Uke 8

Programmeringsmønstre

Hva er et programmeringsmønster?

- Standardisert kode vi ofte bruker for å oppnå noe
- Kan skrives fra bunnen av, eller implementere et grensesnitt
- Designmønstre dere har sett tidligere:
  - Iterator
  - Decorator?
- Nye designmønstre:
  - Singleton
  - Observer

Hvorfor bruker vi programmeringsmønstre?

La oss se på iterator

Vi kan lett gå igjennom en array slik:

```
int[] tall = {...}

for ( int i = 0; i < tall.length; i++ ) {
  System.out.println(tall[i]);
}
```

Men hva om vi bruker en annen type beholder?

```
LenketListe<Integer> tall = {...}

for ( int i = 0; i < tall.length; i++ ) {
  System.out.println(tall[i]);
}
```



Da har vi et designmønster som fungerer for begge:

```
LenketListe<Integer> tall = {...}

Iterator it = tall.iterator();

while( it.hasNext() ) {
  System.out.println( it.next() );
}
```

Dette designmønsteret er formalisert i Java:

```
LenketListe<Integer> tall = {...}

Iterator it = tall.iterator();

for (Integer t : tall) {
  System.out.println(t);
}
```

Decorator

Se for dere at dere skal lage et billettsystem for en kino.

Kinoen tilbyr vanlige billetter, men billettene skal også ha tillegg som ekstra benplass, ekstra behagelige seter, og inkluderte snacks. Man kan kombinere flere tillegg på én billett.

Hvordan ville dere implementert dette?

```java
public interface Billett {
    int hentPris();
    String hentBeskrivelse();
}
public class VanligBillett implements Billett {
    @Override
    public int hentPris(){ return 60; }

    @Override
    public String hentBeskrivelse() {
        return "Kinobillett";
    }
}
public abstract class Dekorator implements Billett {
    protected Billett temp;
    public Dekorator (Billett b){ temp = b; }

    @Override
    public int hentPris(){ return temp.hentPris(); }

    @Override
    public String hentBeskrivelse() {
        return temp.hentBeskrivelse();
    }
}
```

```java
public class BedreSeter extends Dekorator {

  public BedreSeter(Billett b) {
    super(b);
  }

  @Override
  public int hentPris(){ return temp.hentPris() + 40; }

  @Override
  public String hentBeskrivelse() {
    return temp.hentBeskrivelse() + ", i behagelig sete";
  }
}
public class EkstraBenplass extends Dekorator {
    public EkstraBenplass(Billett b) {
        super(b);
    }

    @Override
    public int hentPris(){ return temp.hentPris() + 30; }


    @Override
    public String hentBeskrivelse() {
        return temp.hentBeskrivelse() + ", med ekstra
benplass";
    }
}
public class MedSnacks extends Dekorator {
    public MedSnacks(Billett b) {
        super(b);
    }

@Override
public int hentPris(){ return temp.hentPris() + 50; }

@Override
public String hentBeskrivelse() {
return temp.hentBeskrivelse() + ", med snacks";
}
}
```

Observer

Hvordan effektivt “fortelle” objekter at noe har skjedd

```java
// Subjektet (en klasse vi designer sånn vi vil)
class YouTubeChannel {
  // Lager en liste av Observers
    private List<Subscriber> subscribers = new ArrayList<>();

    public void subscribe(Subscriber s) {
        subscribers.add(s);
    }

  // På metoden vi vil observere, looper vi gjennom listen.
    public void uploadVideo(String title) {
        System.out.println("Uploading: " + title);
        for (Subscriber s : subscribers) {
            s.update(title);
        }
    }
}
```

```java
// Grensesnittet felles for alle Observatører
interface Subscriber {
    void update(String videoTitle);
}

// En implementasjon (vi kan ha flere)
class VideoFan implements Subscriber {
    private String name;

    public VideoFan(String name) { this.name = name; }

    @Override
    public void update(String videoTitle) {
        System.out.println(name + "! New video is out: " + videoTitle);
    }
}
```

Singleton

En klasse med kun ett objekt

```
class Singleton {
  private static Singleton instance;

  private Singleton();

  public static getInstance() {
    if (instance == null) instance = new Singleton();
    return instance;
  }
}
```

Nå, seminaroppgave!