OMD

Vad varje hjälper med och eventuella implementationstips

- Strategy
    - .
    - H: OCP, DIP, D: SRP, IS
- Command
    - .
    - H: DIP, OCP D: SRP
- Composite
    - .
    - H: SRP,

Tror jag ska skriva returvärdet så länge det inte är en voi
Frågade honom:
Kommer aldrig att behöva strukturera swing.
Och småfel i koden spelar inte speciellt stor roll så längë som han fattar vad jag mennar

todo:

- **Gör seminarieuppgift 2.**
- **Skriv en generell lista på allt som kan potentiellt vara dåligt med kod**
- **Hur man skriver generella klasser nu igen**
- **Skriv en generell lista av allt man kan lösa med**
    - **Patterns**
    - **Lambda (kombinera patterns med lambda)**
- **Ha bra koll på de två projektens struktur inför tentan**

Allmänna tips:

- **Börja alltid med att önsketänka**
- **Lös så allmänt som möjligt**
    - **Gör inte en 2DpointFactory och 3DPF. Gör en NDpointFactory**
        - **Sedan kan NDpointFactory vara en abstract klass**
            - **De andra pF kan sedan ärva av NFPF.**
                - **Delar allt**
- **Tänk alltid vad som sker om jag vill implementera er kod**
    - **Måste jag ändra massor**
        - **DVS är det OCP**
- **Är lösningen lätt att användas fel?**
    - **DVS kan användaren lätt fylla in exempelvist fel inparametrar**
- **Om två funktioner som är väldigt lika i två klasser som ärver från samma klass borde jag sätta den funktionen i den abstrakta klassen och sedan enbart implementera den lilla skillnaden i en helt ny funktion, som ligger i subklasserna**
    - **Annars är det inte template method.**
- **Om man vill använda sig av den abstrakta klassens konstruktor:**
    - **Använd super()**
- **Lär dig små sakerna, som final, protected, polymorphism osv från PFK**

Föreläsning 2

Varför deklareras typen?

- Typen deklareras för att beskriva vad man kan göra med den
- Inte för att säga vad det är för typ

Storleken på en typ

- Turtle är större än en acroTurtle
    - Varför
- Double är större än int

Turtle t = new acroTurtle()

- = subtyps polymorpish

Deklarera typer så allmänt som möjligt (gärna med interface)

- Om man vill göra List saker och inte ArrayList saker
    - Deklarera då typen som ArrayList
    - Annars kan det i onödan bli onödigt kopplat till ArrayList

UML (Unified Modeling Language)

Läs detta istället:

https://fileadmin.cs.lth.se/cs/Education/EDA061/reading/umlsyntax.pdf

Tips då jag bygger ett UML diagram:

- Är ett = arv eller implementation
    - Implementation eller abstrakt klass om det inte är ett konkret klass
        - T.ex en sats. Man kan inte BARA vara en sats.
            - Måste vara implementerat på något sätt: typ en if-sats
        - Arv om det man ärver från är lite mer konkret
- Föredra kompostion över arv dock
- Består av eller innehåller kan översättas till association eller aggregering
    - Aggregering om saken försvinner om man tar bort den andra saken
- Om antalet är antingen a eller b så kan jag skriva * som default okänt antal

Note: En klass kan båda ha en association till att annat ruta samt implementera den rutan

- Rutan är ett interface i detta fall

Glöm inte att lägga till lämpliga metoder

- Exempelvist execute (when in doubt)

- Standard för att visualisera olika aspektera av ett susyem
- Olika slags diagram
    - Funktionell modell: Use cases
    - Statisk modell: Klassdiagram
    - Dynamisk modell:

Tre fack:

1. Namn (med <Interface> över om det är ett interface)
2. Attribut
    a. - (private) +(public) #(unimplemented?)
3. Metoder (ofta visas enbart enbart metodsignaturerena (och inte parameternamnen)
    a. DVS man visar parameternamnen nedan

Static metoder understryks, abstrakta kursiveras

- parametriserade klasser har typen uppe i hörnet av klassen

Ärvda metoder och attribut skriv inte ut, om de inte överskuggas (de skrivs ut)

- Kolla upp hur man gör med arv (vilka typer av pilar)

Associationer:

- Relationer har riktning: pil
- Streck är enbart relation
- På linjen kan man skriva 1 och *
    - En person kan jobba på ett företag
    - men ett företag kan ha många anställda
    - Pilar kan även få namn

Aggregation:

- Är ett slags association
    - Där något utgör en del av av något annar, men där delarna “har ett eget liv”
    - Pilar har en romb och en pil på andra sidan
    - En student kan vara pekad på av en aggregationspill av en studentunion.

Komposition

- En ifylld romb med ett sträck
- Från keyboard till key
    - En kombposition från A till B är ett slags starkare aggregering,
    - där B ägs av A och B försvinner om A försvinner

Beroenden

- Sträckad linje
- Från A klass till B klass om A klass inte har något attribut av B, men får ett B-objekt som implementerar eller returvärde i någon metod

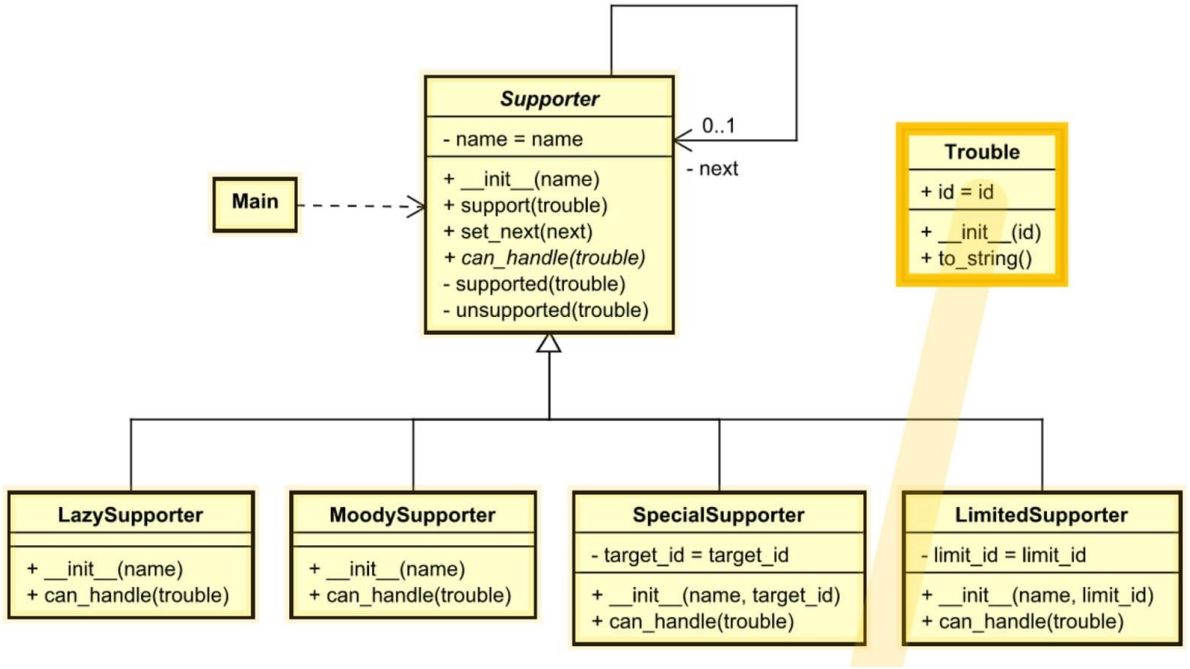
(kolla slidesen för pilarna)

Principer:

- För att få high cohesion skall ett paket eller en klass eller en metod syssla med saker som…

Ett block:

- Namn
-



SOLID-principerna (Martin …)
Fem principer:
Tänk hur du skulle lägga till kod och hur lätt det är (eller svårt)

- **Single responsibility principle**
  - Hög sammanhållning ges om en sak sysslar med en sak
- Open closed principle
  - Objects should be open for expansion, but closed for modification
- Liskov substitution principle
  - objects of a superclass should be replaceable with objects of its subclasses without breaking the application
- Interface segregation
  - *clients should not be forced to implement interfaces they don't use*
- **Dependency inversion**
  - high-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.
    - Generellt ska koden inte bero på något hårdkodat

Template pattern

- Man gör ett ytterligare lager som innehåller allt som är gemensamt mellan två klasser, istället för att subklasserna ska ha mycket likt för varandra
  - Detta utförs genom en abstrakt klass som innehåller det gemensamma
  - execute kommer då att läggas i den abstrakta klasssen
- Note den metod som förut låg i båda underklasserna ska nu ligga i superklassen

Föreläsning 3

Command pattern

- Hur man kan skriva ett program som låter oss utföra saker på ett organiserat sätt:
  - DIP: Bra att införa interface för mycket generellt

        - Som innehåller en execute() (som får saker att hända)
            - Kan vara något mindre konkret så som tillexempel value() som evaluerar tal
        - Samt en undo() som ångrar det
    - SRP: Skriva klasser för var och en av de saker de vill göra, låta dom samla på sig information om dom själva
        - Klasserna ska bara hantera en väldigt specfik sak
    - På köpet av DIP och SRP får vid OCP:
        - Det blir relativt enkelt att lägga till nya kommandon

Här finns förklaringar till seminariet (bra att läsa efter du har svarat på frågorna)

Composite Pattern
Mönster där en del av ett system består av delar som var för sig har samma struktur som hela systemet
- Ex ett ritprogram i ett ritprogram

Factories och Factory Method Pattern
En factory metod är något som skapar instanser av den klass metoden befinner sig i.
- Ex vi vill kunna skapa polära och kartesiska komplexa tal:
    - (double re, double im) och double mag, double arg) har samma signatur
        - kan inte enbart skriva var z = new Complex(0.0,1.0)
            - Kompilatorn vet ej vad vi menar
    - Därmed kan vi skapa två factory metoder: cartesian(d,d) och polar(d,d)
        - Där man i polar gör om talen till cartesian och skapar en cartesian instans

Kan även introducera ett interface för factories som producerar någon typ av värden för någon väldigt generell typ
- *Och kan sedan använda olika implementationer av detta factory-interface för att få den typ av värden när vi kör programmet*
- *Ex Point interface som skapar*

Lite oklart

Föreläsning 4
Fluent interfaces
Ska vara lätt att förstå vad som händer, även om komplexiteten göms.
- Nästlade funktioner , där man ser vilka funktioner som kallas, steg för steg

Idén att skriva metoder som går att kedja ihop
(args.findIntegers().mapToInt().filterPrimes().sum().println()) kallas för **Method chaining**

Det interface som gär Method Chaining möjligt kallas fö ett **Fluent interface**

Build pattern
När man instansierar klasser kan det vara enkelt att blanda ihop ordningen på in-parametrarna till konstruktorn. Detta är speciellt farligt om konstruktorn inte ens märker att du har blandat ihop ordningen på grund av man exempelvist har samma typ på parametrarna.

Utgå från att den klass man ska instansiera är en person klass. Ett sätt att åtgärda problemet är att introducera en klass vars enda uppgift var att skapa Person-objekt (bra SRP med!) och om en sådan klass hade ett fluent interface skulle man kunna skriva:

```
var person = ...
   .firstName("Albert")
   .phone("(609)299-792-458")
   .lastName("Einstein")
   .email("abbe@ias.edu")
   ...;
```

Exempelvist så tar .lastName in en sträng och lagrar strängen som efternamnet i personen

- Objektet som bygger personen kallas för en **builder**
- och detta mönster kallas för ett **build pattern**

Problem som ska lösas:

- Måste hitta var Buildern ska ligga
    - Lättast är att lägga det som en publik intre klass i Person
        - Kommer vara static eftersom man enbart behöver en “instantion”
- Man vill att metoderna ska returnera en referens till buildern själv för att kunna chain:a
        - Så ändra buildern och returnera buildern
- Måste uppdatera tillståndet i personen
    - build() metod i buillern
    - För att man inte ska behöva sätta in builderns argument i rätt ordning till person så byts Persons konstruktor till att ta in en builder.
        - Sedan uppdateras Personerns argument för att passa med buildern
- Vill förhindra att användaren ska kunna anropa den röriga “vanliga” konstruktorn i Person
    - private förklarar konstruktorn
      Skapa sedan en static method i Person som returnerar en builder
        - Private förklara builderns konstruktorn

```
var p = Person.builder()
        .firstName("Albert")
        .phone("(609)299-792-458")
        .lastName("Einstein")
        .email("abbe@ias.edu")
        .build();
```

Strategy pattern

Om man använder sig av ett template mönster, och skapar:

- En abstrakt klass “Account”
- Två klasser:
    - FixedFeeAccount
    - RelativeFeeAccount

Hur gör man då för att byta sortens konto man har?

- Man kan ju typ konvertera men det ändrar inte själva dynamiken
    - Dvs de metoder som tillkallas

Grundproblemet är att man har hårdkodad beräkningen av avgiften

- En mer flexibel lösning hade varit att delegera avgiftsberäknignen till någon annan, till exempel ett “withdrawal policy objekt” t.ex FixedFeePolicy och i account ha:
    - UsePolicy(policy) metod
        - ändrar this.withdrawPolicy
    - Och om man vill byta account bara byter man via UsePolicy()

detta mönster, att delegera till ett annat objekt att göra beräkningen, kallas för Strategy Pattern

- Note: Ett av kursen viktigaste mönster

Det som karaktäriserar just ett Strategy Pattern är att Account har har en referens till ett objekt som hanterar någon algoritm (i detta fall finns en referens från Account till WithdrawalPolicy implementationer som hanterar fee beräkningen)

- Denna referens ska lätt även kunnas byta ut mot ett objekt som implementerar samma interface (dvs WithdrawalPolicy)

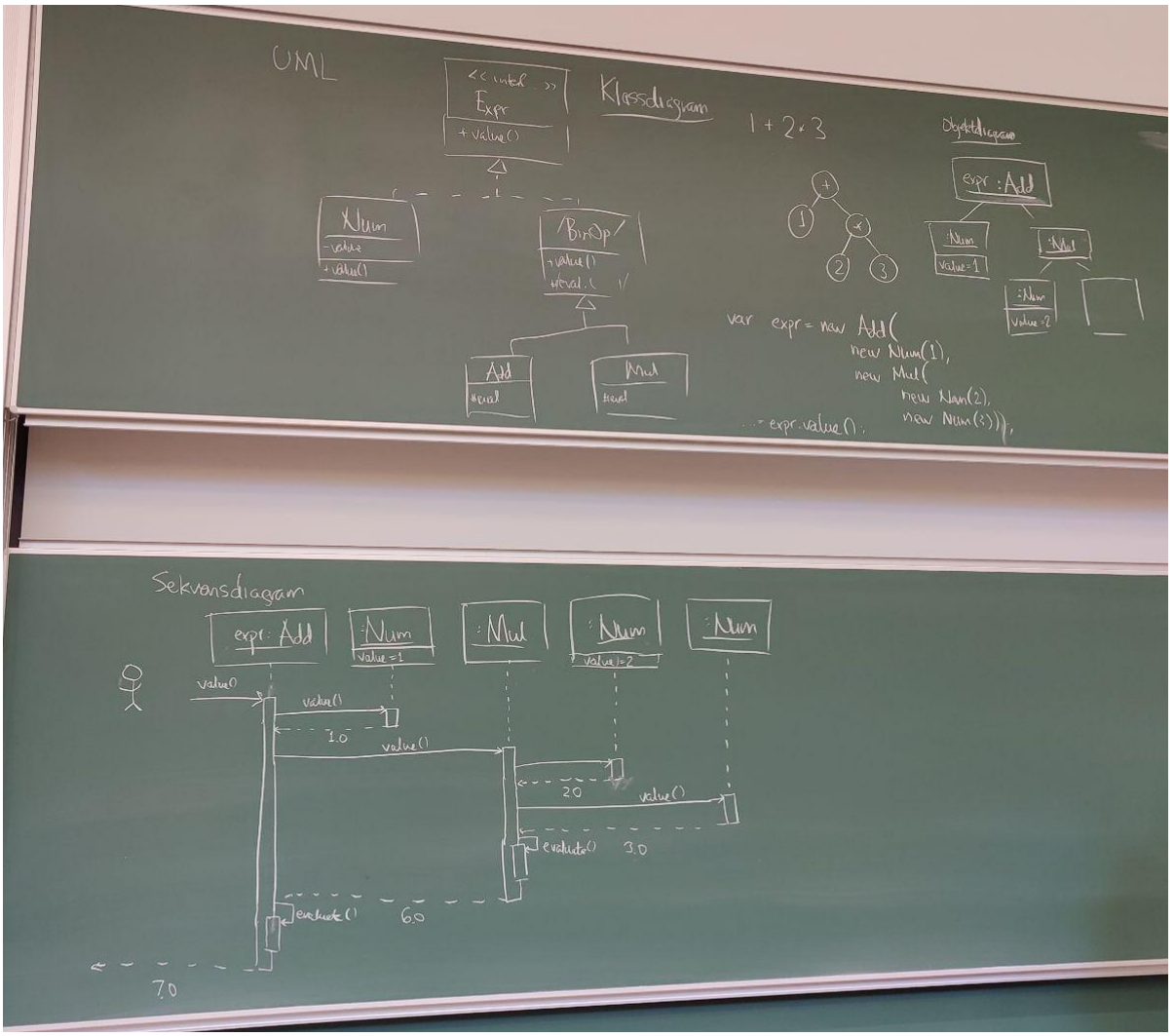
Hur Strategy Patten förhåller sig till andra patterns

- Command Pattern är egentligen ett slags specialfall av Strategy Pattern. Vi har i Command Pattern ett interface, och ett antal olika implementationer av interfacet som gör olika saker, och vi kan i princip byta ut olika kommandon mot varandra. I praktiken får vi dock oftast en ström av kommandon att utföra när vi använder Command Pattern, vi brukar inte ha en referens som vi kan byta ut som i konto-exemplet.

- När vi använder Template Method Pattern kan det ytligt se ut som att vi i vår template-metod delegerar till någon annan att göra en beräkning, men som vi såg ovan är det i slutänden this vi anropar, så Strategy och Template Method är strukturellt helt olika.

Föreläsning 5

BILD FRÅN 13/sep



Klassdiagram; UML kartan av klasserna
Objektdiagram: De objekt som finns när vi kör och hur de ser ut i minnet

För BinOp då ett träd av 1+2*3 evalueras:

- var expr = new Add(new Num(1), new Mul(new Num(2), new Num(3)

NOTE:

- Skriver ut värden med?
- Om de har variabelnamn, exempelvist Add har variabelnamn expr skrivs det som expr:Add
- Om det inte finns ett variabelnamn skrivs det som :typ
    - DVS generelllt variabel : typ

Sekvensdiagram:
Note:

- BinOp saknas i och med att de inte är objekt och skapas aldrig
- Lilla gubben är användaren
- Staplarna är hur länge det objektet används
- Pilen frammåt är vilken funktion som anroppas på objektet (som pekas på)
- Pilen tillbaka är då ett returvärde ges

- Tiden går alltså neråt
- Om man vill göra det riktigt tydligt kan man rita en stapel lite åt sidan, för att visa att en annan metod i samma objekt tillkalllas (Note det värdet från ex. eval() skickas tillbaka tilll value() som tillkalla den och det värde returneras till ett annat objekt
    - Så båda staplarna ska fortsätta
- För en void så dras ingen pil tillbaka

Event sourcing
Kommentar från förra föreläsningen angående Account:

- vill inte enbart spara värdet av antalet pengar man har i banken
    - Problematiskt om flera ändrar värdet exakt samtidigt
    - Vet inte historiken
- Vill ha en lista av alla transaktioner
    - Kan då bara lägga till transaktionerna för varje person
        - Då kan man pusha till den listan exakt samtidigt och det gör inte skillnad vilken ordning transaktionerna läggs till
    - Historiken är känd

- När man vill veta hur mycket pengar man har räknas bara summan av historiken

Decorator (formerly known as wrapper) pattern
Om man har en klass som man absolut inte vill ändra i, men man fortfarande vill förändra dess funktion

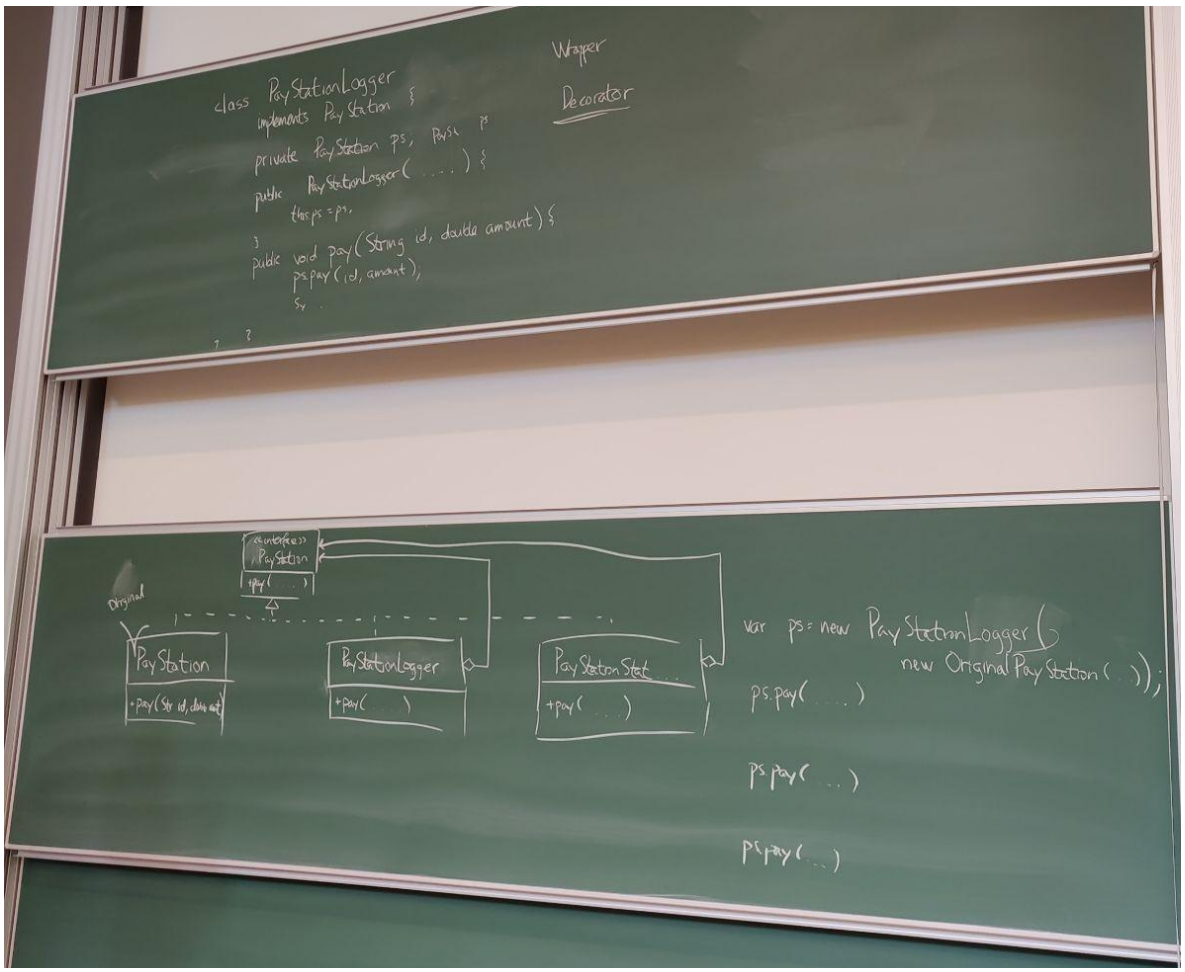
Skriv ett interface ovanför den klassen (A)

- Byt namn på klassen
- Skriv en till klass (B) under interfacet
- Ha en aggression från B till interfacet.

kan nu skriva new B(new A());

Varför gör man en aggression från B till interfacet och inte A?

- Om man vill ha flera decorators så kan man kedja ihop och välja vilka decorators man vill ha utan någon extra komplexitet. Bar kedjar new C(new B(new A)))
    - Så C.pay() anroppar B.pay() som anroppar A.pay()



Exempelvist med PayStation interface, OriginalPayStation (A), PayStationLogger(B)

- ha pay i interfacet
- pay i PayStation är att betala
- Om man vill decorera Paystation med att även logga:
    - I PayStationLogger så är pay() = {ps.pay(), log() … }
- Har då lagt till en feature till OriginalPayStations pay() utan att ändra OriginalPaystation

NOTE: Att förenkla detta.

- Använd template method för att abstrahera en decorator. Så lägger till allt som är gemensamt för en decorator. Väldigt bra att ha, om du potentiellt ska ha flera decorators.

Förhåller sig till andra patterns

- en form av strategy
- en form av command pattern

Lambda uttryck

Funktionellt interface (SAM-interface): Ett interface med enbart en metod

- kompilatorn förstår att det är den som ska kallas utan att man behöver säga det konkret

Ex: new intergrate(x->x², 0, 1)
interface RealValueFunction{

```
        double valueAt(double x)
}

intergrate(RealValueFunction f, double min, double max){
. . . räkna integral
    - Kan använda f.valueAt()
}
```

Null

Optional<typ>

- En sorts wrapper kring ett värde som hindrar det från att vara null
- Används om en variabel har risk att bli null

OBS undvik get()

har tre viktiga metoder på en optional

- map(...)
    - Används för att hämta det som finns i optional objektet
    - Om det inte finns ett värde inkapslas så hände inget och raden ignoreras
    - ex kan göra findAuthorByISBN().flatMap(author -> author().adress).map(adress -> author.zipCode()).orElse(“”) för att få en author från ISBN och returna det. Om det inte finns så returneras “”.
- flatMap(...)
    - Returnerar en optional som potentiellt innehåller värdet
- orElse(...)
    - Att raden ignoreras kan behandlas i orElse() funktionen.
        - Kan välja vilket värde man vill ha

```
  public T orElse(T defaultValue) {
      return
          isPresent()
          ? value
          : defaultValue;
  }

public <U> Optional<U> map(Function<T, U> f) { //f är en lambda funktion
      return
          isPresent()
          ? of(f.valueAt(value))
          : empty();
  }

   public <U> Optional<U> flatMap(Function<T, Optional<U>> f) { //f är en lambda funktion
      return
          isPresent()
          ? f.valueAt(value)
          : empty();
  }
```

Skillnaden mellan dom är att map har en of() kring värdet som f evaluerar

- DVS

Värdet null:
- Flatmap empty
- Map empty

Värdet inte null
1. f ger null
    - Flatmap ger null?
    - Map ger en optional som omkapslar null?
    - .
2. f ger inte null
    - Flatmap ger värdet från f i en optional
    - Map ger värdet från f i en optional

Exempel:
Optional<Book> findByISBN(String isbn){
    return Optional<Book>.of(värde(isbn))
}
- Skriver den som Optional<Book> i och medan att varje sträng inte nödvändigtvist har en bok kopplat till den
    - Så boken kan bli null
        - Den inkapslas därför av en Optional

Mutability (och immutability)
Mutability är om en varaibels innehåll kan ändras

```
var rut = new Person("Rut");
displayInfo(rut);

var adam = new Person("Adam");
var liv = new Person("Liv");
rut.addFriend(adam);
rut.addFriend(liv);
displayInfo(rut);

var name = rut.name();
name += "ger";
// name.append("ger");
System.out.println(name);
displayInfo(rut);
```

Exempelvist här har det blivit fel. Man har tagit Rut + ger men det kommer enbart få namnet Rut. String klassen är immutable så += kommer skapa en ny sträng som name kommer att referera till (med innehåll Rutger) men i och med att de är immutable så kommer rut name objekt inte att ändras.

En sak man ska vara försiktig med är mutability. Om friends är private men man ger en referens till friends listan via getFriend(), då kan friends att ändras via metoden till en nya referens, vilket kanske inte var tanken

- Kan vara bra att använda mestadels omutbara.

Observer/observable (och PubSub)
Ett mönster som kan användas för att minska "coupling i ett system där ett eller flera objekt är intresserade att veta vad som händer med ett annat objekt

Mönstret kräver:

1. Har ett interface, observer, för objekt som är nyfikna på något annat objekt
    a. Interfacet har enbart en metod, update, som anropas så snart som det objekt vi observerar ändras
        i. **Note, det kan alltså implementeras med en lambda funktion. Kul!**
2. Har en klass observable, som har en lista med alla sina observatörer
    a. Så snart den ändras anropas update metoden i observatörerna

Exempel på strukturen för observer:
Man har en Account klass. Extendar observable.

- Den måste alltså ha en void changed() metod

Säg sedan att något ska tracka ett eller flera accounts.

- Skapar då en Account Tracker som implementerar observer.
    - Den måsrte då ha en update(observvable obs, Object obj)
        - Som updaterar tracker

PubSub (note: Kursen användas obserer pattern
Publish-subscribe

- I observer mönstret måste de som observeras hålla koll på vilka som är nyfikna på dem
    - Men de måste iallafall inte hålla koll på några detaljer

I ett pubsub mänstet frikopplar vi helt den observeras från den som observerar och vice versa

- Kopplingen hanteras där av ett mellanliggande objekt
    - Det objektet kallas för **en event bus** eller **message broker**

Fördelar med detta

1. Ännu lite mindre coupling än sobserable
2. Synkroniseringen mellan obserable och observer hanteras av ett seperat objekt

Nackdelar:

1. Mer komplicerat än observer pattern (**För denna kursen är det enbart viktigt att förstå principen att minska coupling mellan obserable och observer genom att införa ett generisk gränssnit för kommunikation mellan dom, så kursen kommer använda observer pattern)**

Model View Control - MVC
Bra sätt att få SRP med ett GUI

Model: Ansvarar för vår modell av problemområdet
- Denna modell bör vara oberoende av hur vi vill presentera den
- Massor logik av hur exempelvist beräkningar görs.

View: Har ansvar för hur vi presenterar vår modell
- Vi kan ha flera olika views till varje modell

Control: Har ansvar för att hantera användarinteraktion och se till att model uppdateras
- Den kan även användas för att synkronisera Model och view (se Observable/observer ovan)

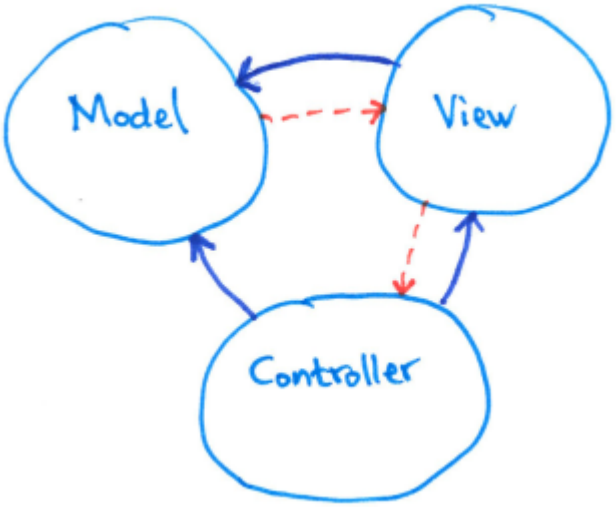


Bild på vem som ser vad (blå) och vem som kan prata med vem (röd)

NOTE för den röda kan observer pattern anvöändas

Note: Många använder namnet MVC även om besläktade mönster

Microservices
Ett väldigt drastiskt sätt man kan bryta upp M från VC:
- Görs genom att låta modellen köra som en service på någon dator
    - Och sedan ha frontenden som pratar med vår service

Tekniken att dela upp programmet i en eller flera tjänmster som kommunicerar med ett GUi, och kanske även andra program kallas microservices
- **Han vill bara att vi ska vara veta att dennna teknik är väldigt vanlig idag**

Stream
Ett sätt att slippa loopa och skapa (muterbara) listor

Stream är ett verktyg/mönster som hjälper oss om vi exempelvist har ett stort antal forloopar, som långsamt transformerar en lista

Tre faser:

1. Skapa strömmen på något sätt
    a. Arrays.stream(args)
    b. list.stream()
    c. str.chars()
2. Bearbeta strömmen
    a. Filter
    b. Map
    c. MapToInt
    d. MapToObj
    e. Parallell() (coolt sätt att använda flera kärnor)
3. Sammanställ resultatet (efter detta blir strömmen en lista)
    a. Sum()
    b. Min()
    c. allMatch()
    d. reduce()

Stream kan göra koden mer **delklarativ**

- Dvs den säger mer vad den gör istället för hur den gör det

Seminarium 3

Uppg 1.

Software ska vara beroende av hardware. I och med att software är en lägre modul. Men nu beror även hardware av software. Det bryter mot principen D i solid.

Man kan lyfta upp Operand till hardware (den lätta lösningen)

Men en bättre lösning är potentiellt att använda en wrapper (decorator som ärver Operand. En operandWord som ärver Operand och kapslar om Word. SÅ den behöver enbart att veta att Word existerar och delegera sina beräkningar till word. Men word behöver inte veta att opeandWord existerar (detta ger ett enkelt riktat beroende - med association från WordOperand till Word

Uppg 2.

Bryter mot single responsibility principle. Tänk MVC. De ska vara uppdelade.

- Det är en view för att den behandlar buttons
- Den är en controller för det har en ActionListener.
- Det är en model för den behandlar logik, dvs den har i detta fall en boolean.

Vill göra ett huvudprogram (kolla lösningarna)

Bättre lösning med Flowsync:
De bra sätten att få MVC att vara uppbyggt på rätt sätt

- Flowsync (**Kolla anteckningar, de kommer att uppdateras)**
- Obs sync

1. Skapa tre klasser, som behandlar M, V och C.

Boilerplate. Hur bra behöver man veta det.

M: boolean on, toggle(), isOn()
V: Extends JButton. Tillgång till en model objekt. Det kan tas in via konstruktören,
- display();

C: Ser till att allting händer på rätt sätt. Har tillgång till view och model via konstruktorn.
- I switch Control. Behöver även en view.addActionListener (kan ta in en lambda funktion

Om man istället hade använd Obs sync
Då hade även m extendat en observable, medan de andra ska vara Observer
- DVS att skapa obs sync är att synca med observer pattern
- Då måste även model att veta att det finns en observer. Kan göra model.addObserver(...) i c.

Kan låta V vara kvar som den är

Controller ändras som i ovan.
Kolla även lösningarna i anteckningarna

Man får välja vilken sync man tycker är enklast.

Uppg 3.

a. List.of(1,2,3,4,5).stream().filter(i -> i%2 == 1).mapToInt(i -> i*i).sum()
    - Note, måste ha mapToInt för att sum ska veta att det är massor av integers.
    - kan även använda map och reduce(0, (acc,k) -> acc + k) istället för su,m

b. Tråkig
c. .
d. Factorial. Använd reduce. Man börjar på talet man tar lägger in. Man tar det som a, och det första talet i listan som b. Sedan blir den uträkningen nästa a, medan b blir nästa tal i listan. Osv

Uppg 4.
Se mina lösningar i anteckningar

Uppg 5.
Problemet med Random är att allting är hårdkodat. O:et i solid säger att man inte ska behöva gå in i den gamla koden. Man vill alltid kunna skriva rng.next() och sedan bör det finnas underklasser till Random och man kan skicka in Random som man vill. DVS det är svårt att få OCP Random.

Man vill få statistik på det, utan att pilla på simulation. Detta kan göras med en decorator.
- Simple standard implementering.

Föreläsning 7
Note: Den abstrakta klassen skapas inte som ett objekt. Det skapas instanser av dess underobjekt. Så det finns enbart en instans

Note: För sekvensdiagram, se till att skilja på olika instanser av ord. För varje instans, skriv en ny kolumn.

Facade:
new DrawSquare(10,10,20).execute(drawing); //svårläst

//någonstans står det var d = new DrawingFacade(drawing);
d.square(10,10,20); //bättre

skapar en facade för att gömma ett bökigt interface.

Skapar en Facade klass som innehåller square som gör new DrawSquare(10,10,20).execute(drawing);
Det är en sorts inkapsling bakom en snällare facade.

Adapter
Vi har en A interface (har en op())
Vi har en B klass. (har en operation())

Har även en adapter som kan ta ett B objekt och göra om det till ett A objekt

Mål använda ett inkapslat B objekt för att kunna göra metoder som finns för A objekt

Object adapter: Kräver att BToAAdapter implementerar A för att det ska kunna göra dess metoder.
- sedan definierar man alla A metoder utifrån B
    - dvs op(){ b.operation }
        - DÄr b ges i konstrukorn

Class adapter: Kräver att AToBAdapter implementerar A för att det ska kunna göra dess metoder samt att det extendar B
- Sedan i op() i AToBAdapter kan man göra this.operation();
- Denna adapter är inte lika föredraglig att använda eftersom man även får med alla andra metoder (eftersom den extendar b)

Singleton
Om jag bara vill ha en instans hela tiden av en klass.
En rimlig, men lite ovanlig, tolkning av SIngelton är att man bara skall returnera instansen en enda gång

Implementering: göra en static final objekt INSTANCE;

samt en public static A getInstance()
som innehåller
```
if(INSTANCE == null){
        INSTANCE = new objekt();
} else {
        throw new RuntimeException(“oh no”)
}
```

Ett vanligt simgelton-objekt introducerar globalt tillstånd
Metoder som använder globalt tillstånd döljer sina beroenden, signaturen på metoderna beskriver bara en del av deras kontakt med omvärlden
Metoder som använder globalt tilllstånd är svåra att testa

- Betydligt bättre att slcoal om parameter som byts till mock objekt

Båda ett pattern och ett anti pattern (undvik om du kan)

Liskov Substition Principle
Subtypes must be substitutionable for their base types

- Får inte kräva mer av subklasser än man kräver av superklassen

Ex på något som bryter.
Om man har IntPair extends Object
och har en equals(Object)
Då är det fel att i equals(Object) att klassbyta från objekt till IntPair. Kräver mer av underklassen än överklassen
Lösningen på detta är att kolla om Object är en instans av IntPair och då klasscasta (klassomvandla) till IntPair

Note: Kan finnas saker som inte syns. T.ex att Object har en hashcode metod. Den kräver inget. Men underklassen har inte en sådan metod, som gör att de kräver något extra

UML tillståndsdiagram:
Visa tillståndsdiagrammet för en klocka med två knappar, en + knapp och en mode knapp som gör det möjligt att ställa om först timmar och sedan minut
ett tredje tryck mode-knappen gör att klockan går tillbaka till att visa tiden

Pricken är där man börjar. (rita input och output (dvs ex hours++)
Ritar varje olika tillstånd (olika sätt klockan kan se ut)
Rita vad som händer vad som händer när man klickar på olika knappar

Begrepp som vi ha sett iallafall (kommer ej på tenta)

- Metrik för stabilitet
- Regler för när man ska ha saker i samma paket
    - Single responsibility
    - Dependency inversion
        - Båda gäller också för paket

Principer från effective java (inte på tenta, men bra att kunna)

1. Föredra använda static factory, istället för constructor
2. COnsider a builder when faced with many constructor parameters
3. Prefer dependency injections into hardwiring resources
    - DVS föredra att en metod tar in parametrar. Hårdkoda inte allt
4. Försök göra allt så lite åtkomligt som möjligt. Protected och private är bra att ha.
5. Minimize mutability
6. favor composition over inheritance
7. Prefer lists to arrays

8. Prefer interfaces over abstract classes
9. Consistently use override annotation
10. Return empty collections, not null (eller generellt returnera null)
11. Minimize the scope of variables
12. Prefer streams and/or for-each to traditional for
13. Avoid string
14. Refer to object by their interfaces (va beroende av så abstrahera typer som möjligt)
15. Use exceptions only for exceptional conditions