DD2385

Programutvecklingsteknik

Några bilder till föreläsning 10

6/5 2014

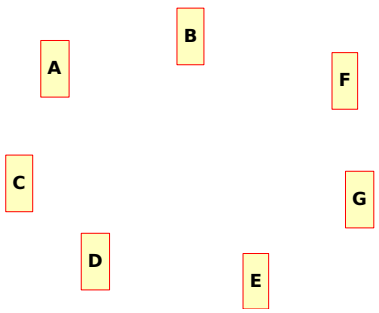
Innehåll

- Designmönster:
  - Mediator
  - Facade
  - State
  - Decorator

Mediator

Objekt ska kommunicera *ibland*

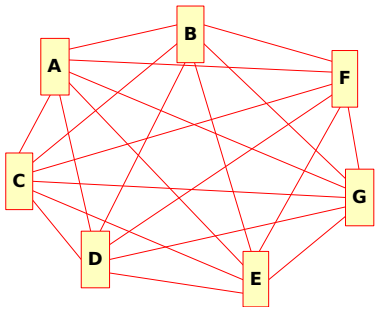
– alla med alla!



Detta är objekt, EJ klasser

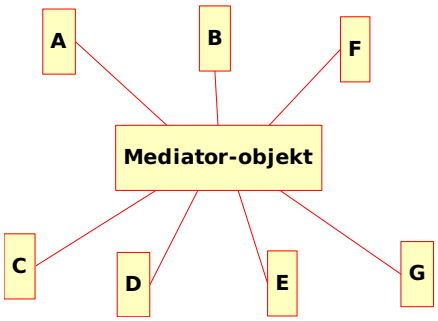
Var och en får en referens till alla andra

– ingen bra lösning!



Detta är objekt, EJ klasser

Bättre med ett kommunikationsobjekt i mitten



Objekt, EJ klasser

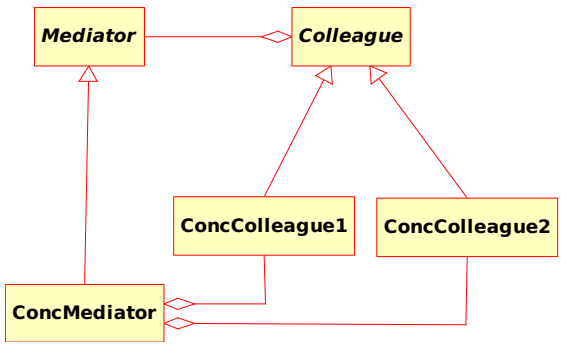
Mönstret Mediator

- 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*

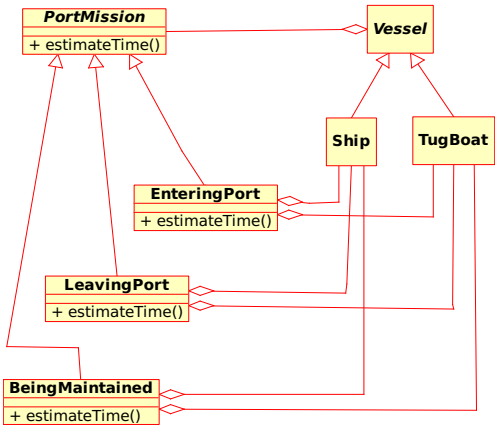
Mediator



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

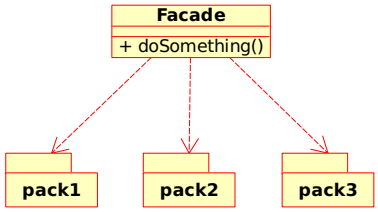
Mediator Harbour Application



Mönstret Facade

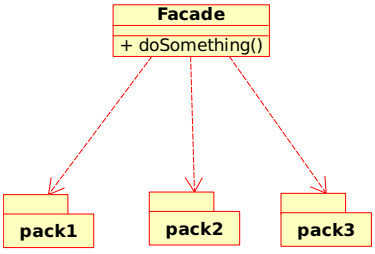
En enda klass representerar ett stort system

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



Pattern Facade

Simplifies a clients interface



Facade – miniexempel

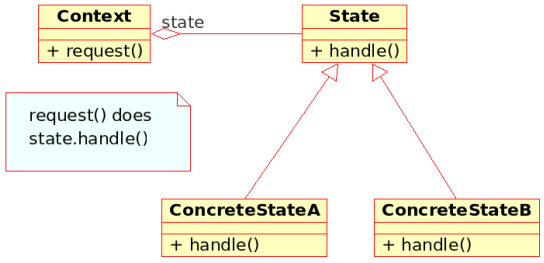
```
import pack1.*;
import pack2.*;
class Facade {
   Class1 class1 = new Class1("@XTU", 1729);
   Class2 class2 = new Class2( 'W', 362);

   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.

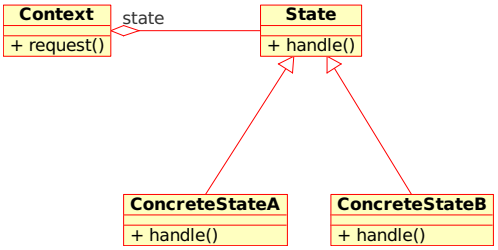
State




```
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
  }
}
```

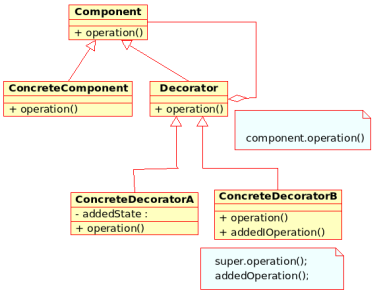
State



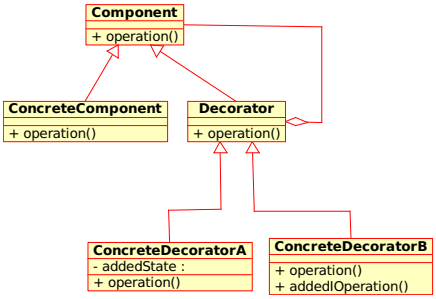
Mönstret Decorator

- 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 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, 75

    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();
  }
```