

IN1010 - Programmeringsmønstre

Eyvind W. Axelsen - eyvinda@ifi.uio.no

```java
public class GodEttermiddag {
  public static void main(String[] args) {
    System.out.println("Hei på deg!");
    System.out.println("Du bør kunne lese dette - sitter du for langt bak?");
  }
}
```

Gjenbruk

- Når vi programmerer så er *gjenbruk* av kode en viktig del av hva vi holder på med
- Dette gjøres gjerne ved hjelp av bibliotek, f.eks. standardbibliotek som følger med Java og Python.
  - F.eks.

```java
import java.util.HashMap;

public class HashTest {
    Run | Debug
    public static void main(String[] args) {
        HashMap<String, String> hovedsteder = new HashMap<String, String>();
        hovedsteder.put(key: "Norge", value: "Oslo");
        hovedsteder.put(key: "Aserbajdsjan", value: "Baku");
    }
}
```

Men: kanskje enda viktigere enn gjenbruk av kode er...

«Gjenbruk» av *<u>kunnskap</u>*:

At vi lærer oss hvordan man kan kjenne igjen, og gå frem for å løse gitte typer av problemstillinger

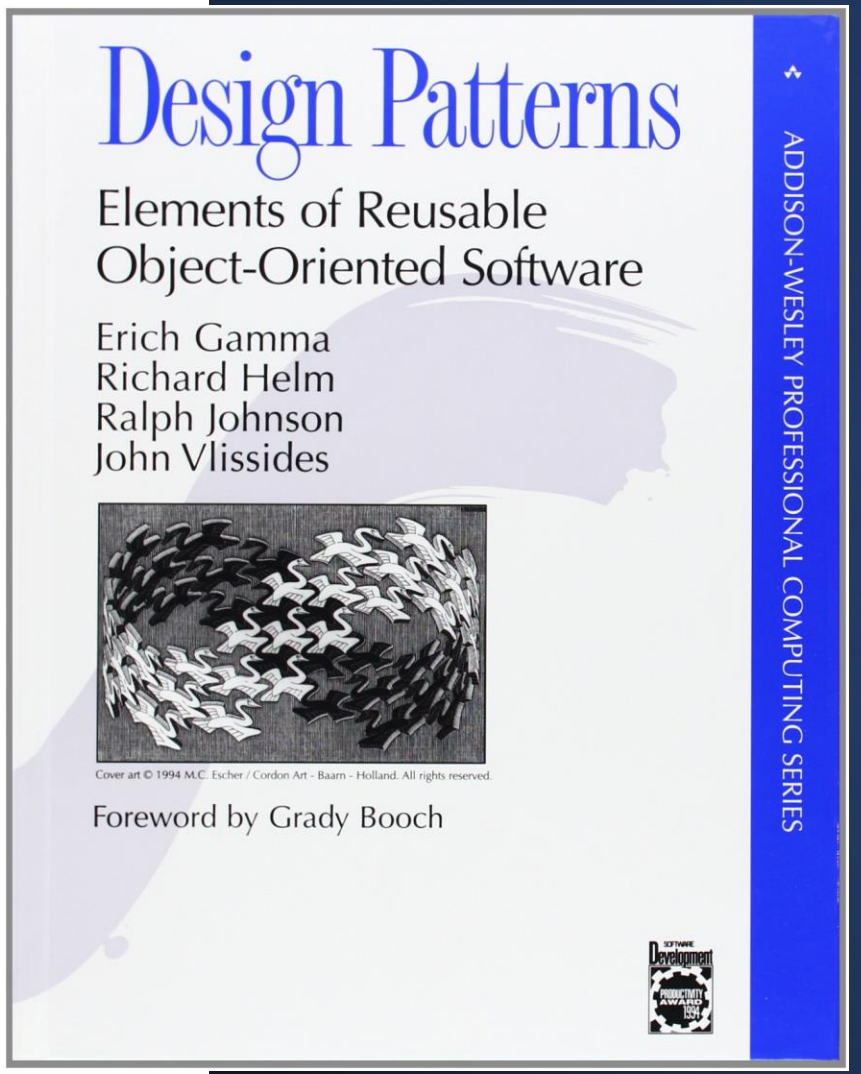


Programmeringsmønster = Design Pattern

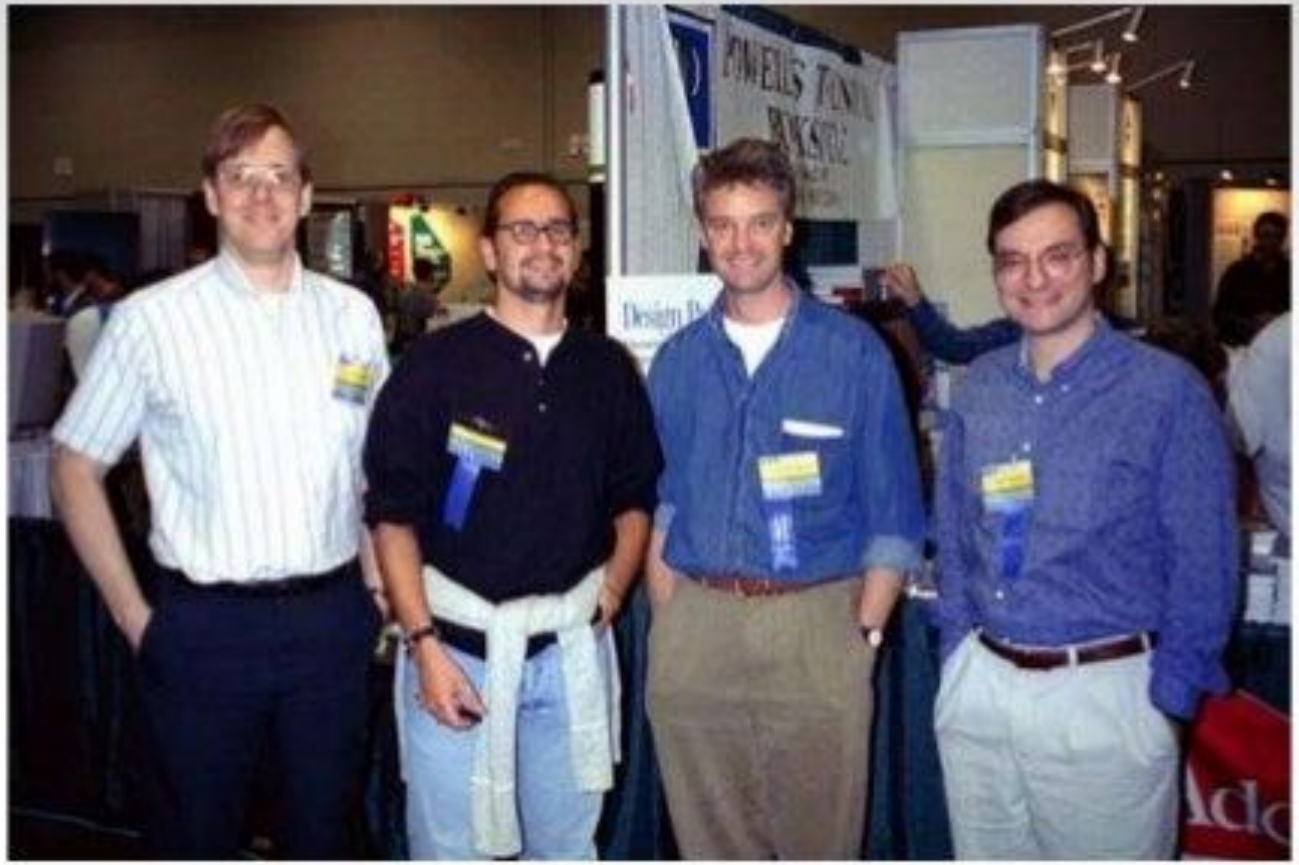
Gang of Four: Gamma, Helm, Johnson, Vlissides:

Hvilke mønstre har folk faktisk brukt for å løse problemer?

Hvordan strukturere kode for å løse kjente klasser av problemer



OOP Design Patterns: The Gang of Four!



It all started around 1994 when a group of 4 IBM programmers:
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
based on their coding experience , were able to observe and document
a set of about 23 common problems and their best accepted solutions

< https://www.drupal.org/project/oop_design_patterns >

Bakgrunn

- Idéen om designmønstre stammer fra byplanlegging
  - The Pattern of the Streets, C. Alexander, 1966, Journ. American Institute of Planners
  - Beskriver mønstre som gjenbrukbare designkonsepter for veier i storbyer

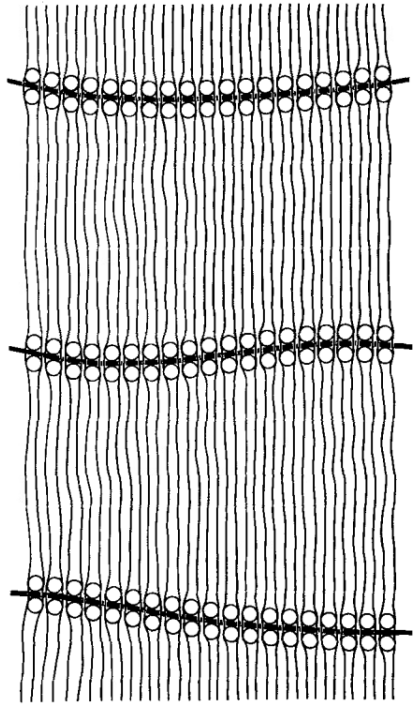


FIGURE 1
The streets and freeways are deliberately drawn crooked. It is not essential that they be straight, only that they be roughly parallel. Their exact alignments will be determined by local variations in land-use and topography.

THE SOLUTION

FIGURE 2



FIGURE 3

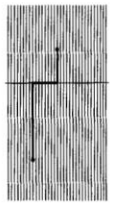
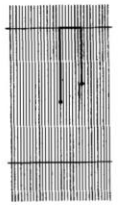


FIGURE 4



The only space-filling pattern of lines which is free from intersections, and yet entirely at one level, is a packing of parallel lines. Requirement 1, 2, and 4 therefore demand a pattern of parallel streets without cross streets. To solve requirement 3, streets are connected by infrequent freeways which run at right angles to the streets. How far apart can these freeways be before they start creating serious detours?

Suppose that the freeways are m miles apart. Let us call each area between two freeways, a "band." Each band is m miles wide. There are now two essentially different kinds of trip:

a) Those which start and finish in different bands, shown in Figure 3. These are exactly the same length they would be if cross-streets existed.

b) Those which start and finish in the same band, shown in Figure 4. These are longer than they would be if cross-streets existed. The amount of the detour is twice the shorter distance to the nearest freeway. Given independent random distributions of starting points and destinations, integration shows that the mean detour, on such a trip, will be $m/3$ miles.[6]

Let us now ask what proportion of all trips do in fact start and finish in the same band. We begin by tabulating the frequencies of different trip lengths in a modern metropolis. The figures below are from San Diego.[7]

We observe now, that for any given trip length ℓ, some trips will fall entirely within a band, while others will start in one band and finish in another. We may think of the trips as lines of length ℓ falling at random onto the city, and assume that trips are equally likely to fall at all angles to the freeways. The probability then that a trip of length ℓ will fall entirely within a band is the same as the probability that a matchstick of length ℓ, thrown at random onto a pattern of parallel lines m apart, will fall entirely between two lines. This is given by the standard formula:[8]

$$p(m,\ell) = 1 - \frac{2\ell}{\pi m} \text{ if } \ell \leq m$$

$$= \frac{2}{\pi}\left[\arcsin\frac{m}{\ell} - \frac{\ell}{m} + \frac{\sqrt{\ell^2 - m^2}}{m}\right] \text{ if } \ell > m$$

For m equal to 1 mile, 2 miles, and 3 miles respectively, this function yields the following probabilities of a one-band trip, for the trip lengths shown:

Design patterns – designmønstre i programmering

- Tanken om at det finnes gode mønstre for hvordan man løser vanlige problemer er også anvendbart for programmering

- Har du sett, eller programmert, noen mønstre i koden som du kjenner igjen?

Klassifisering av programmeringsmønstre

- Behavioral – identifiserer hvordan objekter kommuniserer, og hvordan slik kommunikasjon kan realiseres som klasser og objekter
  - Eksempelvis Command, Iterator, Observer, Visitor
- Creational – kontroll over hvordan objekter lages, utfra gitte forutsetninger
  - Eksempelvis Abstract Factory, Builder, Singleton
- Structural – hvordan man kan strukturere klasser og objekter for å løse gitte former for problemer
  - Eksempelvis Adapter, Bridge, Decorator, Proxy

En god oversikt over mange patterns: https://en.wikipedia.org/wiki/Software_design_pattern

Iterator-mønsteret – repetisjon fra forelesning 27.02

Itererbar Arrayliste – endret:

- Liste<T> extends Iterable<T>
  - Utvidet med metoden **iterator**
- ArrayListe
  - implementerer metoden **iterator**
  - .. som oppretter Iterator-objekt for Arrayliste
  - .. fra den indre klassen ListeIterator som tilbyr **hasNext-** og **next**-metoder
- TestArrayliste kan nå
  - bruke for-each (Java oppretter og bruker Iterator-objekt bak kulissene) - eller
  - lage et eget Iterator-objekt for å hente ut elementer i rekkefølge (må da importere java.util.Iterator)

```
import java.util.Iterator;
public class Arrayliste<T> implements Liste<T>{
    @SuppressWarnings("unchecked")
    private T[] data = (T[]) new Object[10];
    private int iBruk = 0;

    class ListeIterator implements Iterator<T> {
        private int pos = 0;
        @Override
        public T next() {
            pos++;
            return data[pos-1];
        }
        @Override
        public boolean hasNext() {
            return pos < iBruk;
        }
    }

    @Override
    public Iterator<T> iterator() {
        return new ListeIterator();
    }

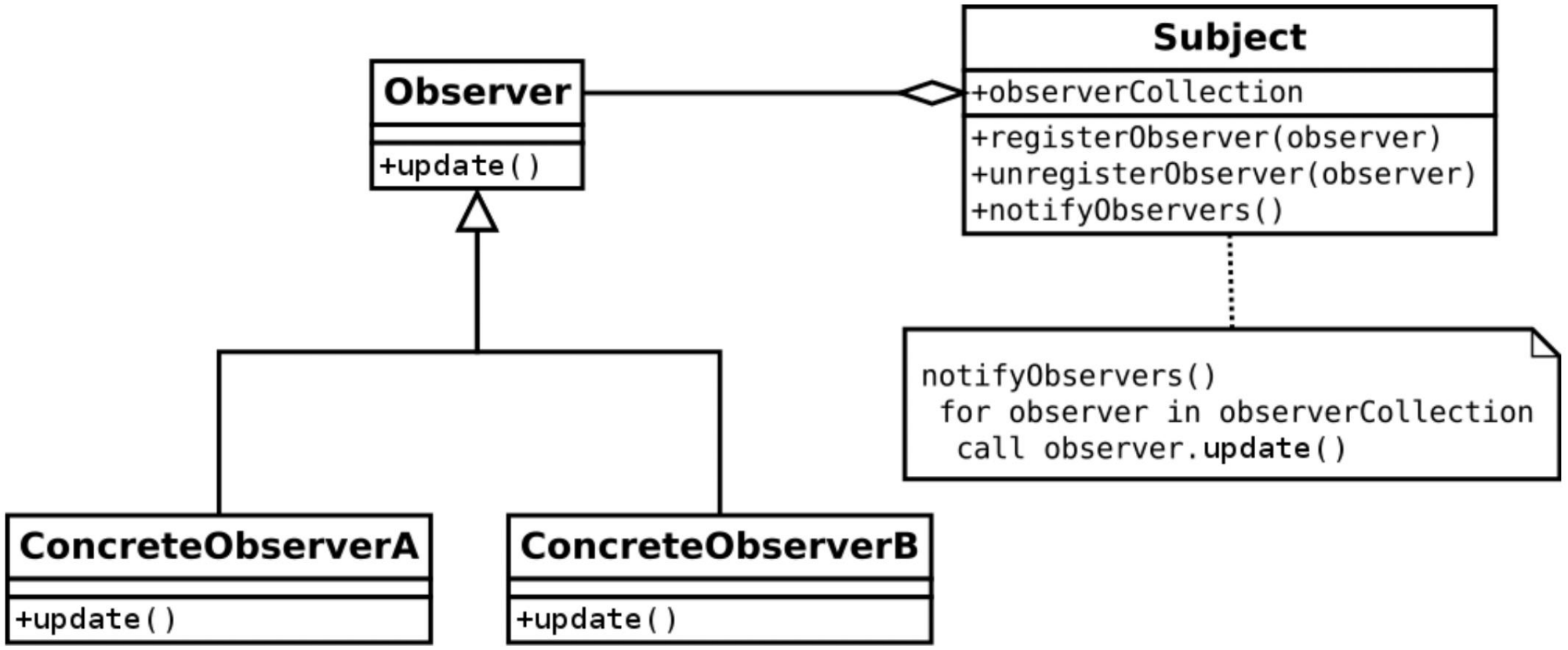
    // Andre metoder
```

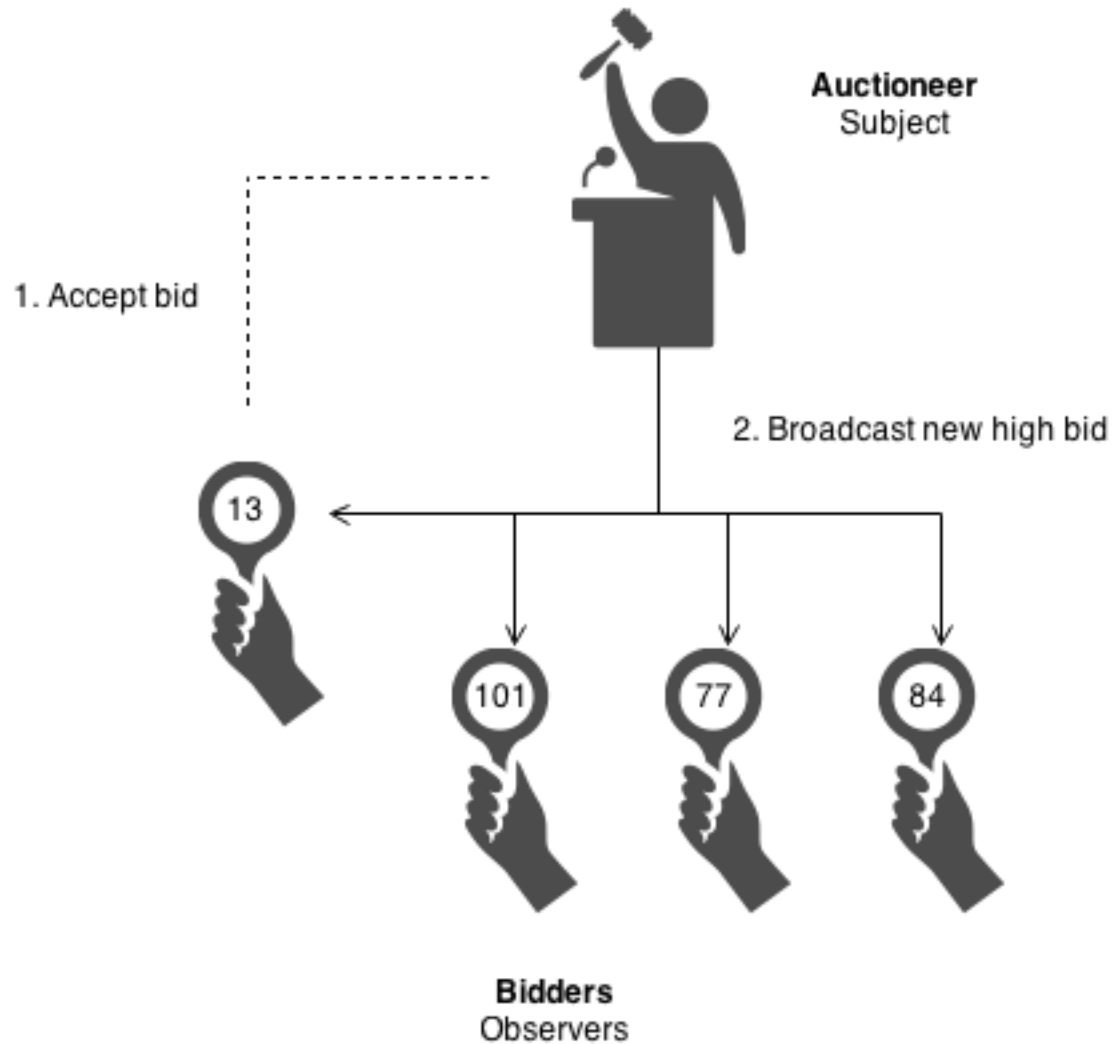
Observatør-mønsteret («Observer pattern»)

- Mønsteret har to *roller*
  - En *observatør* er interessert i å få beskjed når *noe* endrer seg
  - Et *subjekt* kan observeres av en eller flere *observatører*

- Mønsteret forsøker å løse følgende:
  - Man etablerer et en-til-mange-forhold mellom objekter, uten å gjøre dem tett koblet
  - Når tilstand endres i ett objekt, så kan et antall andre få beskjed

UML-diagram for Observer-pattern fra GoF-boka




Auctioneer
Subject
1. Accept bid
2. Broadcast new high bid
13
101
77
84
Bidders
Observers

Eksempel

- Vi har nyhetskilder («feeds»)
- Og noen som er interessert i å få beskjed når det kommer nyheter i feeden



interface Subject

void addObserver(Observer o)
void removeObserver(Observer o)
void notifyObservers()

observerer

blir observert av

interface Observer

void update(Subject s)

implementerer

implementerer

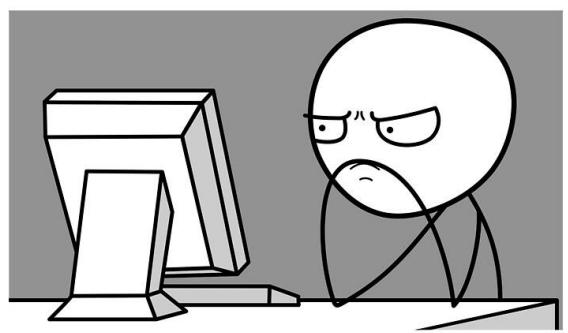
class NewsFeed

void newArticle(headline, url)

class Reader

void subscribe(NewsFeed n)

La oss se på litt kode!



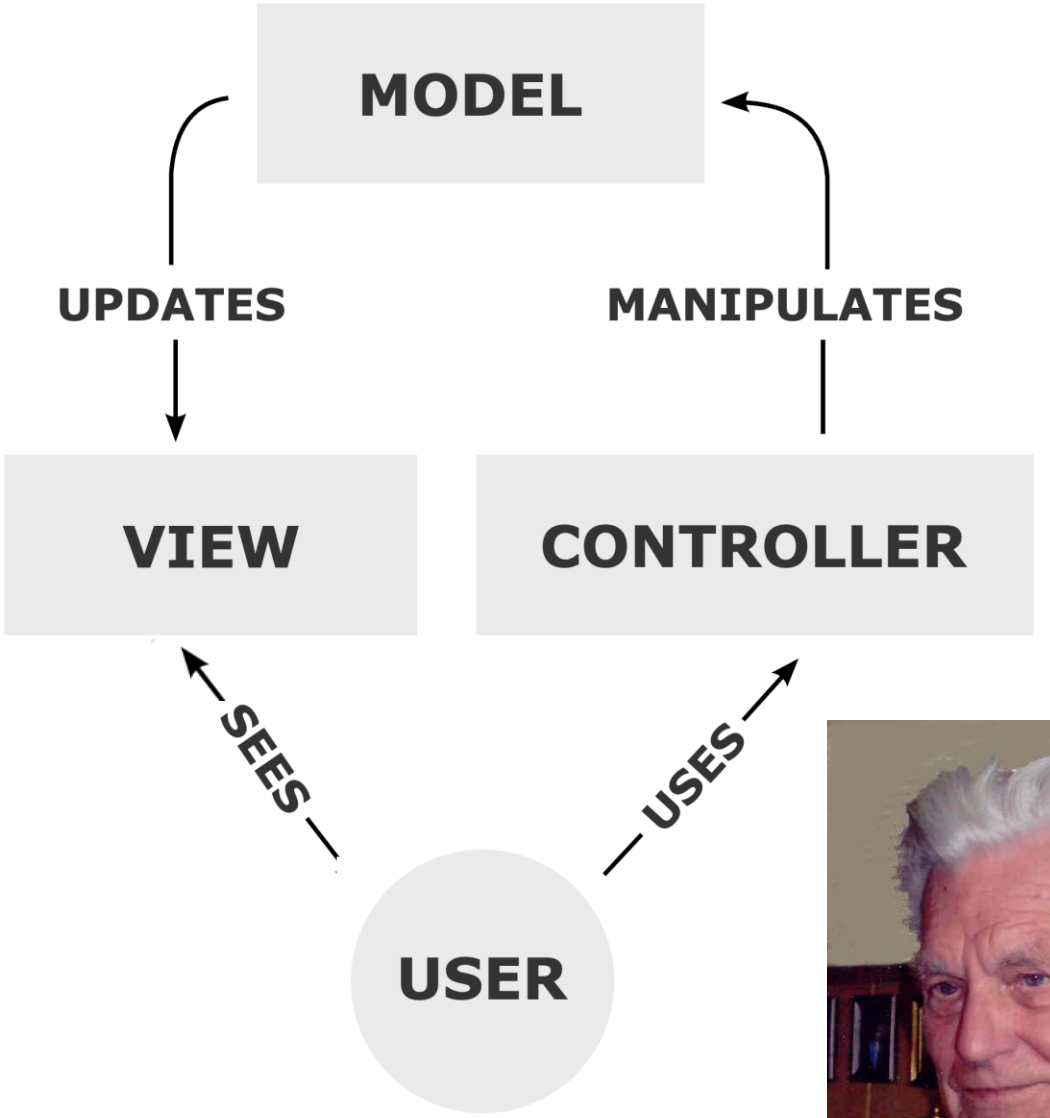
Observer oppsumert

- Skiller mellom *observatører* og *subjekter*
- Nyttig når man har observatører som er interessert i endringer i subjekter
- Flere observatører kan lytte til endringer på ett subjekt
- Et subjekt kan ha flere observatører som lytter på sine endringer

- Innebygd i Java i flere former:
  - Predefinert Observer interface
  - java.util.EventListener
  - javax.servlet.http.HttpSessionBindingListener
  - etc

Model – View - Controller

- Designmønster formulert av Trygve Reenskaug, ca 1979
- Hvordan strukturerer man en GUI-applikasjon?
- Har vært veldig populært i senere tid, særlig i web-applikasjoner (ASP.NET MVC f.eks.)
- View-et som følger med på modellen kan bruke observer-mønsteret

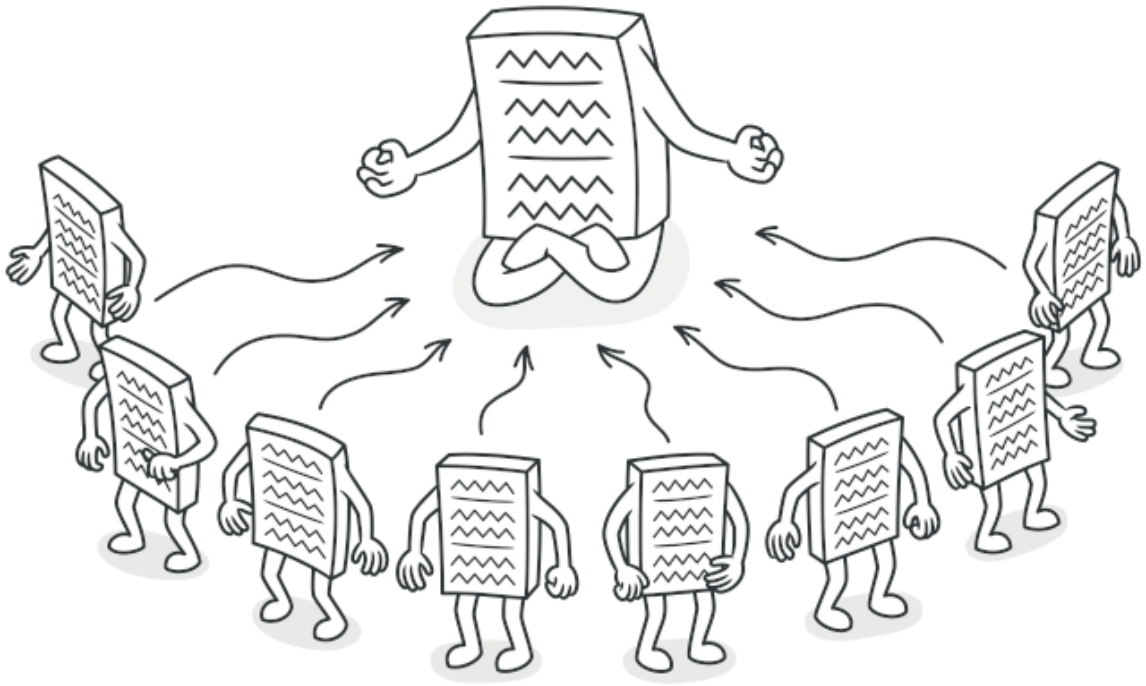




Singleton-mønsteret

- Av og til er det nyttig at vi kan være sikre på at det kun finnes ett eneste objekt av en gitt type

- En Singleton sørger for at
  - Det opprettes maks én objektinstans av en gitt klasse
  - Det er enkelt å få tilgang til denne instansen
  - Kan i tillegg ha støtte for «lat» initialisering



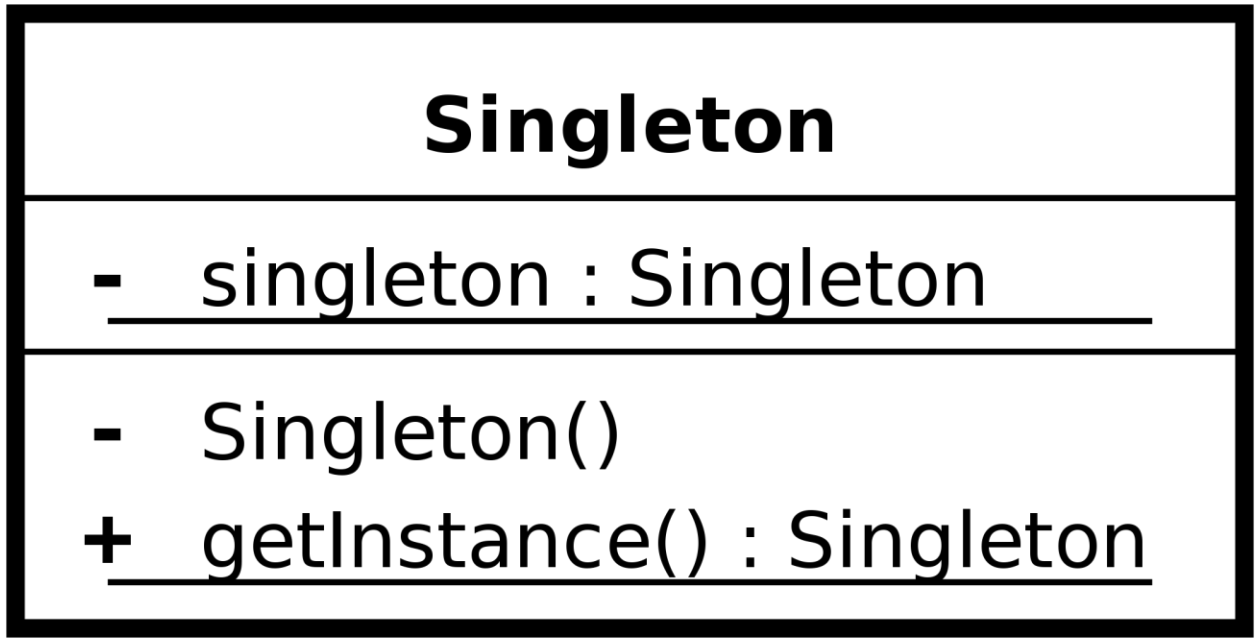
https://refactoring.guru/design-patterns/singleton

Hvorfor trenger man singleton-objekter?

- Forslag?

- Tilgang til delte ressurser, f.eks. databaser, disk, logging, etc
- Kontrollere tilgang til sikkerhetskritiske ressurser
- Begrense ressurskrevende objekter til en enkelt instans
- Etc.

Singleton klassediagram fra GoF-boka



```java
public class Singleton {

    private static Singleton instance = null;

    private Singleton() { }

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

        return instance;
    }
}
```

Instansen er statisk – den er global og delt av alle

Konstruktøren er private, så ingen kan si new Singleton();

getInstance()-metoden sjekker om en instans er opprettet, og oppretter en ny hvis ikke. Deretter returneres denne.

Men: vi har et problem her – ser noen hva det er?
(Hint: dette har ikke vært forelest ennå 😱)

Frampek mot forelesningen 9. april

I mange typer programmer, f.eks. dataspill, så må flere ting skje *samtidig.*

Dette løser vi ved hjelp av noe som kalles *tråder*, som er biter av programmet som kjører i parallell.



spillMusikk()

flyttFiender()

tegnMario()

...

Tråd A ⟷ Kjører samtidig ⟷ Tråd B

```
public class Singleton {

    private static Singleton instance = null;

    private Singleton() { }

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

        return instance;
    }
}
```

```
Singleton a =
    Singleton.getInstance();
```

```
instance = new Singleton();
```

```
Singleton b =
    Singleton.getInstance();
```

```
instance = new Singleton();
```



```java
public class Singleton {

    private static volatile Singleton instance = null;
    private static ReentrantLock l = new ReentrantLock();

    private Singleton() { }

    public static Singleton getInstance() {
        if (instance == null) {
            l.lock();
            try {
                if (instance == null) {
                    instance = new Singleton();
                }
            } finally {
                l.unlock();
            }
        }
        return instance;
    }
}
```

Volatile betyr at endringer skal gjøres tilgjengelig for andre tråder umiddelbart

Bruk av lock gjør at bare en tråd aksesserer denne biten av koden av gangen

Mer om dette senere 😬

```java
import java.util.concurrent.locks.ReentrantLock;

public class RessurskrevendeTing {

    private static volatile RessurskrevendeTing instance = null;
    private static ReentrantLock l = new ReentrantLock();

    private RessurskrevendeTing() { }

    public static RessurskrevendeTing getInstance() {
        if (instance == null) {
            l.lock();
            try {
                if (instance == null) {
                    instance = new RessurskrevendeTing();
                }
            } finally {
                l.unlock();
            }
        }
        return instance;
    }

    public void metodeSomBrukerMasseRessurser() {
        /* gjør noe ressurskrevende greier her... */
    }
}
```

Singleton oppsummert

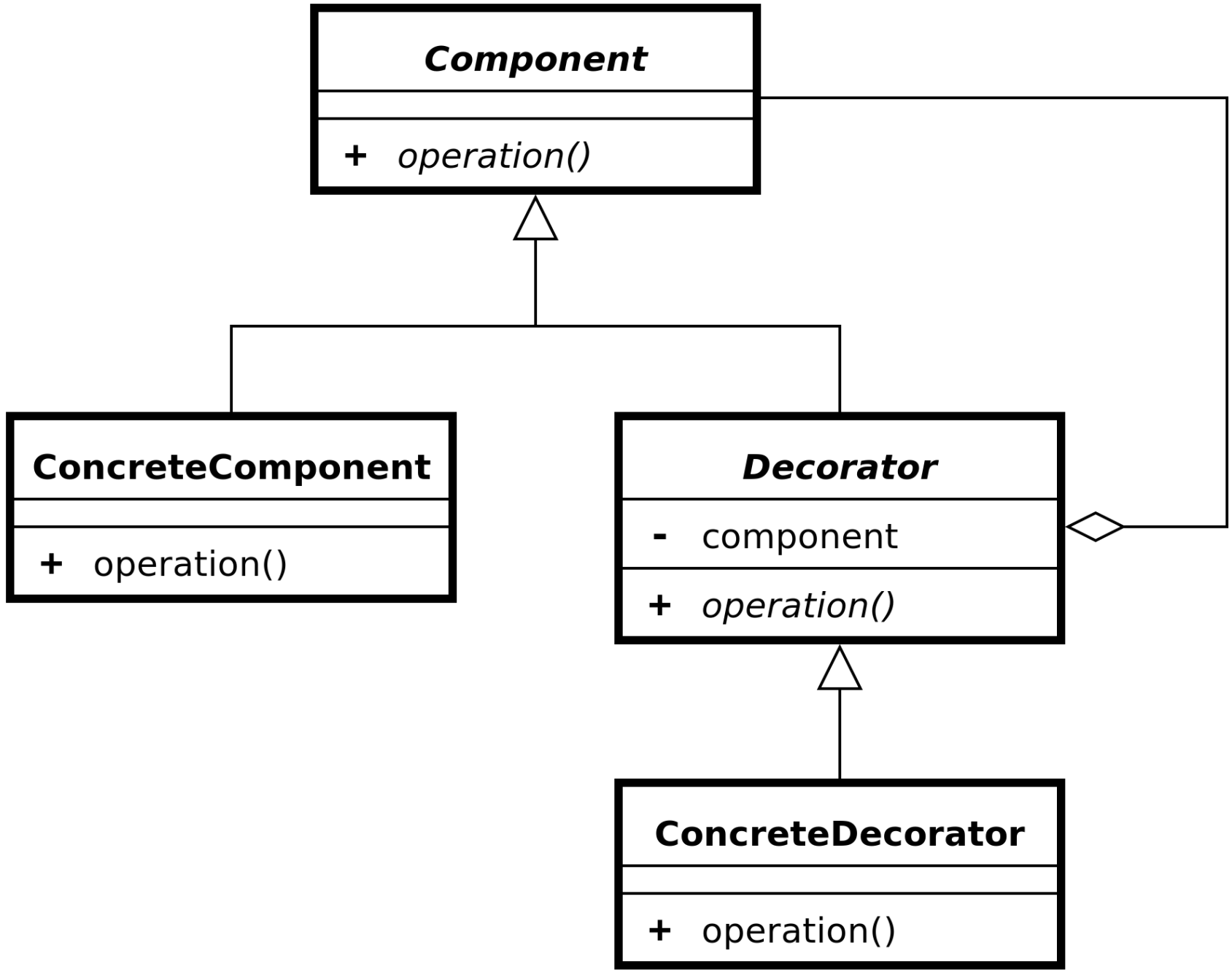
- Et enkelt mønster når du bare vil ha én objektinstans av en gitt klasse i programmet ditt.
- Eksempler fra Javas API:
  - java.awt.Desktop.getDesktop()
  - java.lang.Runtime.getRuntime()
- Når vi har flere tråder må vi passe litt ekstra på (mer om det på senere forelesninger)

- Kritikk
  - Introduserer en global tilstand i programmet – kan gjøre det vanskeligere å resonnere rundt programmet
  - Kan øke graden av kobling («coupling») i programmet

Decorator-mønsteret

- Lar deg legge til eller fjerne funksjonalitet fra et objekt under kjøring
- Kan være et alternativ til arv/subklassing
- Vi skal se på et eksempel: kaffe, som kan serveres med eller uten diverse egenskaper




Component
+ operation()
ConcreteComponent
+ operation()
Decorator
- component
+ operation()
ConcreteDecorator
+ operation()

```java
interface Coffee {
    public double getCost(); // Kaffens kostnad
    public String getIngredients(); // Kaffens ingredienser
}
```

```java
// En helt vanlig kaffe
class SimpleCoffee implements Coffee {
    @Override public double getCost() {
        return 10; // koster 10 kr
    }

    @Override public String getIngredients() {
        return "Kaffe"; // inneholder bare kaffe
    }
}
```

```java
abstract class CoffeeDecorator implements Coffee {
    private final Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee c) {
        this.decoratedCoffee = c;
    }

    @Override public double getCost() {
        return decoratedCoffee.getCost();
    }

    @Override public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}
```

```java
class WithMilk extends CoffeeDecorator {
    public WithMilk(Coffee c) {
        super(c);
    }

    @Override public double getCost() {
        return super.getCost() + 5; // melk koster 5 kr ekstra
    }

    @Override public String getIngredients() {
        return super.getIngredients() + ", Melk";
    }
}
```

```java
class WithSugar extends CoffeeDecorator {
    public WithSugar(Coffee c) {
        super(c);
    }

    @Override public double getCost() {
        return super.getCost() + 2.5; // sukker koster 2,50 kr ekstra
    }

    @Override public String getIngredients() {
        return super.getIngredients() + ", Sukker";
    }
}
```

```java
public class CoffeeMain {
    public static void printInfo(Coffee c) {
        System.out.println("Kostnad: " + c.getCost() +
            "; Ingredienser: " + c.getIngredients());
    }

    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();
        printInfo(c);

        c = new WithMilk(c);
        printInfo(c);

        c = new WithSugar(c);
        printInfo(c);
    }
}
```

```
src — -zsh — 80×24
(base) eyvinda@Crushinator src % java CoffeeMain
Kostnad: 10.0; Ingredienser: Kaffe
Kostnad: 15.0; Ingredienser: Kaffe, Melk
Kostnad: 17.5; Ingredienser: Kaffe, Melk, Sukker
```

Decorator oppsummert

- Brukes for å utvide funksjonalitet til et objekt
- Dette kan gjøres dynamisk (under kjøring), i motsetning til subklassing, som bare skjer statisk (ved kompilering)
- Men: den ønskede utvidelsen må være planlagt på forhånd – man kan f.eks. likevel ikke legge til nye metoder under kjøring

- Eksempler fra Javas API:
  - Alle subklasser til java.io.InputStream, OutputStream, Reader og Writer har en konstruktør som tar inn et objekt som dekoreres. Eksempel:

```
BufferedReader in
    = new BufferedReader(new FileReader(fileName:"foo.in"));
```

Programmeringsmønstre – design patterns

- Programmeringsmønstre er gjenbrukbare måter å strukturere kode på
- De har blitt dokumentert gjennom å studere bruk i praksis, fremfor å bli «pønsket ut»
- Eksempler
  - Observer
  - Model – View – Controller (MVC)
  - Singleton
  - Iterator
  - Decorator

