

TDT4100 - Objektorientert programmering

Observatør-observert

Dag Olav Kjellemo, med mye stoff fra

Børge Haugsets forelesninger tidligere år

Vær oppmerksom på at forelesningen vil bli tatt opp og gjort tilgjengelig på Blackboard.

For mer informasjon, se

<https://s.ntnu.no/video-opptak>

Læringsmål for forelesningen

- **Objektorientering**
 - Observatør-observert-samspill

Standard kodingsteknikker

- *“a design pattern is a general repeatable solution to a commonly occurring problem in software design”*
- Gjenbrukbare kodingsteknikker som har vist seg å være praktiske/effektive løsninger i mange



SMS-varsling

- En registrerer sin interesse for en viss type hendelser
- Når hendelsen inntreffer blir man varslet

Scoringsvarsel SMS

Få sms-varsling om scoringer og utvisninger fra ditt lags kamper.

SMS Varsling er lansert!

SMS

Nå kan du få tilsendt scoringsvarsel på alle kampene i Tippeligaen og Premier League. Du kan velge å få scoringer for en liga, et lag eller for en enkelt kamp.

[Les mer her](#)



Tannlegene tilknyttet det offentlige tannhelsetilbudet opplever at ni prosent av pasientene i alderen tre til 18 år ikke dukker opp.

– Dette koster oss rundt ti millioner kroner i året, anslår fylkestannlege Inge Magnus Bruvik.

NYTT DATASYSTEM

Nå innføres det **SMS-varsling** til alle mobilbrukere som kommer inn under tannhelsetjenesten.



Oppdatering av avhengig objekt

- En del av et program blir informert om noe som skjer i en annen del, f.eks.
 - GUI'et må hele tiden være oppdatert ift. intern tilstand
- Generelt er det lurt å skille mellom det som er GUI-spesifikt og den "indre" logikken.
 - La GUI reflektere «indre» tilstand, ikke tett integrert med
 - Det blir mindre arbeid å lage flere varianter av GUI'et, f.eks. JavaFX, mobil-Java eller HTML.
 - Det blir lettere å dele oppgaven på flere personer
- Det finnes en standard kodingsteknikk for å holde et objekt oppdatert ift. et annet.



Observert/observatør Oppdatering av avhengig objekt

- Ganske ofte må et objekt B vite når et annet objekt A er endret, for å sikre konsistens.
- Endring i objekt **a**, kaller en oppdateringsmetode i **b**.
- Gjøre dette på en fleksibel måte uten sterke bindinger mellom objektene
 - Mer anvendelig/gjenbrukbart



Oppdatere: rigid -> fleksibel

- Minst fleksibelt: «hardkode» kall til metode i B. Ulempe: Dette er statisk, A må vite om B
- Litt mer fleksibelt: B implementerer interface. Fortsatt statisk, men vi kan oppdatere andre typer objekter med samme kall.
- Løse bindingen ytterligere: bruke grensesnitt.
- En enda mer generell løsning er en generisk meldingssentral (postvesen?)
- Et helt generelt oppsett er kanskje ikke hensiktsmessig, (typesikkerhet, en må jo uansett vite om hva som skal kunne meldes gjennom systemet)

Idol, Idolfan og Beundrer

- Et Idol har en hårfarge som Beundrer ønsker å kopiere.
- Hver gang Idol skifter hårfarge ønsker Beundrer å gjøre det samme.
- Idolet på sin side nyter å bli beundret og vil gjerne hjelpe Beundrer.
- Idol går med på å si fra til Beundrer hver gang hårfargen skifter.
- Hva skjer når fargen endres?

```
public class Idol {  
    private Color hårfarge = Color.white;  
    private Beundrer beundrer;  
  
    public Idol() {  
    }  
  
    public void setHårfarge(Color c) {  
        hårfarge = c;  
        beundrer.hårfargeEndret(c);  
    }  
  
    public void setBeundrer(Beundrer beundrer) {  
        this.beundrer = beundrer;  
    }  
}
```

```
public class Beundrer {  
    private Color hårfarge;  
  
    public Beundrer() {  
    }  
  
    public void setHårfarge(Color c) {  
        hårfarge = c;  
    }  
  
    public void hårfargeEndret(Color c) {  
        setHårfarge(c);  
    }  
}
```

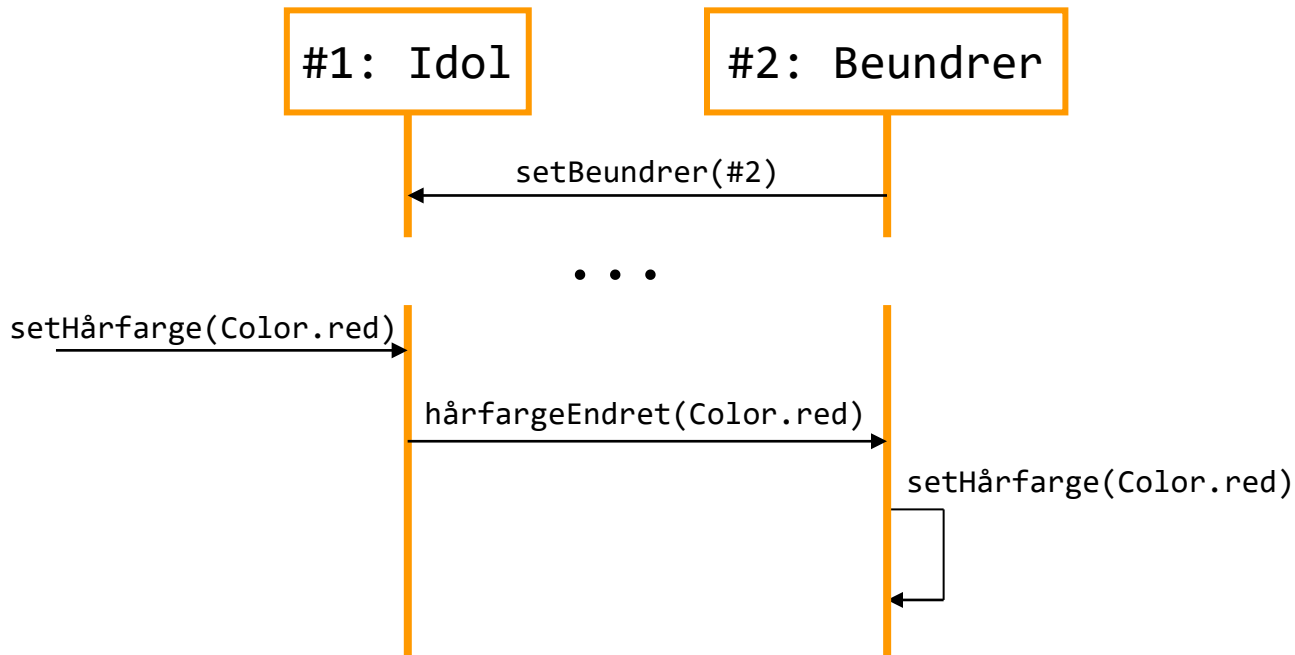


Generell kodingsteknikk

- Beundrer vil vite når Idol endrer tilstand
- I dette tilfellet
 - Idol = observert
 - Beundrer = observatør
- Beundrer ønsker å få beskjed om endringer
 - Idol må kjenne til beundrer, ved at beundreren kan registrere seg
 - Ved endringer vil Idol gi beskjed (sende melding)
- Idol må ha en metode slik at en beundrer kan legges seg til
- Beundrer må ha en metode slik at han/hun kan få beskjed
- Ved endring må Idol sørge for at observatør (beundrer) får beskjed om endringen



Sekvensdiagram for skifte av hårfarge



Problemer med Idol-Beundrer-forholdet (1)

- Et Idol kan ha mange flere egenskaper som Beundrere kan ønske å kopiere.
 - En kan lage flere xxxEndret-metoder, eller
 - En kan ha en generell metode noeEndret som tar et "hva"-parameter som forteller hvilken egenskap som endret seg. Beundrer henter selv ut verdien.
- En Beundrer kan ha flere idoler
 - Endringsmetoden bør ta som parameter hvilket Idol som endret seg.




Problemer med Idol-Beundrer-forholdet (2)

- Dersom Idol har en beundrerskare, bør alle behandles likt, dvs. få beskjed om endringer.
 - Vi bytter ut Beundrer med liste av Idolfans.
- Dersom Beundrer har mange flere metoder, er det greit å definere eksplisitt hvilke(n) metoder som Idol krever at Beundrer har, nemlig xxxEndret evt. noeEndret.
 - Vi kan definere et *grensesnitt* **Idolfan** som inneholder relevante metoder.
 - Dette vil la hvilken som helst klasse fungere som beundrer, også nye klasser




Idolfan og Idol

- Idolfan-grensesnitt med én metode for å si fra om endring
- Idol har liste med fans.
- Hver gang en egenskap endres, må listen med fans på beskjed.
- Lager hjelpemetode for å gå gjennom fan-lista og kalle endringsmetoden.
- Ekstra metoder for å legge til og fjerne fans fra lista.

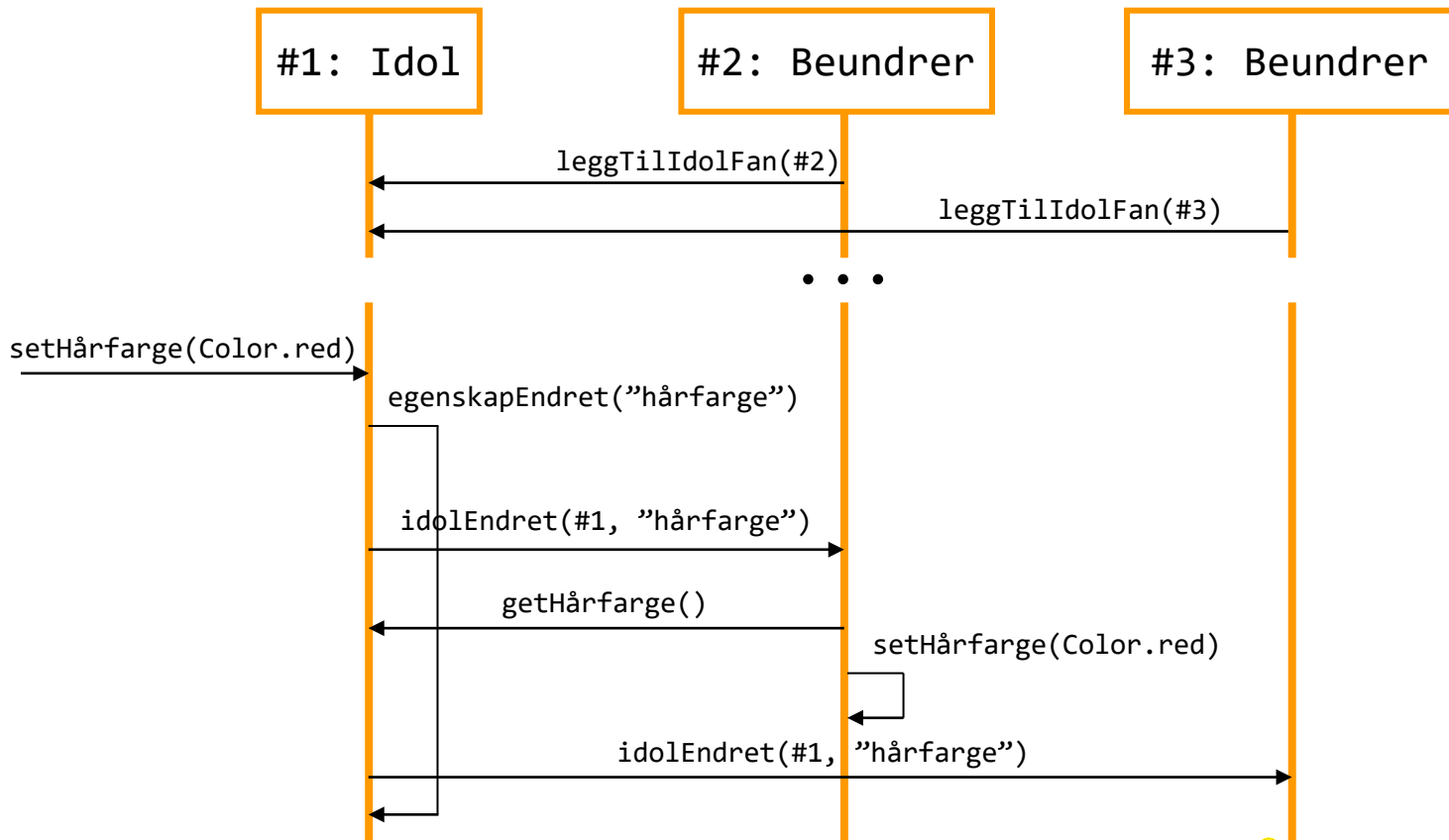


```
public interface Idolfan {  
    public void idolEndret(Idol idol, String hva);  
}
```

```
public class Idol {  
    private Color hårfarge = Color.white;  
    private List<Idolfan> fans;  
  
    public Idol() {  
    }  
  
    public Color getHårfarge() {  
        return hårfarge;  
    }  
    public void setHårfarge(Color c) {  
        hårfarge = c;  
        egenskapEndret("hårfarge");  
    }  
  
    private void egenskapEndret(String hva) {  
        for (int i = 0; i < fans.size(); i++) {  
            fans.get(i).idolEndret(this, hva);  
        }  
    }  
    public void leggTilIdolFan(Idolfan fan) {  
        if (! fans.contains(fan)) {  
            fans.add(fan);  
        }  
    }  
    public void fjernIdolFan(Idolfan fan) {  
        fans.remove(fan);  
    }  
}
```



Nytt sekvensdiagram for skifte av hårfarge



Generell struktur på observatør-observert-problem

- Observert er et objekt som sier fra til andre når noe er endret.
- Observatøren er objektet som ønsker å følge med på en eller flere observerte.
- Kodingsteknikk
 - Et observatør-grensesnitt definerer metoder for å få beskjed om endringer.
 - Observert har en liste med observatører.
 - Hver gang en (relevant) endring skjer, må alle observatører få beskjed, ved at en observatør-metode blir kalt (callback).



Variasjoner på løsning på observatør-observert-problem

Observatør-grensesnittet kan ha én eller flere metoder:

- Én metode:
 - krever flere parametere for å si hva som er endret, men har fordelen av grensesnitt blir funksjonelt og en kan bruke Java 8-syntaksen for funksjonsobjekter
 - Disse kan alternativt legges i et eget objekt, som en *endringshendelse*, med referanser til hvilken egenskap ved hvilket objekt som ble endret
- Mange metoder:
 - Mer direkte/eksplisitt, lettere å se i kildekoden hva som skjer,
 - men mindre fleksibelt.
 - kan gjøre **observatør**-grensesnittet stort, og kanskje ikke alle metoder er relevante for alle klasser.



Observatør-mønstret i Java

Det er mange eksempler i Java-bibliotek som benytter seg av observatør-mønstret eller tilbyr mekanismer som vi kan bruke.

Vi skal se på to:

- `Java.beans.Property...` grensesnitt og klasser
- Brukt i JavaFX (som vi skal jobbe mer med om litt)

(Det finnes et *utdatert* grensesnitt (pre Java 9) `java.util.Observable` og `Observer`. Ikke prøv å bruke dette.)

PropertyChange... fra java.beans

- **observatør-grensesnitt `PropertyChangeListener`**
 - med metoden **`propertyChange`**
- **`PropertyChangeEvent`-klasse**
 - som inneholder informasjon om endringen
- **`PropertyChangeSupport`**
 - et hjelpeobjekt vi delegerer håndtering av observatører og for varslingsmekanismen.

Se kode-eksempel `PersonBean`

PropertyChange... fra java.beans

Den observerte må implementere:

- **add/removePropertyChangeListener**-metoder
- en metode (ofte kalt **firePropertyChange**) for å lage og sende endringshendelser
- For hver logiske "property" defineres:
 - **String**-konstant med navnet på property'en
 - privat felt for å holde verdien
 - **set<Property>**-metode, som først endrer det private feltet og så kaller metoden for å sende hendelser

Observatøren må implementere **PropertyChangeListener**-grensesnittet med

- **propertyChange**-metoden

Se kode-eksempel **PersonBean**



NTNU

NB! I disse eksemplene har vi ikke noen underliggende modell, alt er implementert i GUI'et.

Dette skal vi endre på når vi kommer til GUI-programmering med JavaFX.



Observatør-mønsteret I JavaFX

(se **Main.java**)

- I JavaFX er grafiske elementer representert ved objekter (no way!)
- En `textField` har et felt brukeren kan skrive i, og lagrer dette i et *observerbart* property-felt som innkapsler selve teksten.
- **Label-objekt** som blir oppdatert.

```
//utdrag fra uke10.observerfx
```

```
TextField textField = new TextField();  
Label label = new Label();
```

```
// Add a listener to the TextField's text property  
textField.textProperty().addListener(new  
ChangeListener<String>() {  
    @Override  
    public void changed(ObservableValue<? extends  
String> observable, String oldValue, String  
newValue) {  
        System.out.println(observable); // for å  
logge kallet og hva observable-objektet inneholder  
        label.setText(newValue);  
    }  
});
```





Observatør-mønsteret I JavaFX

(se `Hello.java`)

- Vi registrerer hos Button-objektet **btn** en et lambda-objekt som skriver «Hello» til konsollet når action-eventet «fires» (f.eks. trykke på knappen)

// Utdrag fra uke10/observerfx/Hello.java

```
Button btn = new Button();  
btn.setText("Say 'Hello'");  
btn.setOnAction(event ->  
    System.out.println("Hello"));
```

