DD1385
Programutvecklingsteknik
Några bilder till föreläsning 7

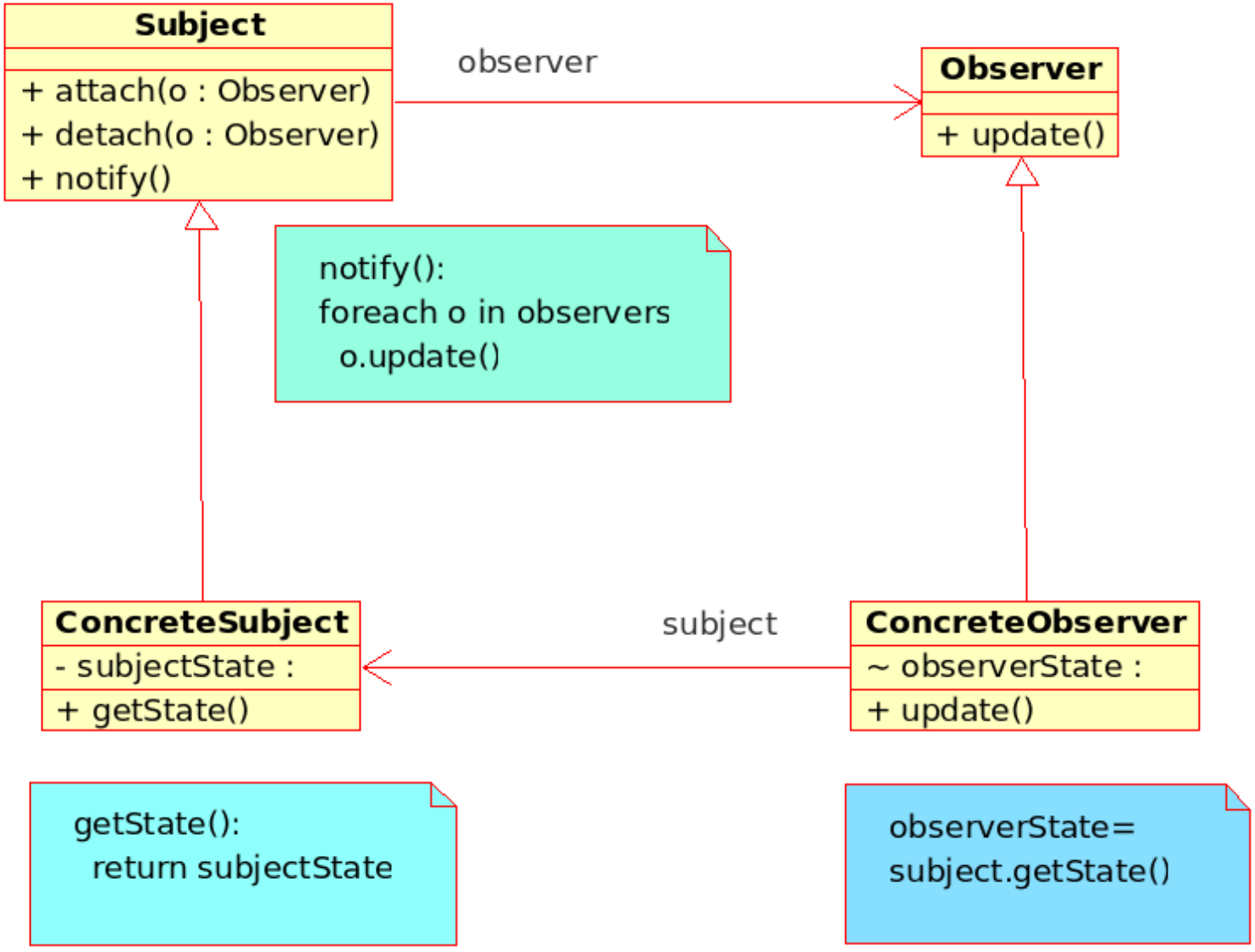
Innehåll

- Designmönstret Observer
- Observer i Java
- Decorator
- Mediator
- Facade
- State
- Uppräkninstyper-enum

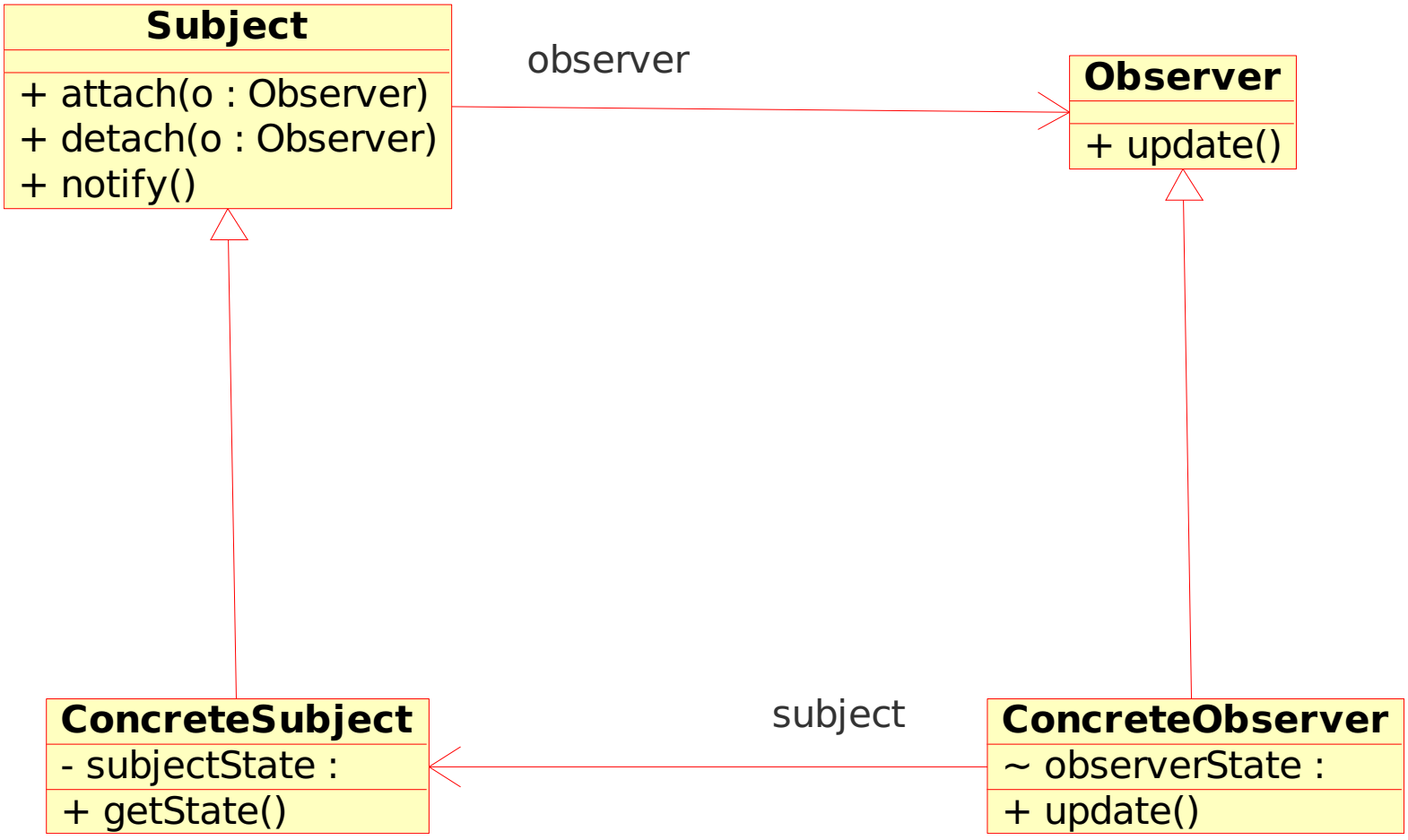
Mönstret Observer

- Ett system med många samarbetande och beroende klasser måste kunna upprätthålla konstistensen mellan objekten
- *Om B:s tillstånd beror av A så måste ändring av A $\rightarrow$ ändring av B*
- I mönstret ingår en/ett Subject och godtyckligt antal Observers
- Alla Observers underrättas när tillståndet i Subject ändras

Observer


Subject
+ attach(o : Observer)
+ detach(o : Observer)
+ notify()
observer
Observer
+ update()
notify():
foreach o in observers
o.update()
ConcreteSubject
- subjectState :
+ getState()
subject
ConcreteObserver
~ observerState :
+ update()
getState():
return subjectState
observerState=
subject.getState()

Observer


Subject
+ attach(o : Observer)
+ detach(o : Observer)
+ notify()
observer
Observer
+ update()
ConcreteSubject
- subjectState :
+ getState()
subject
ConcreteObserver
~ observerState :
+ update()

- De två översta klasserna *Subject* och *Observers* innehåller mönstrets egenskaper
- *Subject* och *Observers* kan programmeras oberoende av tillämpningen
- De två nedre klasserna *ConcreteSubject* och *ConcreteObserver* innehåller den aktuella tillämpningen

- *Subject* har en lista med sina *Observers*
- *attach(obs)* och *detach(obs)* ändrar den listan
- *notify* går igenom listan och uppdaterar alla *Observers*
- Alla *Observers* måste ha metoden *update*
- *Subject* och *Observer* finns alltid i mönstret
- *ConcreteSubject* och *ConcreteObserver* är den aktuella tillämpningen

Observer, forts

- I ett program kan ett objekt registreras som *Observer* för flera subjekt men en "mönsterinstans" innehåller bara ett *Subject*.

Exempel från Java-API:n

- *Subject* = grafisk komponent
- *Observer* = lyssnarobjekt
- En grafisk komponent kan ha flera lyssnare
- En lyssnare kan lyssna på flera grafiska komponenter

Observer-exempel: Modell med flera vyer

- Subject/ConcreteSubject är en modell
- Observers är objekt med olika bilder av data,
  - Tabell
  - Stapeldiagram
  - Tårtdiagram
- Nya data i modellen $\rightarrow$ `notify()` anropas och alla vyer/Observers uppdateras

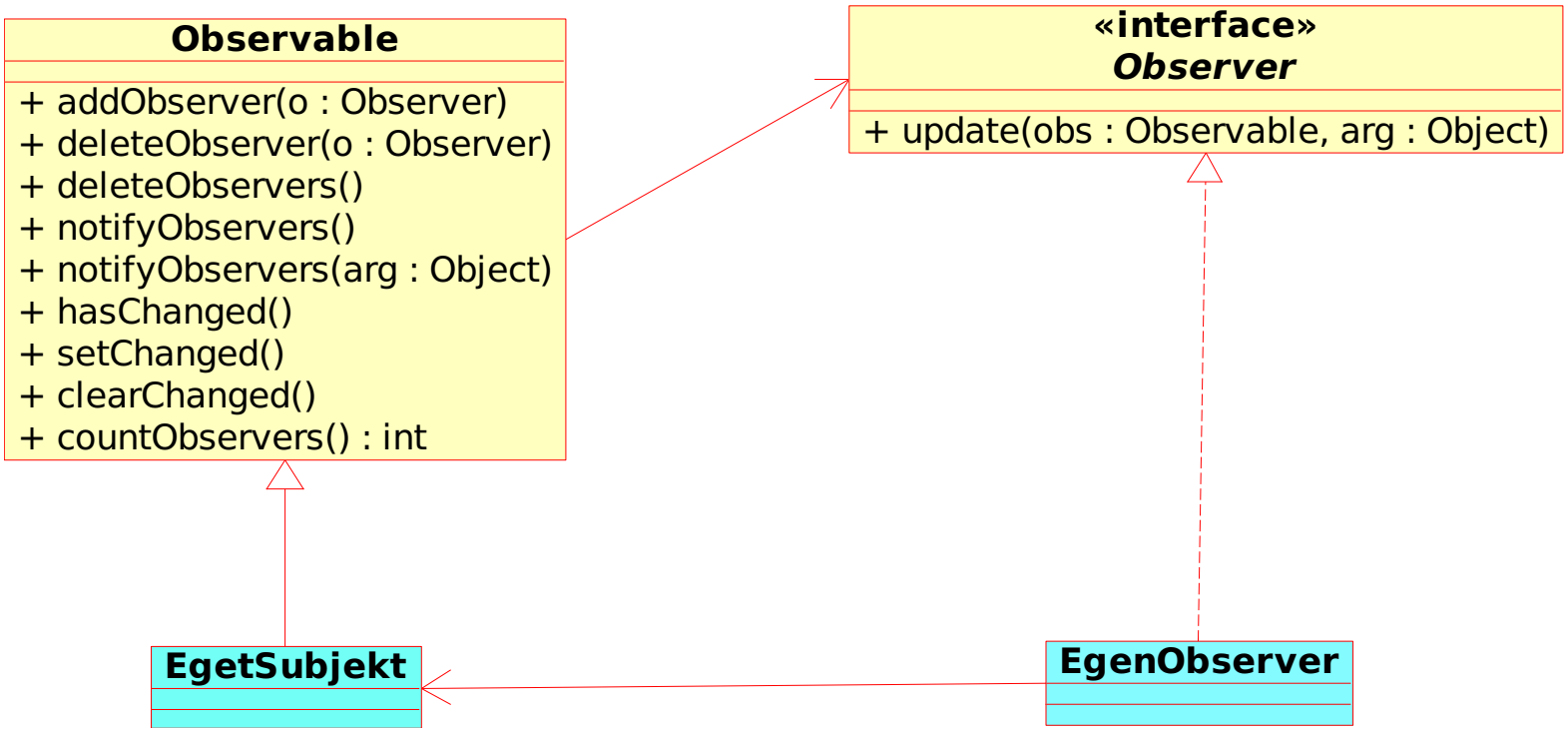
Observer - hjälp i Java-API:n

I paketet `java.util` finns

- Klassen `Observable` med 9 metoder
- Interfacet `Observer` med metoden `update`

Kan användas för egna Observer/Observable

Observable och Observer i Java-API:n



Övre klass och interface finns i biblioteket, paketet `java.util`

De två nedre klasserna skriver man själv.

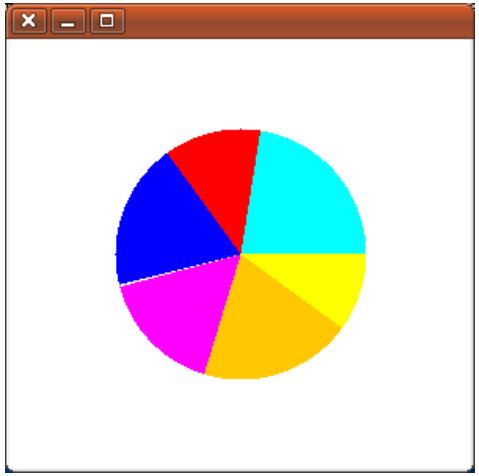
Rösträkningsexempel, demonstration av Observer

- `VotesModel` är vår Observable, lagrar antal avgivna röster.
- `TextView, PieView, BarView` är våra Observers, har var sin bild av modellen.
- `VotesInput` används för att påverka `VotesModel` men är inte ett kontrollobjekt enligt MVC

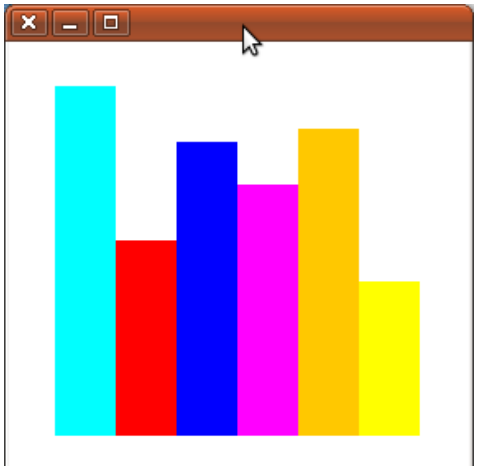
Exemplet demonstreras och läggs på kurshemsidan

På följande sidor finns det viktigaste ur varje klass


Neuhaus
Valrhona
Amedei
Charlemagne
Chocovic
Godiva




Neuhaus : 25
Valrhona : 14
Amedei : 21
Charlemagne : 18
Chocovic : 22
Godiva : 11



VotesModel

```
class VotesModel extends Observable {
  private String[] chocs={"Neuhaus","Valrhona",...
  private Color[] cols={Color.cyan,Color.red,...
  private int n = chocs.length;
  private int[] freqs = new int[n];

  void count(int i) {
    freqs[i]++;
    change();  //*** VotesModel was changed ***
  }
  void change() {
    setChanged();          // call methods from
    notifyObservers();     // Observable
  }
  // access methods not shown
}
```

VotesModel

- *VoteModel* ärver nio färdiga metoder från *Observable*, t.ex *addObserver()* och *notifyObservers()*
- *Observable* innehåller en datastruktur för att lagra listan med de *Observers* som anmäler sig. Strukturen ärvs men vi använder den bara implicit genom metodanropen.
- *notifyObservers()* kommer att att anropa *update* för alla observers. Konkret *update* skrivs av användaren. innehåller den aktuella tillämpningen
- Metoden setChanged() måste anropas före notifyObservers(), annars blir notifyObservers()

VotesInput

```
class VotesInput extends JFrame implements
                                    ActionListener {
  VotesModel mod;

  VotesInput (VotesModel m) {
    mod = m;
    // set LayoutManager, create buttons
    // connect ActionListener and
    // usual JFrame-stuff
  }

  public void actionPerformed (ActionEvent e){
    NumberButton b = (NumberButton) e.getSource();
    int m = b.number;
    mod.count(m); // Update model with new vote
  }
}
```

TextView

```
class TextView extends JFrame implements
                                        Observer {
  JTextArea ta = new JTextArea();
  TextView () {
     // initialize with size, Font etc.
  }

  // From interface Observer
  public void update(Observable obs, Object arg) {
    VotesModel votes = (VotesModel) obs;
    StringBuilder tex = new StringBuilder("\n");
    for (int i = 0; i<votes.getN(); i++)
      tex.append("␣"+ votes.getChoc(i)+ ...)

    ta.setText(tex.toString());
   }
}
```

PieView (BarView has the same structure)

```
class PieView extends JFrame implements Observer{
  VotesModel mod;
  Piepanel panel;

  PieView () {
    setSize(300,300);
    panel = new Piepanel();
    add(panel);
    setVisible(true);
  }

  // From interface Observer
  public void update(Observable obs, Object arg){
    this.mod = (VotesModel) obs;
    panel.repaint();
  }
  class Piepanel extends JPanel{...} //inner class
}
```

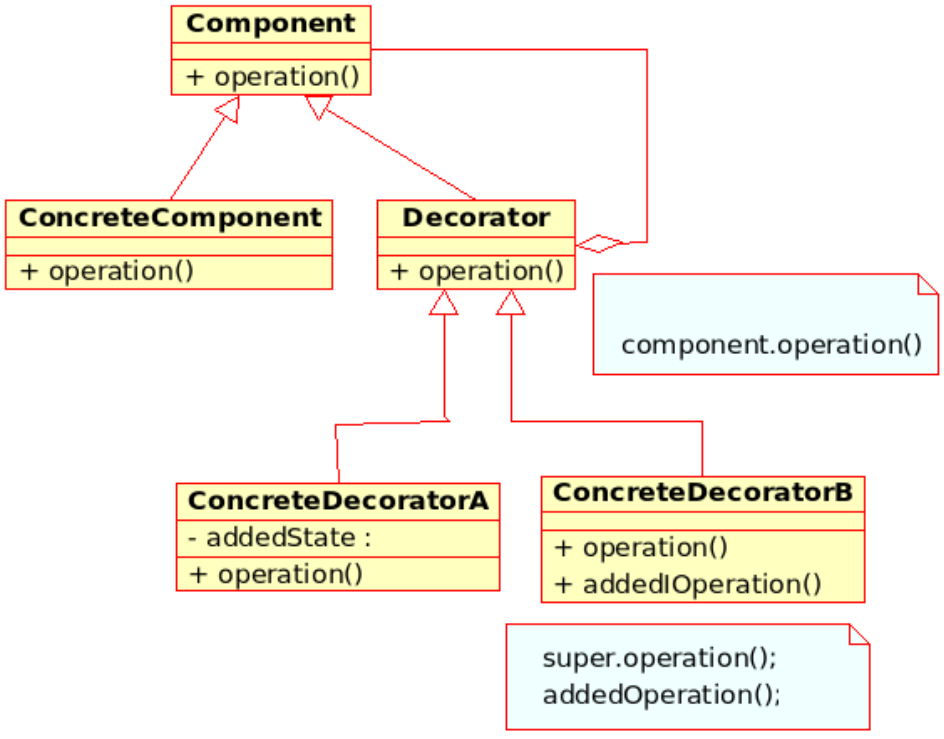
VotesDemo, puts it together

```
class VotesDemo {
  public static void main (String[] u) {
    VotesModel mod = new VotesModel();
    VotesInput vi = new VotesInput(mod);
    vi.setLocation(70,70);  // screen position

    TextView tv = new TextView();    tv.setLoc...
    mod.addObserver(tv);   //register as Observer

    PieView pv = new PieView();      pv.setLoc...
    mod.addObserver(pv);

    BarView bv = new BarView();      bv.setLoc...
    mod.addObserver(bv);

    mod.change(); // to show initial state
  }
}
```

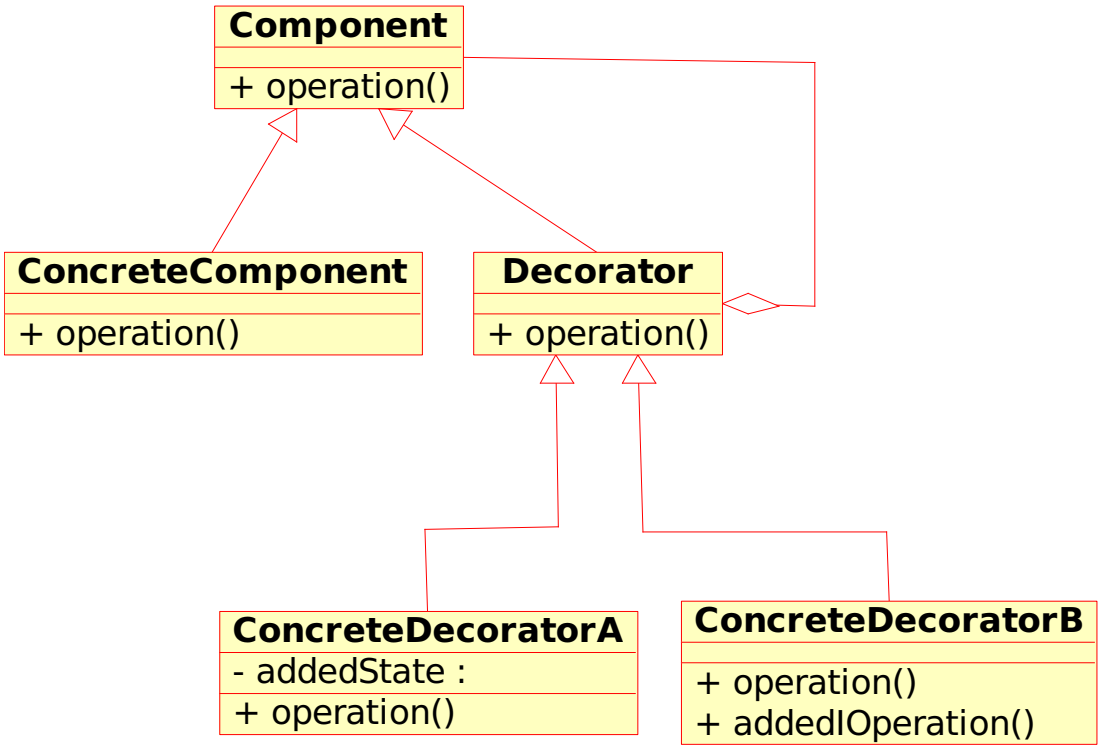
Mönstret Decorator

- Ansvar och kompetens läggs till ett objekt dynamiskt
- Flexibelt alternativ till (för många) subklasser
- En Decorator har referens till sin Component
- Javas klasser för strömmar använder Decorator
- Annat exempel: En klass för fönster som man vill utöka med kombinationer av HScroll, VScroll, Border m.m.

Decorator


Component
+ operation()
ConcreteComponent
+ operation()
Decorator
+ operation()
component.operation()
ConcreteDecoratorA
- addedState :
+ operation()
ConcreteDecoratorB
+ operation()
+ addedIOperation()
super.operation();
addedOperation();

Decorator


Component
+ operation()
ConcreteComponent
+ operation()
Decorator
+ operation()
ConcreteDecoratorA
- addedState :
+ operation()
ConcreteDecoratorB
+ operation()
+ addedIOperation()

Decorator-exempel

- `LibraryItem`
  motsvarar Componenent
- `Book, Audiobook`
  motsvarar ConcreteComponent
- `Decorator`
  motsvarar Decorator
- `Sellable, Borrowable`
  motsvarar ConcreteDecorator

```
abstract class LibraryItem {
    private int numCopies;

    public int getNumCopies () {
        return numCopies;
    }

    public void setNumCopies (int v) {
        numCopies = v;
    }

    public abstract void display();
}
```

```
class Book extends LibraryItem {
  private String author;
  private String title;

  public Book(String author,String title,int numC){
    this.author = author;
    this.title = title;
    setNumCopies(numC);
  }

  public void display() {
    System.out.println("\nBook ------- ");
    System.out.println(" Author: "+ author);
    System.out.println(" Title: "+ title);
    System.out.println
      (" # Copies: " + getNumCopies());
  }
}
```

```
class Audiobook extends LibraryItem {
  private String voice;
  private String title;
  private int playTime; // minutes

  public Audiobook( String voice, String title,
                    int numCopies, int playTime ) {

    this.voice = voice;
    this.title = title;
    setNumCopies (numCopies);
    this.playTime = playTime;
  }

  public void display() {
    // prints voice, title, numCopies, playTime
    // just like in Book
  }
}
```

```
abstract class Decorator extends LibraryItem {

  protected LibraryItem libraryItem;


  public Decorator (LibraryItem libraryItem) {
    this.libraryItem = libraryItem;
  }

  public int getNumCopies() {
    return libraryItem.getNumCopies();
  }

  public void setNumCopies(int c) {
    libraryItem.setNumCopies(c);
  }
}
```

```
class Sellable extends Decorator{

  protected int price;
  protected int numberSold;

  public Sellable (LibraryItem libraryItem ,
                   int price){
    super( libraryItem ) ;
    this.price = price;
  }

  public void sellItem() {
    libraryItem.setNumCopies
                (libraryItem.getNumCopies()-1);
    numberSold++;
  }

  public void display() {...} // *** next page ***
```

```
class Sellable extends Decorator{

   // as previous page

   public void display() {
      libraryItem.display();   // *** important ***

      System.out.println("␣#␣Sold:␣"+numberSold);
      System.out.println("␣Price:␣"+price+"␣Euro");
   }
}
```

```
libraryItem.display()
```

skriver ut info om den komponent som dekoreras av

```
Sellable
```

```
import java.util.*;
class Borrowable extends Decorator{
  protected ArrayList<String> borrowers =
                                  new ArrayList<String>();

  public Borrowable(LibraryItem libraryItem){
    super(libraryItem);}

  public void borrowItem(String name){
    borrowers.add(name);
    libraryItem.setNumCopies
                    (libraryItem.getNumCopies()-1);}

  public void returnItem(String name){
    borrowers.remove(name);
    libraryItem.setNumCopies
                    (libraryItem.getNumCopies()+1);}

  public void display() { *** see next page ***
```

```
class Borrowable extends Decorator{

  // as previous page

  public void display(){
    libraryItem.display();    // *** ***
    for (String borrower:borrowers)
      System.out.println("␣borrower:␣" + borrower);
  }
}
```

```
libraryItem.display()
```

skriver ut info om den komponent som dekoreras av

Borrowable

```
public class DecoratorApp {
  public static void main( String[] args ) {

    // Create book and audiobook and display

    Book book = new Book
      ("Lauri Lebo", "The Devil ...",10);

    Audiobook audiobook = new Audiobook
     ("Stephen Fry",
      "The Complete Harry Potter Collection", 23, 7

    book.display();
    audiobook.display();

    // *** to be continued ***
```

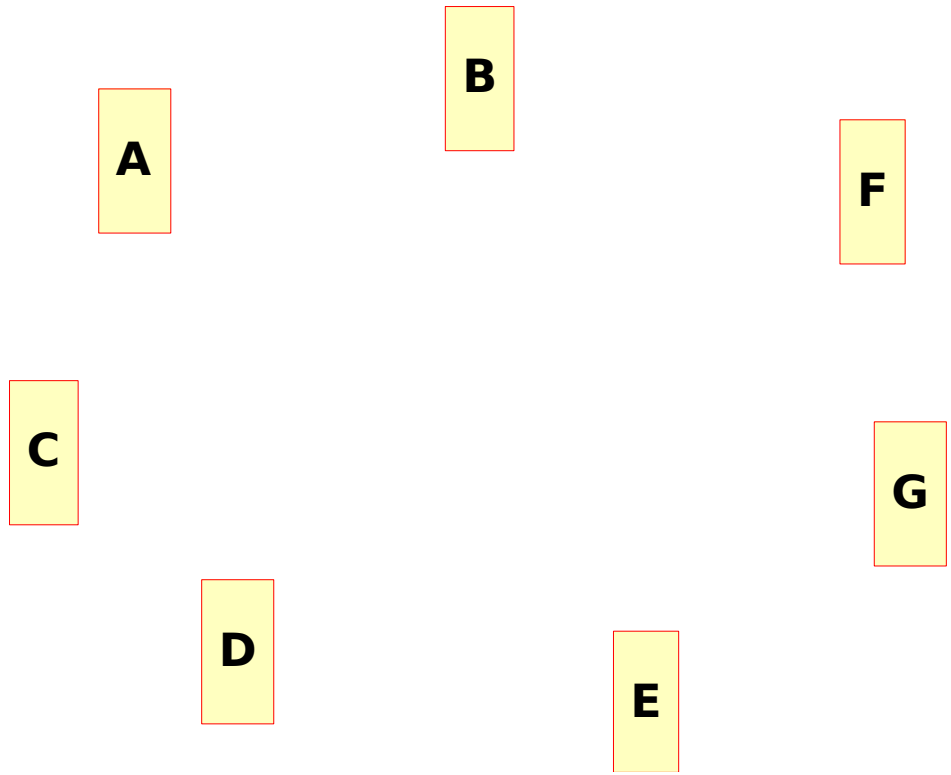
```
// Make book sellable, display, sell, display
System.out.println("\nBook made sellable");

Sellable sBook = new Sellable(book, 13);
sBook.display();
sBook.sellItem();
sBook.display();

// Make audiobook borrowable, borrow and display
System.out.println
  ( "\nAudiobook made borrowable:");

Borrowable bAudiobook=new Borrowable(audiobook);
bAudiobook.borrowItem("Petter Al");
bAudiobook.borrowItem("Elsie Ek");

bAudiobook.display();
```

```
    // Make audiobook also sellable,
    // sell and display

    System.out.println
      ("\nAudiobook made also sellable:");

    Sellable sbAudiobook = new Sellable(bAudiobook, 1
    sbAudiobook.sellItem(); sbAudiobook.sellItem();
    sbAudiobook.sellItem();
    sbAudiobook.display();
}
```

Mediator

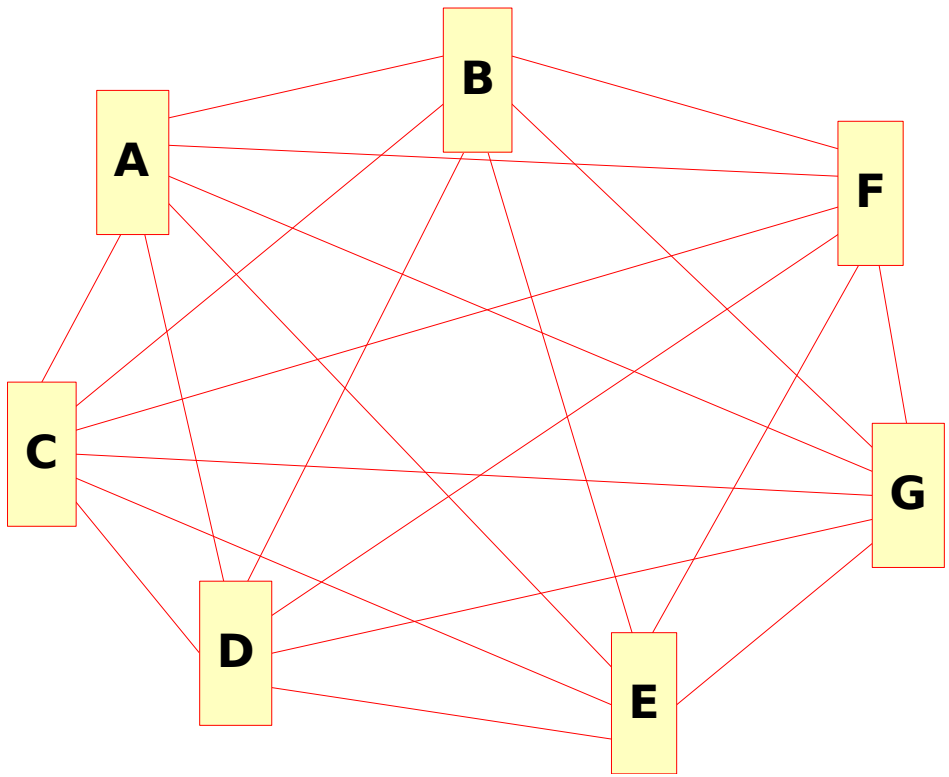
Objekt ska kommunicera *ibland*

– alla med alla!



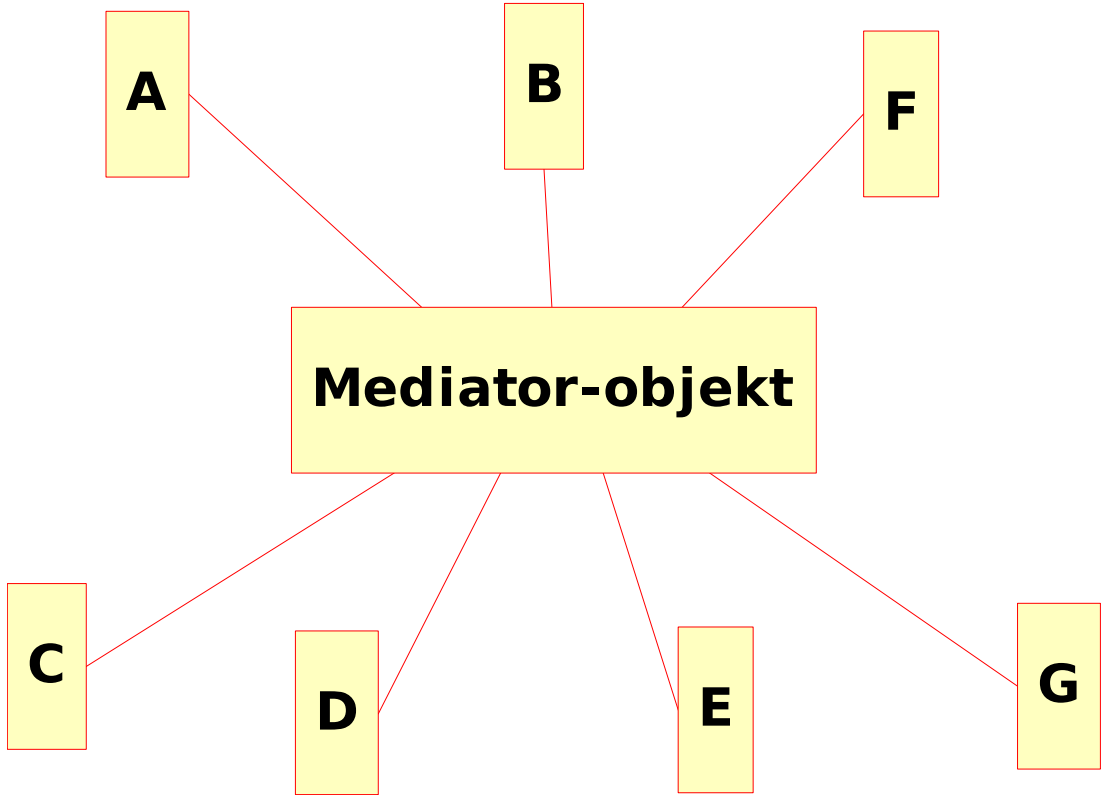
Detta är objekt, EJ klasser

Var och en får en referens till alla andra – ingen bra lösning!



Detta är objekt, EJ klasser

Bättre med ett kommunikationsobjekt i mitten



Objekt, EJ klasser

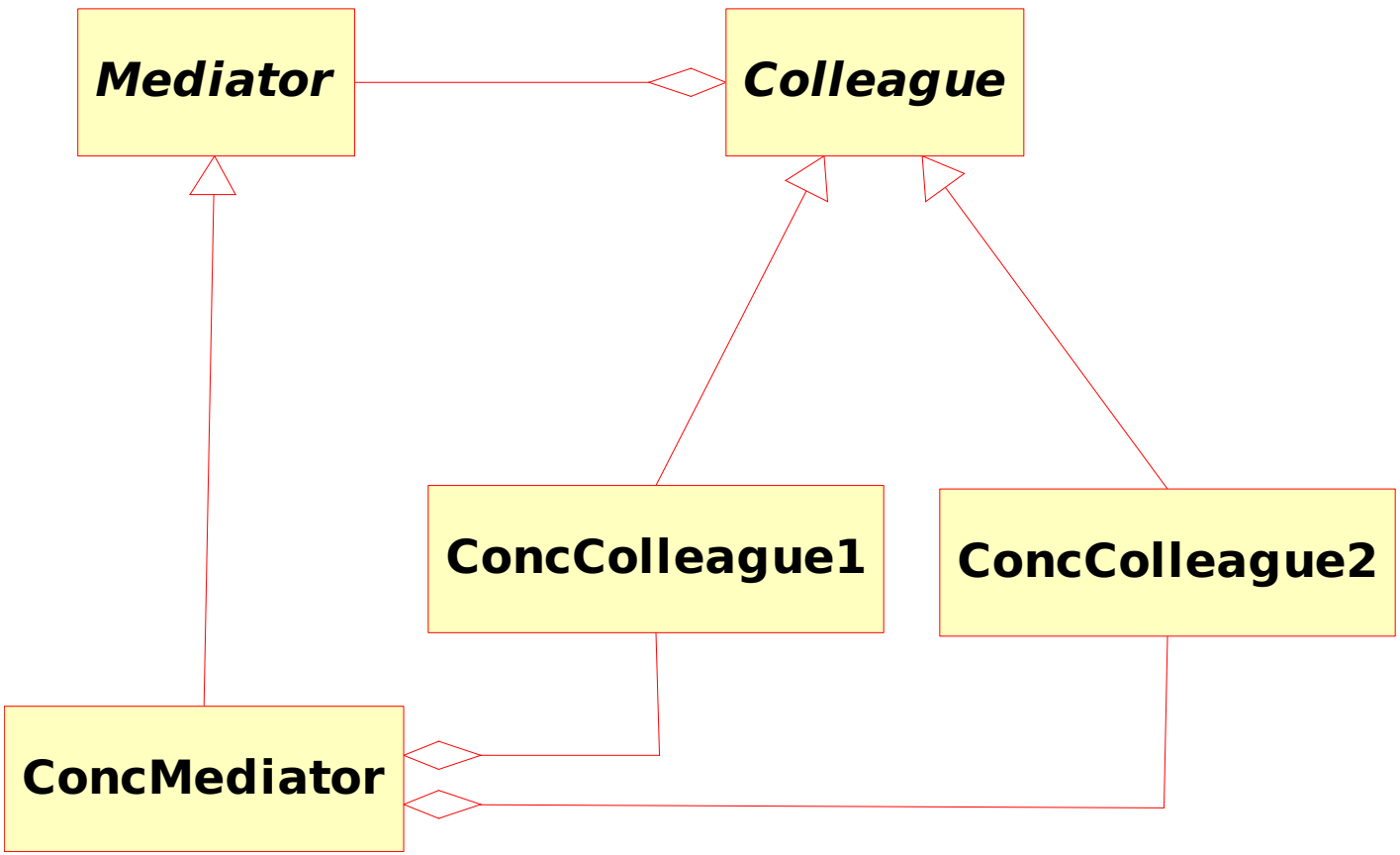
Mönstret Mediator

- Två eller flera objekt interagerar *ibland*
- Objekten ska hållas oberoende av varandra (lös koppling)

Gör så här:

- Klassernas kommunikation läggs i en *Mediator* - klass
- *Mediator* har referenser till objekten den kontrollerar
- De interagerande klasserna har referens till sin (abstrakta) *Mediator*

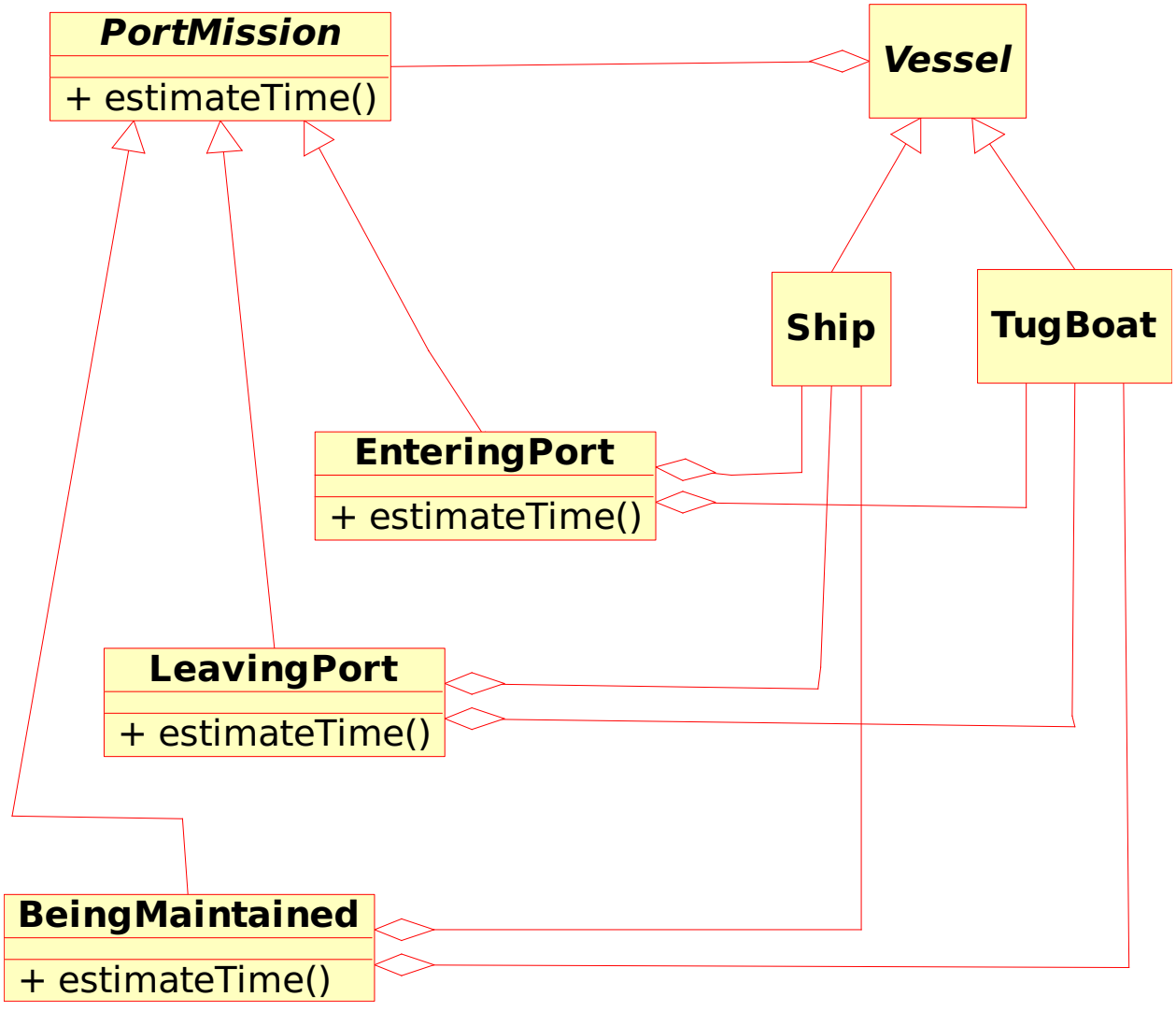
Mediator


Mediator
Colleague
ConcColleague1
ConcColleague2
ConcMediator

Kommentarer till Meditator

- Association kan användas istället för aggregat
- Viktigt att `ConcColleague1` och `ConcColleague2` endast har referens till den *abstrakta Mediator*.
- Varje ”kollega” anropar sin *Mediator* som skickar vidare meddelandet till annan ”kollega”
- Mediator-klasser är sällan återanvändbara
- Typexempel: Chattprogram
- Exempel från bok av *Eric Braude*: Harbour Application

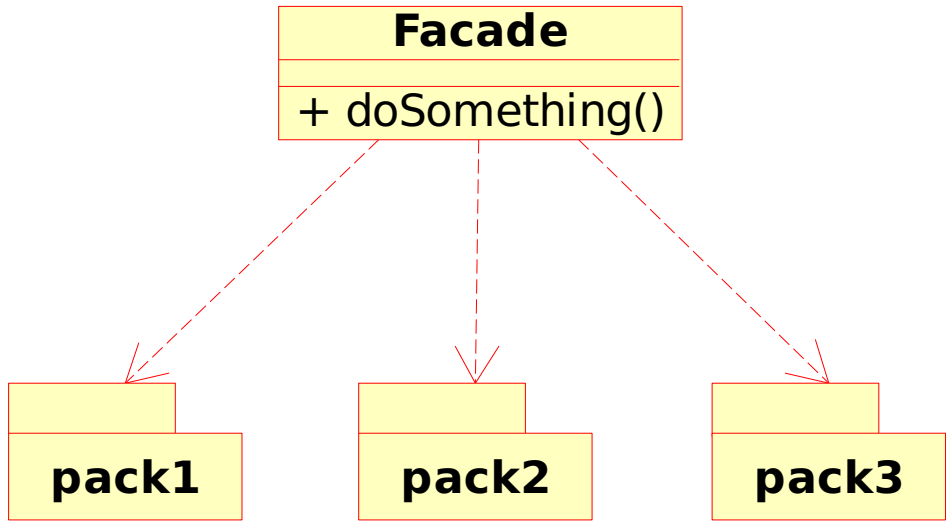
Mediator Harbour Application



Mönstret Facade

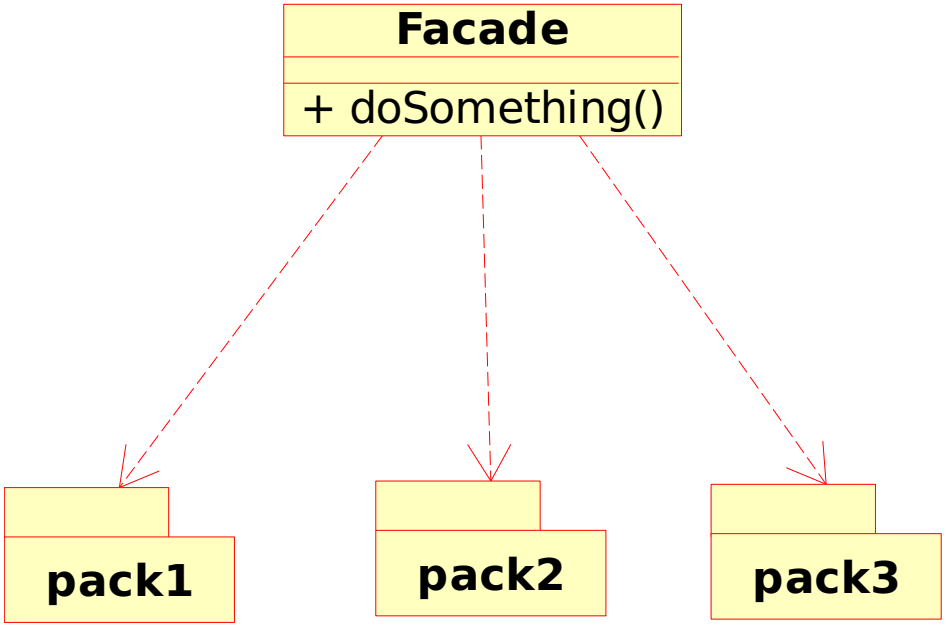
En enda klass representerar ett stort system

- Förenklar för användaren
- Lösare koppling mellan användaren och bibliotek
- Döljer klumpig uppbyggnad av bibliotek



Pattern Facade

Simplifies a clients interface



Facade – miniexempel

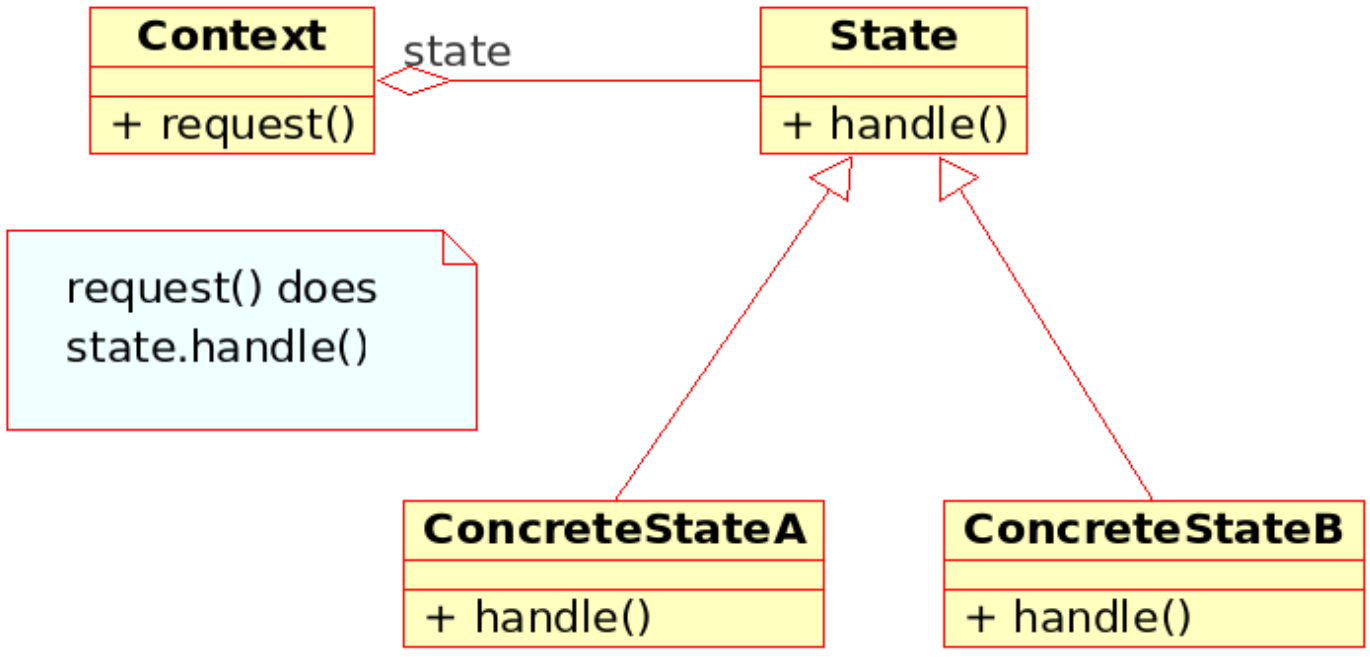
```
import pack1.*;
import pack2.*;
class Facade {
    Class1 class1 = new Class1("@XTU", 1729);
    Class2 class2 = new Class2( 'W', 362);

    void doSomething(int k) {
        class1.method1(k, 3*k, "HHW");
        class2.method2(1.0/k, "0110");
    }
}
```

Det nya enkla gränssnittet utgörs av metoden `doSomething()` med en enkel parameter.

Mönstret State

- Ett objekts beteende ändras när dess tillstånd ändras
- Istället för if-satser som testar en tillståndsvariabel:
  Låt ett objekt ta hand om tillståndet.
- När tillståndet ändras, byt aktuellt tillståndsobjekt.

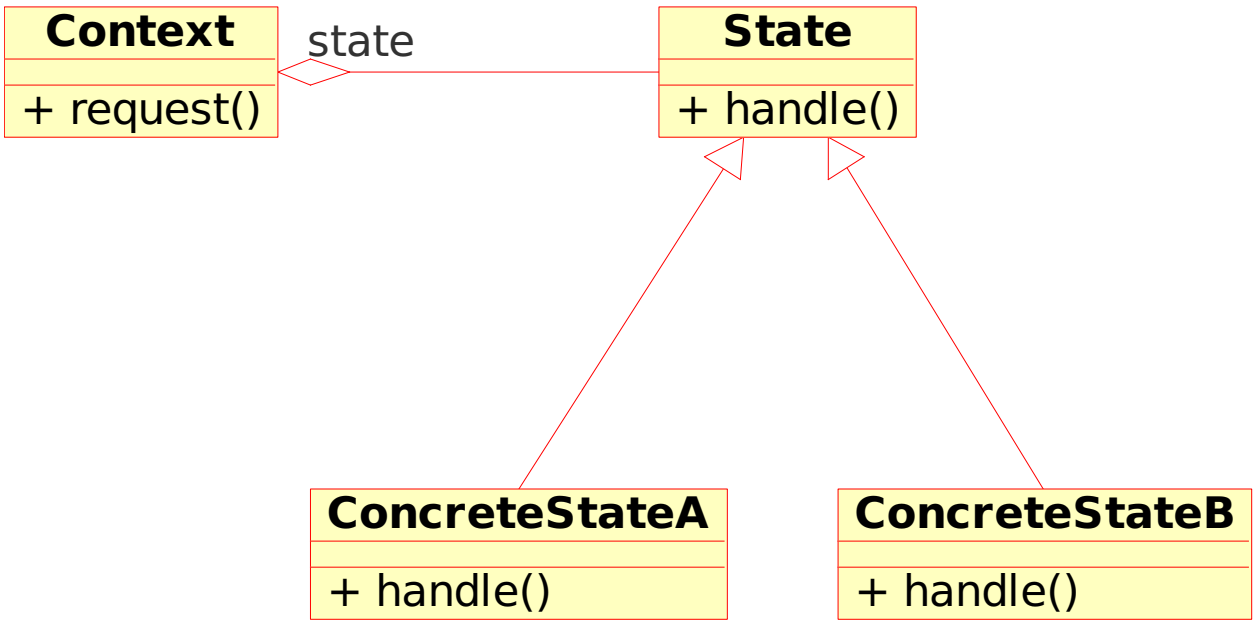
State


Context
+ request()
state
State
+ handle()
request() does
state.handle()
ConcreteStateA
+ handle()
ConcreteStateB
+ handle()

```
class Game {

  int level;

  void move1() {
    if (level == 1) {...}       // simple action
    else if (level == 2) {...}// med. level action
    else if (level == 3) {...}// advanced action
  }

  void move2() {
    if (level == 1) {...}       // simple action
    else if (level == 2) {...}// med. level action
    else if (level == 3) {...}// advanced action
  }

}
```

```
class Game {
  int level;
  State currentState = new Level1State(this);

  void move1() {
    currentState.move1();
  }

  void move2() {
    currentState.move2();
  }

  void checkstate() {
    // if state change is required, change the
    // state object. If-clauses needed here!

      currentState = ... //change the state object
  }
}
```

State


Context
+ request()
state
State
+ handle()
ConcreteStateA
+ handle()
ConcreteStateB
+ handle()

Uppräkningstyper – enum

Definiera egen typ med *egna* värden:

```
enum Day {Monday, Tuesday, Wednesday,
          Thursday, Friday, Saturday,
          Sunday}
```

Day är en klass med konstanta värden.
Värdena refereras med

```
Day.Wednesday      Day.Saturday
```

```
Day theDay = Day.Saturday;
```

```
Day[] specialdays =
{Day.Tuesday, Day.Thursday, Day.Sunday};
```

Uppräkningstyper – enum

Alla värden inom typen:

```
for (Day d: Day.values())
    System.out.println(d);
```

ger utskrift

```
Monday
Tuesday
Wednesday
Thursday
...
```

Uppräkningstyper – enum

Ordningsnummer med ordinal():

```
for (Day d: Day.values())
    System.out.print(d.ordinal() + " ");
```

ger utskrift

```
0 1 2 3 4 5 6
```

Går att jämföra: `implements Comparable`

```
System.out.println
  (Day.Thursday.compareTo(Day.Saturday));
```

ger utskrift

```
-2
```

Spelkort med enum

```
class Spelkort {
    Farg farg;
    Valor valor;

    Spelkort(Farg f, Valor v){
        farg = f;
        valor = v;
    }

    public String toString () {
        return farg + "-" + valor;
    }
}
```

Kortlek

```
class Kortlek {
  Spelkort[] lek = new Spelkort[52];

  Kortlek() {
    int i=0;
    for (Farg farg: Farg.values())
      for (Valor valor:Valor.values())
        lek[i++] =
          new Spelkort(farg,valor);
  }
}
```

Metoder att lägga till i Kortlek : toString()

En lång String med 4 kort per rad

```
public String toString() {
    StringBuilder allt = new StringBuilder();
    int rad = 0;
    for (Spelkort s : lek){
        allt = allt.append(s + " ");
        if (rad++ == 4){
            allt.append("\n");
            rad = 0;
         }
    }
    allt.append("\n");
    return allt.toString();
}
```

Blanda korten

`Arrays.asList(kortlek)`

gör lista intimt kopplad till en array (här kortlek).

Ändringar i listan görs även i arrayen.

```
Collections.shuffle(Arrays.asList(kortlek));
```

Listan blandas $\rightarrow$ Kortleken blandas

Metoder att lägga till i Kortlek : blanda()

```
void blanda() {
    Collections.shuffle(Arrays.asList(lek));
}
```

Testa kortleken: skapa, skriv ut, blanda, skriv ut

```
public static void main(String[] u) {
    Kortlek lek = new Kortlek();
    System.out.println("Kortleken fran borjan:");
    System.out.println(lek);
    lek.blanda();
    System.out.println("\nKortleken blandad:");
    System.out.println(lek);
}
```

Gamla Spelkort

```
class Spelkort {
    String farg;
    int valor;

    Spelkort(String f, int v){
        farg = f;
        valor = v;
    }

    public String toString () {
        return farg + "-" + valor;
    }
}
```

Varför är nya Spelkort bättre än gamla ?

- Med enum så är Farg och Valor väldefinierade
- Största och minsta värden är säkert tillgängliga
  `values()` `ordinal()`
- Med gamla Spelkort kan man skapa t.ex.
  `new Spelkort("VIRUS",-1729)`

Enum är mer än enstaka värden

- Enum är klass med konstruktor och metoder (om man vill)
- Enum - klasser är `final`, går ej att ärva från
- Enum med ett enda värde $\rightarrow$ `Singleton`

```
public enum Singleton {THEONLY}
```

Singleton med lite innehåll

```
public enum Snowwhite {
  THEONLY;

  boolean married = false;
  String occupation;
  int age;

  void incrAge () {
    age++;
  }

  public String toString() {
    return    "Snowwhite " + age;
  }
}
```

Sju dvärgar

```
public enum Dwarf {
    Blick(131), Flick(112), Glick(142),
    Snick(115), Plick(108), Whick(120), Quee(99);

    int age;

    Dwarf (int a) {
        age = a;
    }

    public String toString() {
        return super.toString() + " " + age;
    }

    // fler metoder
}
```

Varje dvärg (konstant instans av klassen) definieras med ett namn och ett kontruktoranrop

```
Plick(108)    //age=108 is set in constructor
```

Sju dvärgar, mainmetod

```
public static void main (String[] u) {
    System.out.println("All dwarfes");
    for (Dwarf d : Dwarf.values())
        System.out.print(d + "  ");
    System.out.println();
}
```