

Øvingsforelesning 6

TDT4100 Objektorientert programmering

07.03.2022

Mathea Berg Vindsetmo

Vitenskapelig assistent, TDT4100

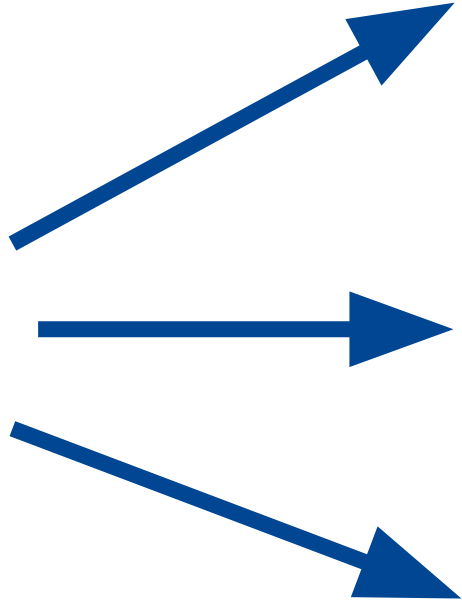
matheabv@stud.ntnu.no



Observatør-Observert og Delegering

- Standard **designmønster** (*engelsk: design patterns*) i programmering
 - Dette er **ikke** noe som er innebygd eller Java-spesifikt
 - Generelle teknikker for informasjons- og programflyt i programmer
- Fellestrekk:
 - Bruk av interface står sentralt. I **delegering** kan både delegatet og objektet som delegerer ofte være ulike klasser som implementerer samme interface. I **observatør-observert** implementerer gjerne **observatørene** samme interface.
 - Handler om samspill mellom to eller flere objekter, ofte i en **(1-n)** assosiasjon

Delegering



Delegering

- Delegering er en teknikk hvor et objekt **tilbyr metoder** for å gjøre en spesifikk handling og **bruker et internt objekt** til å utføre selve logikken, dvs. den **delegerer** oppgaven til det interne objektet
- Delegering i det virkelige liv handler om at én person kan ta ansvar for å utføre noe for en oppdragsgiver, men overlater til en annen (**delegaten**) å gjøre selve jobben uten at dette (nødvendigvis) synliggjøres for oppdragsgiveren

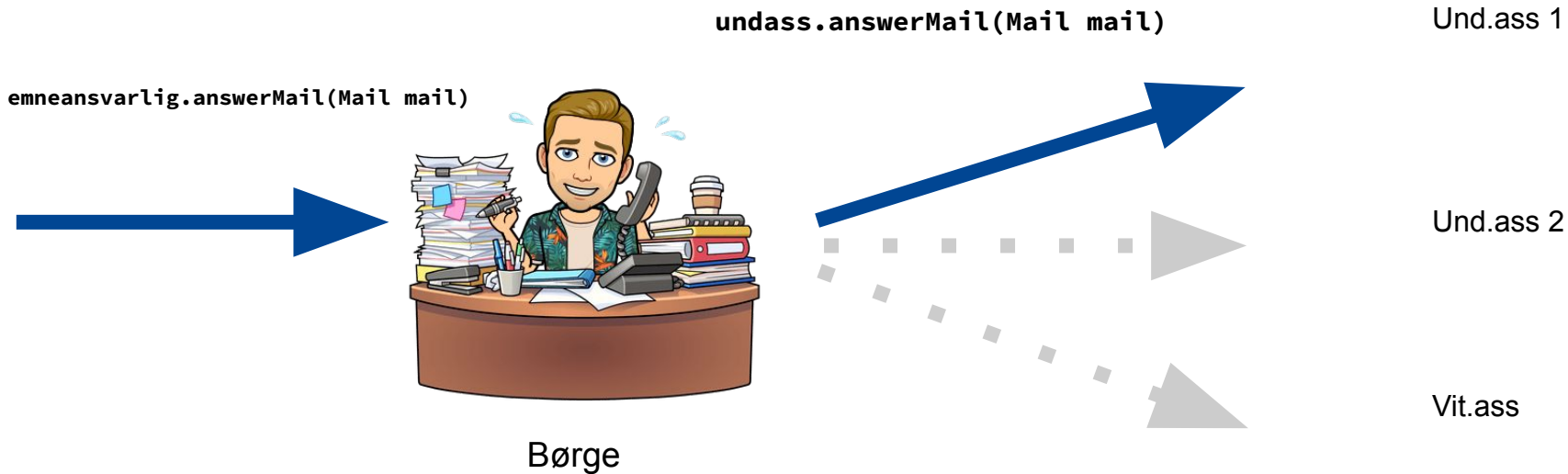
Delegering (2)

- To objekter - **Delegerende** og **delegat**
- **Delegerende** har en referanse til et/flere **delegater**
- **Delegerende** har et sett metoder hvor metoder med ca. samme navn finnes hos **delegat** (*les: grensesnitt*)
- Kall til ett av metodene hos **delegerende** fører til et kall til **delegat**
- Viktig forskjell: Implementasjonen av metodene er (ofte) **grunnleggende forskjellig** hos **delegerende**, sammenlignet med **delegat**

Eksempel 1

Fagstaben i TDT4100

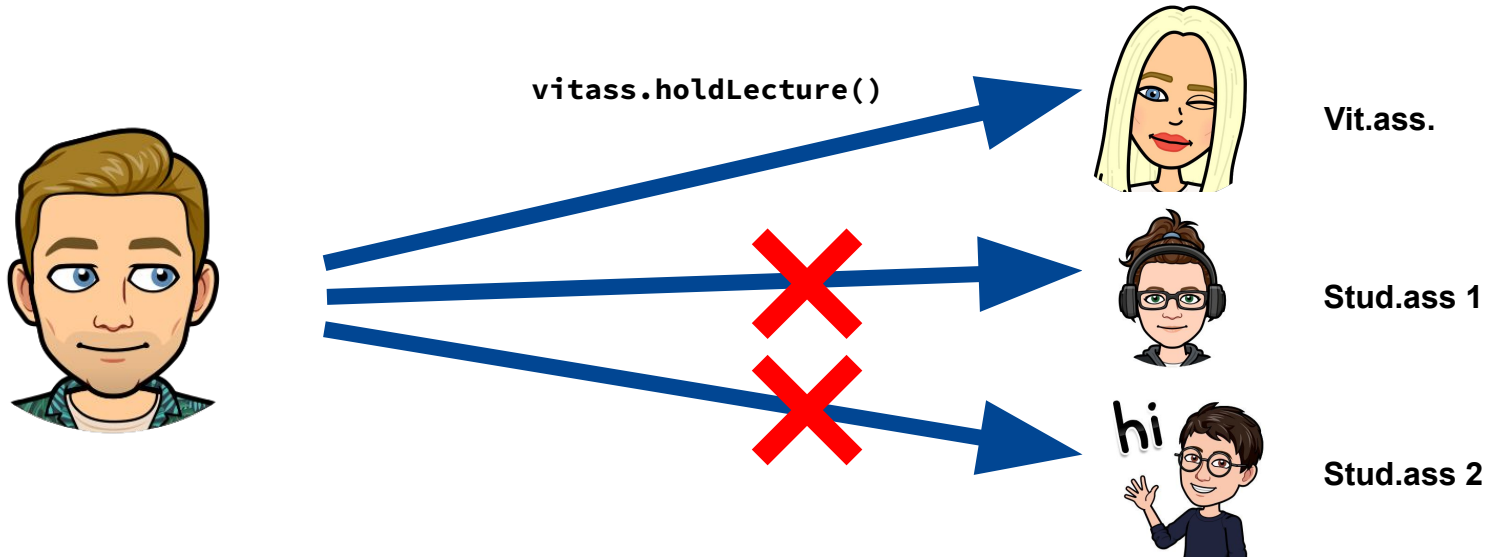
- Alle kan svare på mail, men emneansvarlig kan ikke svare på alle mailer alene, og delegerer derfor dette til und.ass. og vit.ass. (som det er flere av)



Eksempel 2

Fagstaben i TDT4100

- Emneansvarlig har nok med å holde to forelesninger i uka, og øvingsforelesning blir derfor delegert bort til vit.ass, som har dette i stillingsbeskrivelsen. Denne oppgaven kan derimot **ikke** delegeres til en stud.ass, ettersom det ikke er i deres stillingsbeskrivelse.



Praktisk oppgaveløsning

I dag skal vi hjelpe *OOP mini* med å lage en simulasjon av deres sentralbord for kundeservice og hvordan de ansatte arbeider her, som de ønsker å bruke til ressursoptimalisering



Oppgave 1:

Lag `CallRecipient`-interfacet

- Dette blir et felles grensensitt for **delegat** og **delegerende**
- Objekter som implementerer dette grensesnittet skal kunne besvare en henvendelse (**String**) ved å kalle på en **answerCall**-metode.
 - Det er ikke så viktig hva denne gjør, så lenge vi faktisk ser at den blir kalt

Oppgave 2:

Utvid Employee-klassen

- Dette blir vår **delegat**
- Denne inneholder allerede et **navn**, en **rolle**, og skal i tillegg holde orden på **antall oppgaver den har utført**, for å sørge for rettferdig delegering
- Implementer interfacet vi lagde i oppgave 1 og lag **answerCall**-metoden for **Employee**

Oppgave 3:

Lag en CallCenter-klasse

- Dette blir vårt **delegerende** objekt
- Skal inneholde en **liste over ansatte** ved sentralbordet
- Skal ha en metode for å **legge til ansatte**
 - **addEmployee(employee)**
- Klassen skal ha en metode som tar inn en rolle som parameter og velger et passende **Employee**-objekt å delegere denne oppgaven til. Metoden skal også delegere til den blant de aktuelle ansatte som har utført **færrest antall oppgaver** hittil for å gjøre det rettferdig.
 - **getEmployeeForTask(role)**

Oppgave 4:

Implementer `CallRecipient`-interfacet

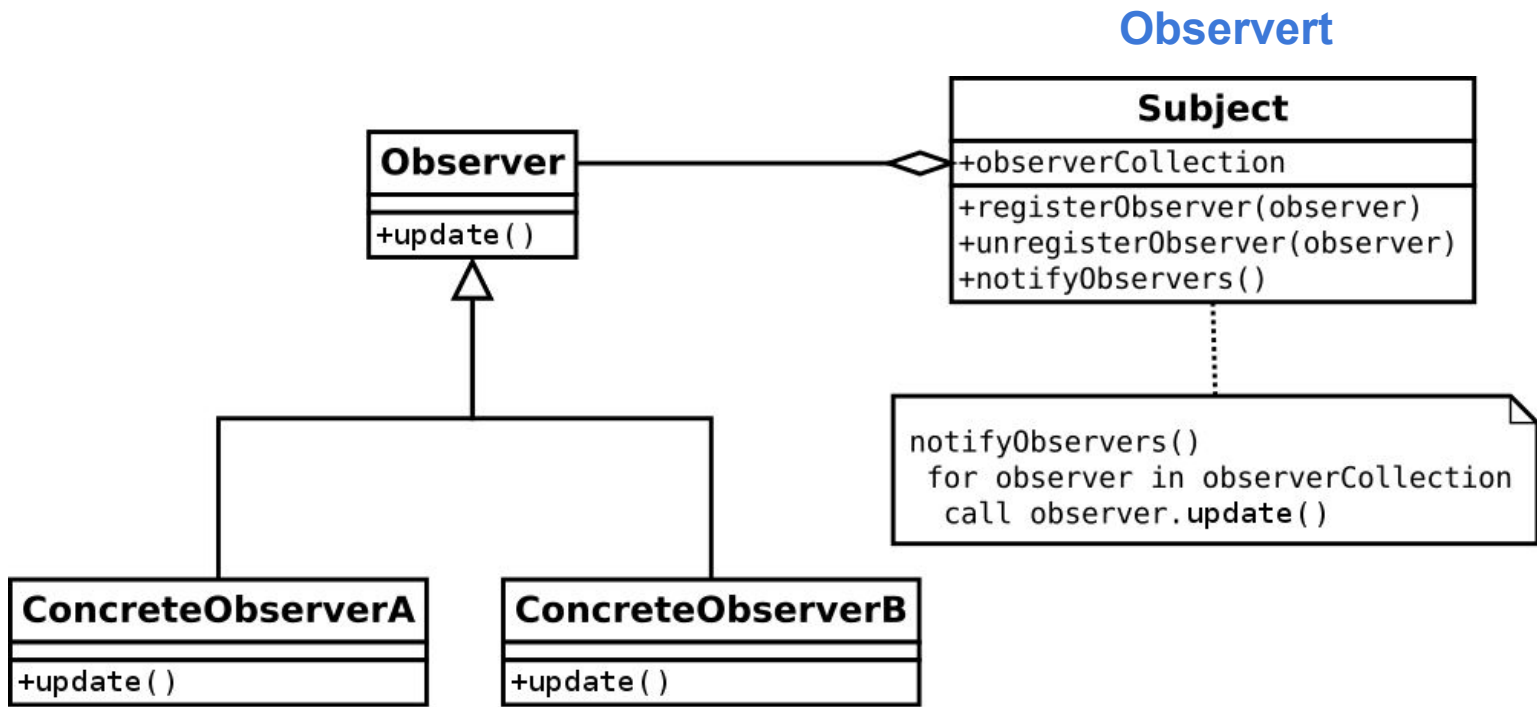
- Vi må finne ut hva `CallCenter`-klassen skal gjøre i forskjellige tilfeller, basert på hvilken henvendelse som skal besvares.
- `answerCall` skal velge en passende ansatt (basert på rolle) å delegere til, samt få en ansatt til å besvare selve henvendelsen.

Delegering

Motivasjon

- Én implementasjon kan være raskere for bestemte argumenter
- **Parallellisering** og/eller jevn fordeling av last på underliggende maskinvare
- Konfigurasjon som avhenger av formål, f.eks. drift og/eller testing – Delegerende kan tilpasse og/eller filtrere metodekallene

Observatør-observert



Observatør

Observatør-observert

- **Observatør-observert**-teknikken: Et **observert** objekt opprettholder en samling **observatører** som varsles hver gang objektets tilstand endres
 - Idé: Viderevende tilstand
- Det er ofte behov for å sikre at informasjonen et objekt holder på stemmer overens med et annet.
 - F.eks. at det som vises i et **brukergrensesnitt** stemmer overens med **tilstanden** til de underliggende objektene i systemet. En god måte å gjøre det på er å bruke observatør-observert-teknikken
- Se øving 8: **Stock**, **StockListener**, **StockIndex**-oppgaven



Observatør-observert (2)

- Består av to distinkte roller: **observatør** og **observert** som må ha følgende metoder:
- **Observert-rollen**
 - Registrere og fjerne observatører
 - Endre tilstanden sin
 - Si fra til observatørene om endring
- **Observatør-rollen**
 - Ta imot beskjed om tilstandsendringen i objektene det observerer
 - Evt. endre tilstander, gjøre spesifikke handlinger basert på beskjed

Observert: krav

1. En **intern tilstand** som kan leses og endres vha. tilgangsmetoder, f.eks. getter- og setter-metoder
2. Metoder for å la *andre* objekter **registrere seg** som **interesserte** i endringer i tilstanden
 - Samling (f.eks. en **List**) med observatører (**interesserte**)
 - Metoder for å administrere denne listen (**addListener**, **removeListener**)
3. Metoder for å **varsle** registrerte observatører hver gang det skjer en endring som observatørene ønsker å vite om
 - Denne metoden heter ofte **fireChangeEvent**, **firePropertyChanged**, etc.
 - **Merk:** disse metodene kan du kalle **hva som helst**, men et viktig aspekt ved å skrive **god kode** er å gi metodene dine **deskriptive og informative navn**

Observatør: krav

1. Implementerer et **Observatør-grensesnitt som vi lager selv**
 - Navnet på dette inneholder gjerne **navnet på objektet som skal observeres**
 - Grensesnittet definerer metoden den observerte skal kalle på for å si ifra om endringer og så skal observatørene **implementere** denne
1. Har ett eller flere objekter den **observerer** (et enkelt felt eller en liste med observerte objekter)
 - **Observatøren** må legges til som observatør i **Observert**-objektet for å kunne varsles om endringer
1. **NB!** Java har en **innebygd klasse** som heter **Observable** og et grensesnitt grensesnitt **Observer**. Disse bør **ikke** brukes.
 - Markert som **deprecated** fra Java 9
 - Det er ikke noe innebygd i Java vi skal bruke for disse teknikkene i dette emnet vi må lage grensesnittene og klassene selv.

Eksempel

Øvingsfrister

“Øving X har utvidet frist”



Blackboard



Student
implements **BlackboardListener**



Student-objektet er allerede lagt til som en observatør av Blackboard via en **addListener**-metode

Dette skjer inne i `fireDeadlineChange` hos den **observerte**:

```
for(BlackboardListener l : listeners) {  
    l.deadlineChanged(exercise, newDate);  
}
```

Dette skjer inne i `deadlineChanged` hos hver enkelt **observatør**:

```
this.exercises  
    .getExercise(exercise)  
    .setDeadline(newDate);
```

Praktisk oppgaveløsning

Vi skal lage et enkelt
Proof-of-Concept for et system
som sender push-notifikasjoner til
kunder som har lastet ned og
installert diverse medlems-apper
fra *OOP*-konsernet.



Oppgave 1:

Lag et AppListener-grensesnitt

- Dette er grensesnittet for **observatører**

Oppgave 2:

Lag App-klassen

- Skal inneholde en liste med **AppListener**-objekter
- Skal ha metoder for å legge til og fjerne objekter fra denne listen:
 - **subscribe()**
 - **unsubscribe()**

Oppgave 3:

Lag metoder for å varsle om endringer

- **App** må ha en metode som sender ut push-notifikasjoner (**String**) til alle lytterne
- **AppListener**-objekter skal ha definert en metode som skal bli kalt når en ny push-notifikasjon sendes ut via appen den lytter på.

Oppgave 4:

Få Phone til å implementere AppListener

- Lag Phone-klassen
- Push-notifikasjonen skal legges til i **Phone** sin liste med notifikasjoner.
- Må skrive ut noe til konsoll så vi vet at den mottar notifikasjonen.

Oppgave 5:

La Phone legge til og fjerne seg selv som lytter

- **Phone** skal ha en metode som kaller på **App** og lar den legge seg selv til som lytter der, samt en metode som lar den avregistrere seg som lytter.

Teoriprøven, eksempler

I hvilke(n) sammenheng(er) må du bruke **Integer** istedenfor **int**?

Velg ett eller flere alternativer

- Når du deklarerer en **List**
- Hvis verdien skal kunne være negativ
- Hvis verdien skal kunne konverteres fra en **String**
- Hvis verdien skal ha desimaltall
- Hvis verdien skal kunne være **null**

Teoriprøven, eksempler

For en klasse definert som:

```
public final class DoesSomething {  
    ...  
}
```

Hvilken synlighetsmodifikator vil det i de fleste tilfeller være hensiktsmessig å bruke på feltene i klassen?

Velg ett alternativ:

- private
- Ingen modifikator
- public
- protected

Teoriprøven, eksempler

Når bør du bruke **final** i deklarasjonen til et felt?

Velg ett alternativ:

- Når feltet er av typen **List**
- Når feltet skal kunne være **null**
- Når feltet ikke skal ha en *get-metode*
- Når feltet aldri skal endre verdi

Teoriprøven, eksempler

Hva er tilstanden til `number` i `test`-objektet etter at følgende kode er kjørt:

```
public class SimpleConstructor {  
    private int number;  
  
    public SimpleConstructor(int number) {  
        this.validateNumber();  
        this.number = number;  
    }  
  
    public void validateNumber() {  
        if (number < 0) {  
            throw new IllegalArgumentException("Can't have negative numbers");  
        }  
    }  
  
    public static void main(String[] args) {  
        SimpleConstructor test = new SimpleConstructor(-5);  
    }  
}
```

Velg ett alternativ:

- Det vil utløses et unntak
- 5
- 0
- Koden vil ikke compilere

Teoriprøven, eksempler

Du har to klasser, **Student** og **Course**. Hver gang feltet **examDate** til et **Course**-objekt endrer seg ønsker du å gi beskjed til alle studentene som er meldt opp til eksamen i emnet om endringen.

Hvilken teknikk er best egnet til å løse dette problemet?

Velg ett alternativ:

- Innkapsling
- Observatør-observert
- Arv
- Grensesnitt
- Delegering

Teoriprøven, eksempler

Basert kun på følgende kode, hva kan vi vite er feil med denne metoden?

```
public boolean validateNumberIsCorrectFormat(String number) {  
    if (!NumberValidator.isNumberCorrectFormat(number)) {  
        throw new IllegalArgumentException("Wrong format");  
    }  
    if (NumberValidator.isInvalidValue(number)) {  
        throw new IllegalArgumentException("Illegal value");  
    }  
}
```

Velg ett alternativ:

- Metoden burde utløse **IllegalStateException**
- NumberValidator** er ikke opprettet og vi kan derfor ikke bruke metodene derfra
- Metoden setter ikke **number** til å være noen verdi
- Metoden returnerer ikke noe