

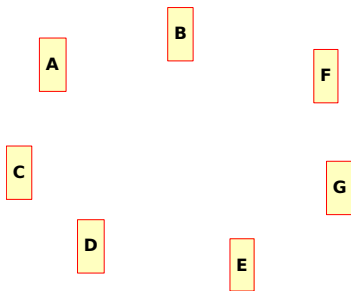
DD2385
Programutvecklingsteknik
Några bilder till föreläsning 10
8/5 2013

Innehåll

- Designmönster:
 - Mediator
 - Facade
 - State
 - Decorator
- Enum m.m.

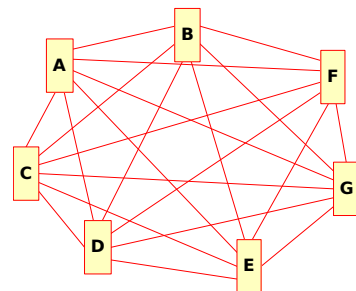
Mediator

Objekt ska kommunicera *ibland*
– alla med alla!



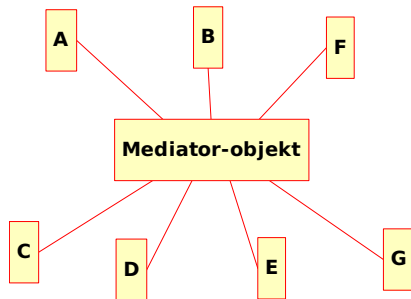
Detta är objekt, **EJ** klasser

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

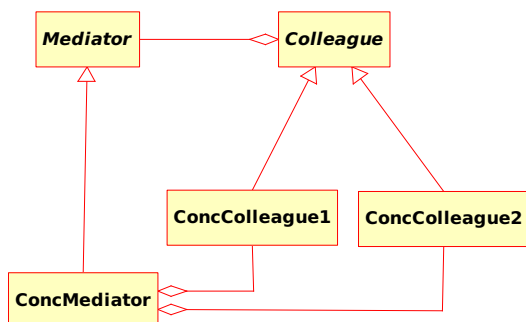
Mönstret Mediator

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

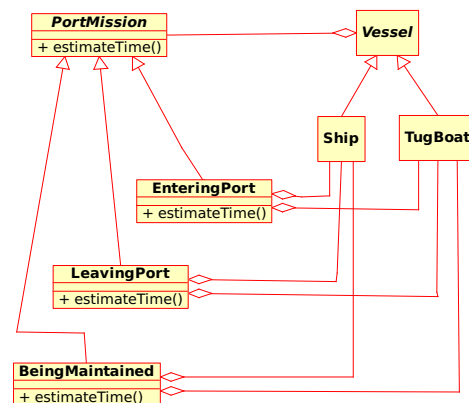
Gör så här:

- ▶ 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 Harbour Application



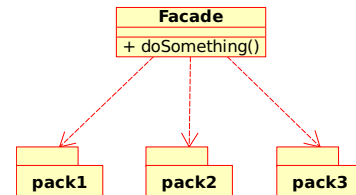
Kommentarer till Mediator

- ▶ 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
- ▶ Exempel: Lyssnare och grafiska gränssnitt. Hur?

Mönstret Facade

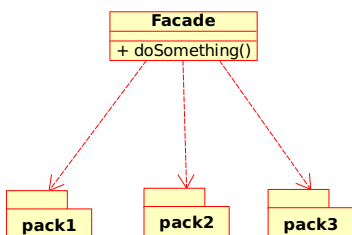
En enda klass representerar ett stort system

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



Pattern Facade

Simplifies a clients interface



Facade – minisexempel

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

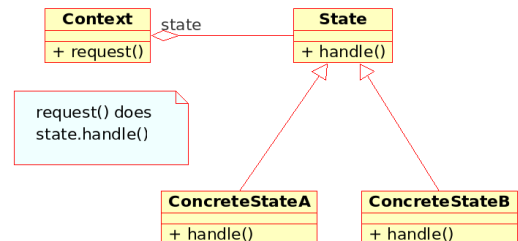
    void doSomething(int k) {
        class1.method1(k, 3*k, "HHW");
        class2.method2(1.0/k, "0110");
    }
}
```

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

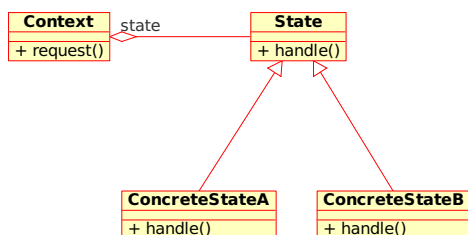
Mönstret State

- ▶ 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



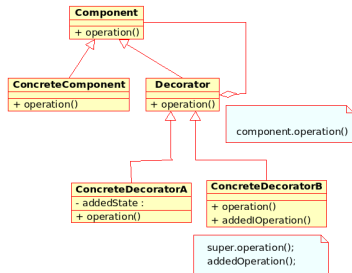
State



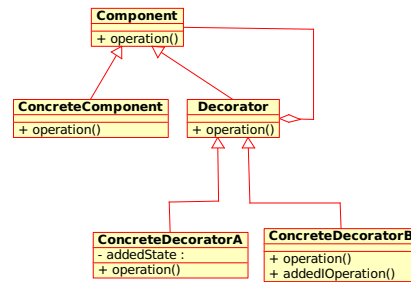
Mönstret Decorator

- ▶ 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



Decorator



Decorator-exempel

- ▶ **LibraryItem**
motsvarar Component
- ▶ **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

```

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

```

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 ***
}

```

```

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, 7500);

        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, 18);
sbAudiobook.sellItem(); sbAudiobook.sellItem();
sbAudiobook.sellItem();
sbAudiobook.display();
}
```

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();  
}
```

Metoder att lägga till i Kortlek : blanda()

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 → Kortleken blandas

```
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 från början:");  
    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 konstruktör och metoder (om man vill)
- ▶ **Enum** - klasser är **final**, går ej att ärva från
- ▶ **Enum** med ett enda värde → **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  
}
```

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();  
}
```