bero av V eller C. Punkt.
    - Världen är vad den är, oavsett hur den presenteras eller styrs.
  - Får V bero av M? Får V bero av C? Får C bero av M och/eller V?
    - Det beror på... Olika varianter kan vara relevanta i olika sammanhang.
      Det finns ingen som har hittat ”den enda lösningen”.
    - Tumregel: Gör det som leder till minst antal kopplingar mellan M, V och C och flest kopplingar inuti M, V och C respektive. (*Hög cohesion och låg coupling)*
  - A väljer vilka delar av M, V och C som ska utgöra programmet.
    - A beror nästan alltid på samtliga delar.
    - Om inte – har du verkligen rätt uppdelning av M, V och C?
    - A representerar ett toppnivå-program – inget ska någonsin bero på A!

Dependency Inversion Principle (DIP)

“Depend on abstraction, not on concrete implementations”

1. High-level modules should not depend on low-level modules. Both should depend on abstractions
2. Abstractions should not depend on details. Details should depend on abstractions.