Advanced Object-Oriented Design
Lecture 12

Design patterns
Part III

Bartosz Walter
<Bartek.Walter@man.poznan.pl>

Design patterns

Catalog of Design Patterns

Decorator: Intent

- **Attach additional responsibilities to an object dynamically**
- Provide a flexible alternative to subclassing for extending functionality

by the Gang of Four

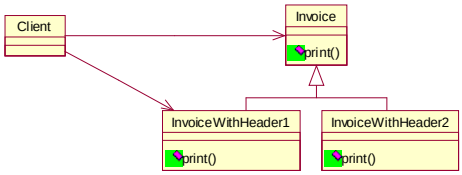
Decorator: Problem

```
public class Invoice {
   String buyer = null;
   String issuer = null;
   List <ListItem> elements = new ArrayList<ListItem>();
   Header header = null;

   private boolean useHeader() {
      return header != null;
   }

   public void print() {
      if (useHeader()) {
          header.print();
      }

      print("Issuer: " + issuer);
      print("Buyer: " + buyer);

      for (e : elements) {
         e.print();
      }
   }
}
```

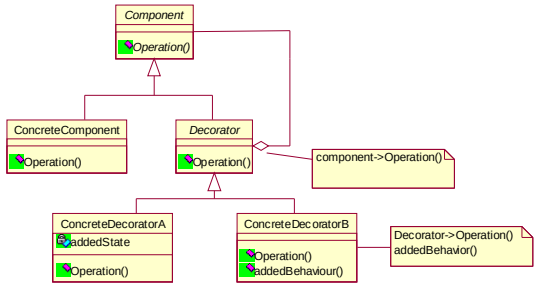
Decorator: Problem

```
public class Invoice {
   String buyer = null;
   String issuer = null;
   List <ListItem> elements = new ArrayList<ListItem>();

   public void print() {
      print("Issuer: " + issuer);
      print("Buyer: " + buyer);
      for (e : elements) {
         e.print();
      }
   }
}
public class HeaderInvoice extends Invoice {
   Header header = null;

   private boolean useHeader() {
      return header != null;
   }
   public void print() {
      if (useHeader()) {
          header.print();
      }
      super.print();
   }
}
```

Decorator: Problem



Decorator: Structure



by the Gang of Four

Decorator: Participants

- **Component**
  declares an interface for objects that may be assigned additional responsibilities
- **Concrete Component**
  implements the Component interface
- **Decorator**
  declares an interface that conforms to Component's interface
  is aware of the decorated Component (which is either a Concrete Component or another Decorator)
- **Concrete Decorator**
  adds responsibilities to the component

by the Gang of Four

Decorator: Consequences

- **more flexible than static inheritance**
  - responsibilities can be added at run-time
  - less complex class hierarchy
- **pay-as-you-go**
  - features are added whenever needed
- **object identities differ**
  - object identity is different from *Decorator's* identity
  - object identity should be encapsulated and hidden from client
- **lot of small, well-defined decorating classes**
- **improved testability**

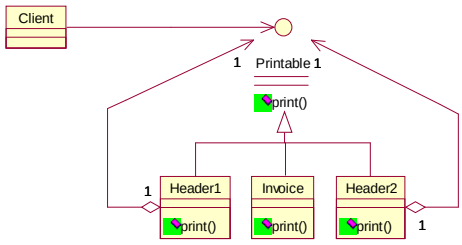
by the Gang of Four

Decorator: Implementation

- **Conformance of interfaces**
  Decorator and *Component* must implement same interface
- **Default *Decorator***
- **Keeping *Component* lightweight**
  the data storage should be deferred to subclasses
- ***Strategy* vs. *Decorator***
  Strategy deals with changing the internals
  Decorator deals with changing the skin

by the Gang of Four

Decorator: Example

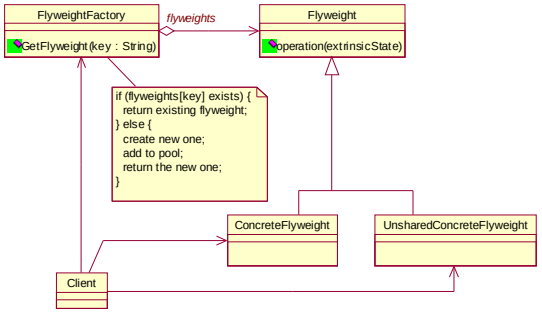


Flyweight: Intent

- Use sharing to **support large numbers of fine-grained objects efficiently**
- **Separate intrinsic (shared) and extrinsic (unique) object state into separate objects**

by the Gang of Four

Flyweight: Structure



by the Gang of Four

Flyweight: Participants

- **Flyweight**
  - declares an interface through which flyweights can receive and utilize the extrinsic state
- **Concrete Flyweight**
  - adds storage for an intrinsic state
  - must be independent of its context (extrinsic state)
- **Unshared Concrete Flyweight**
  - non-sharable flyweight
- **Flyweight Factory**
  - creates and manages flyweight objects
  - ensures that flyweights are shared properly
- **Client**
  - acquires *Flyweight* instances through *FlyweightFactory*

Flyweight: Consequences

- **growing space savings**
  - reduction of total number of instances
  - reduction of intrinsic state
  - extrinsic state may be computed or stored
- **computational costs**
  - managing the extrinsic state

by the Gang of Four

Flyweight: Example

```
public abstract class TeaOrder { // Flyweight
   public abstract void serveTea(TeaOrderContext teaOrderContext);
}

public class TeaFlavor extends TeaOrder {  // ConcreteFlyweight
   String teaFlavor;   // intrinsic (shared) state

   TeaFlavor(String teaFlavor) {
      this.teaFlavor = teaFlavor;
   }

   public String getFlavor() {
      return this.teaFlavor;
   }

   // teaOrderContext is an extrinsic (unique) state
   public void serveTea(TeaOrderContext teaOrderContext) {
      System.out.println("Serving tea flavor " + teaFlavor
         + " to table number " + teaOrderContext.getTable());
   }
}
```

by Lawrence Trurett

Flyweight: Example

```
public class TeaOrderCtx {  // context provides extrinsic state
   int tableNumber;

   TeaOrderContext(int tableNumber) {
      this.tableNumber = tableNumber;
   }

   public int getTable() {
      return this.tableNumber;
   }
}
```

by Lawrence Trurett

Flyweight: Example

```
public class TeaFlavorFactory { // Flyweight Factory
   TeaFlavor[] flavors = new TeaFlavor[10];// <10 flavors can be made
   int teasMade = 0;

   // Factory Method
   public TeaFlavor getTeaFlavor(String flavorToGet) {
      for (int i = 0; i < teasMade; i++) {
         if (flavorToGet.equals((flavors[i]).getFlavor())) {
            return flavors[i];
         }
      }
      flavors[teasMade] = new TeaFlavor(flavorToGet);
      return flavors[teasMade++];
   }

   public int getTotalTeaFlavorsMade() {
     return teasMade;
   }
}
```

by Lawrence Trurett

Flyweight: Example

```
class TestFlyweight {
   static TeaFlavor[] flavors = new TeaFlavor[100]; // orders
   static TeaOrderCtx[] tables = new TeaOrderCtx[100]; // tables
   static int ordersMade = 0;
   static TeaFlavorFactory teaFlavorFactory;

   static void takeOrders(String flavorIn, int table) {
      flavors[ordersMade] = teaFlavorFactory.getTeaFlavor(flavorIn);
      tables[ordersMade++] = new TeaOrderCtx(table);
   }

   public static void main(String[] args) {
      teaFlavorFactory = new TeaFlavorFactory();
      takeOrders("chai", 2);
      takeOrders("chai", 2);
      takeOrders("camomile", 1);
      takeOrders("earl grey", 1);
      takeOrders("camomile", 897);
      for (int i = 0; i < ordersMade; i++) {
         flavors[i].serveTea(tables[i]);
      }
      System.out.println(" ");
      System.out.println("total teaFlavor objects made: "
         + teaFlavorFactory.getTotalTeaFlavorsMade());
      }
   }
}
```

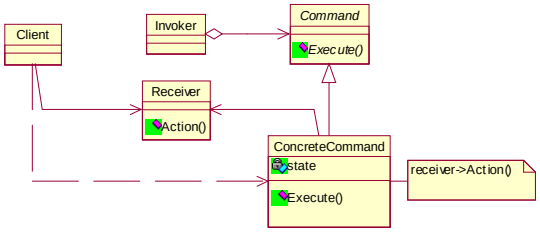
by Lawrence Trurett

Command: Intent

- **Encapsulate a request as an object**
- Allow **parametrizing clients with different requests**
- Support **undoable operations**

by the Gang of Four

Command: Structure



by the Gang of Four

Command: Participants

- **Command**
  - declares an interface for executing an operation
- **ConcreteCommand**
  - defines a binding between a *Receiver* and an action
  - implements *execute()* method
- **Client**
  - creates a *ConcreteCommand* object and sets its *Receiver*
- **Invoker**
  - asks the *Command* to carry out the request
- **Receiver**
  - knows how to perform the concrete operations

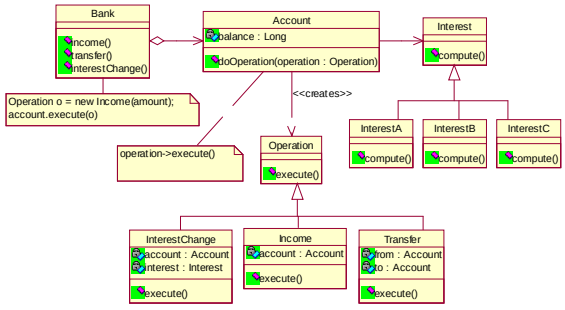
by the Gang of Four

Command: Consequences

- **decoupling the sender from *Receiver***
- ***Commands* can be manipulated and extended like any other object**
- ***Commands* can be assembled into a composite command**
- **adding new *Commands* is easy**
- ***Command* may be undoable (see *Memento*)**

by the Gang of Four

Command: Example



Command: Example

```
public class Bank { // Invoker, Client
   public void income(Account acc, long amount) {
      Operation oper = new Income(amount);
      acc.doOperation(oper);
   }

   public void transfer(Account from, Account to, long amount) {
      Operation oper = new Transfer(to, amount);
      from.doOperation(oper);
   }
}

public class Account {  // Reciever
   long balance = 0;
   Interest interest = new InterestA();
   History history = new History();

   public void doOperation(Operation oper) {
      oper.execute(this);
      history.log(oper);
   }
}
```

Command: Example

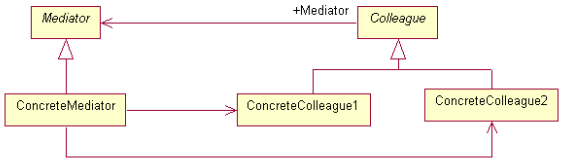
```
abstract public class Operation {  // Command
   public void execute();
}

public class Income {  // ConcreteCommand1
   public Income(long amount) {
      // store parameters...
   }

   public void execute(Account acc) {
      acc.add(amount);
   }
}

public class Transfer {  // ConcreteCommand2
   public Income(Account to, long amount) {
      // store parameters...
   }

   public void execute(Account from) {
      from.subtract(amount);
      to.add(amount);
   }
}
```

Mediator: Intent

- Define an object that **encapsulates how a set of objects interact**
- Promote loose coupling by **keeping objects from referring to each other explicitly**
- Allow **varying their interaction independently**

by the Gang of Four

Mediator: Structure



by the Gang of Four

Mediator: Participants

- **Mediator**
  defines an interface for communicating woth *Colleague* objects
- **Concrete Mediator**
  implements cooperative behaviour by coordinating *Colleagues*
  knows and maintains its *Colleagues*
- **Colleague classes**
  each of them knows the *Mediator*
  colleagues communicate with *Mediator* instead of another *Colleague* directly

by the Gang of Four

Mediator: Consequences

- **limited subclassing**
  - *Mediator* localizes behavior that otherwise would be distributed
  - changing the behavior requires subclassing the *Mediator* only
- **decoupling the colleagues from each other**
- **simplified object protocols**
  - *Mediator* replaces many-to-many associations with one-to-many
- **centralized control**
  - trade-off between complexity of interaction with compexity of a *Mediator*
  - *Mediator* becomes a hard to maintain monolith

by the Gang of Four

Mediator: Example

```
class Mediator {
   private boolean slotFull = false;
   private int number;

   public synchronized void storeMessage( int num ) {
      while (slotFull == true) {
         try {
            wait();
         } catch (InterruptedException e ) { }
      }
      slotFull = true;
      number = num;
      notifyAll();
   }

   public synchronized int retrieveMessage() {
      while (slotFull == false) {
         try {
            wait();
         } catch (InterruptedException e ) { }
      }
      slotFull = false;
      notifyAll();
      return number;
   }
}
```

by Vincent Huston

Mediator: Example

```
class Producer extends Thread {
   private Mediator med.
   private int id;
   private static int num = 1;

   public Producer(Mediator m) {
      med = m;
      id = num++;
   }

   public void run() {
      int num;
      while (true) {
         med.storeMessage(num = (int)(Math.random()*100));
         System.out.print("p" + id + "-" + num + " ");
      }
   }
}
```

by Vincent Huston

Mediator: Example

```
class Consumer extends Thread {
   private Mediator med.
   private int id;
   private static int num = 1;

   public Consumer(Mediator m) {
      med = m;
      id = num++;
   }

   public void run() {
      while (true) {
         System.out.print("c" + id + "-" + med.retrieveMessage());
      }
   }
}
```

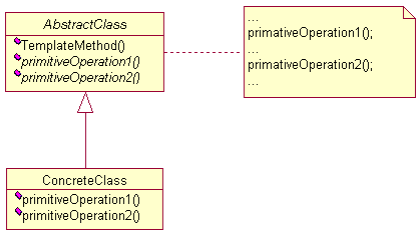
by Vincent Huston

Template Method: Intent

- Define the **skeleton of an algorithm in an operation**, deferring some steps to subclasses
- **Allow subclasses redefining certain steps** of an algorithm without changing the algorithm's structure

by the Gang of Four

Template Method: Structure



by the Gang of Four

Template Method: Participants

- **AbstractClass**
  - defines abstract primitive operations
  - implements a template method defining the skeleton of an algorithm
- **ConcreteClass**
  - implements primitive operations

by the Gang of Four

Template Method: Consequences

- **inverted control structure**
  - a superclass defers implementation to subclasses
  - *don't call us, we will call you*
- **different kinds of operations called by TemplateMethod**
  - concrete operations (on *ConcreteClass* or client classes)
  - concrete *AbstractClass* operations (methods useful for subclasses)
  - primitive operations (abstract methods)
  - factory methods
  - hook operations (default behavior that can be extended)

by the Gang of Four

Template Method: Example

```
public abstract class TitleInfo {
   private String titleName;

   // template method
   public final String processTitleInfo() {
      StringBuffer titleInfo = new StringBuffer();
      titleInfo.append(this.getTitleBlurb());
      titleInfo.append(this.getDvdEncodingRegionInfo());

      return titleInfo.toString();
   }

   // concrete abstract class methods
   public final void setTitleName(String titleNameIn) {
      this.titleName = titleNameIn;
   }
   public final String getTitleName() {
      return this.titleName;
   }

   // primitive operation – must be overriden
   public abstract String getTitleBlurb();

   // hook operation - may be overridden
   public String getDvdEncodingRegionInfo() {
      return " ";
   }
}
```

by Vincent Huston

Template Method: Example

```
public class DvdTitleInfo extends TitleInfo {
   private char encodingRegion;

   public DvdTitleInfo(String titleName, char region) {
      this.setTitleName(titleName);
      this.setEncodingRegion(region);
   }

   // new concrete methods
   public void setEncodingRegion(char region) {
      this.region = region;
   }
   public char getEncodingRegion() {
      return this.encodingRegion;
   }

   // overriden primitive operation
   public String getTitleBlurb() {
      return ("DVD: " + this.getTitleName() + ", starring "
         + this.getStar());
   }

   // overriden hook
   public String getDvdEncodingRegionInfo() {
      return (", encoding region: " + this.getEncodingRegion());
   }
}
```

by Vincent Huston

Template Method: Example

```
public class BookTitleInfo extends TitleInfo {
   private String author;

   public BookTitleInfo(String titleName, String author) {
      this.setTitleName(titleName);
      this.setAuthor(author);
   }

   // new concrete methods
   public void setAuthor(String authorIn) {
      this.author = authorIn;
   }

   public String getAuthor() {
      return this.author;
   }

   // overriden primitive operation
   public String getTitleBlurb() {
      return ("Book: " + this.getTitleName() + ", Author: "
         + this.getAuthor());
   }
}
```

by Vincent Huston

Template Method: Example

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

      TitleInfo bladeRunner =
         new DvdTitleInfo("Blade Runner", '1');
      TitleInfo electricSheep =
         new BookTitleInfo("Do Androids Dream of Electric Sheep?",
            "Phillip K. Dick");
      TitleInfo sheepRaider = new GameTitleInfo("Sheep Raider");

      System.out.println(" ");
      System.out.println("Testing bladeRunner "
         + bladeRunner.processTitleInfo());
      System.out.println("Testing electricSheep "
         + electricSheep.processTitleInfo());
      System.out.println("Testing sheepRaider "
         + sheepRaider.processTitleInfo());
   }
}
```

```
Testing bladeRunner DVD: Blade Runner, encoding region: 1
Testing electricSheep Book: Do Androids Dream of Electric Sheep?,
                            Author: Phillip K. Dick
Testing sheepRaider Game: Sheep Raider
```

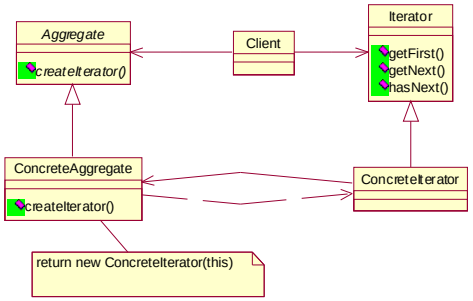
by Vincent Huston

Iterator: Intent

Provide a way **to access elements of an aggregate sequentially** without exposing its internal structure

by the Gang of Four

Iterator: Structure



by the Gang of Four

Iterator: Participants

- **Iterator**
  - declares an interface for accessing and iterating through *Aggregates*
- **Concrete Iterator**
  - performs necessary computations
- **Aggregate**
  - declares an interface for creating *Iterator*
- **Concrete Aggregate**
  - implements *Iterator* interface

by the Gang of Four

Iterator: Consequences

- **supports multiple variations of traversing the aggregate**
  - adapts only to a concrete class, not subclasses
- **multiple traversal allowed**
  - each *Iterator* keeps track of the running traversal
  - *Iterators* are stateful

by the Gang of Four

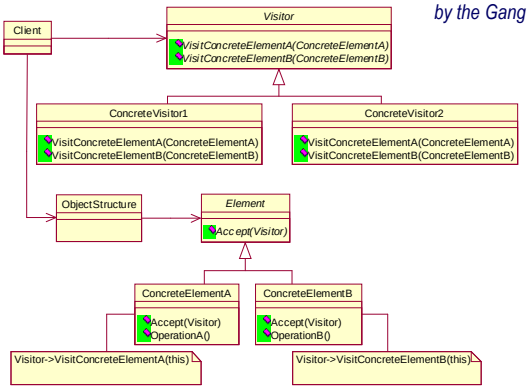
Visitor: Intent

- **Represent an operation to be performed on the elements of an object structure**.
- Allow **defining a new operation without changing the classes of the elements** on which it operates.

by the Gang of Four

Visitor: Structure

by the Gang of Four



Visitor: Participants

- **Visitor**
  - declares operations for every *ConcreteElement* to be visited
- **Concrete Visitor**
  - implements the operations
- **Element**
  - defines *accept()* operation parametrized with *Visitor*
- **Concrete Element**
  - implements *accept()* operation
- **Object Structure**
  - can enumerate its elements
  - may provide a high-level interface to allow the visitor to visit its elements

by the Gang of Four

Visitor: Consequences

- **easy adding new operations**
  - new *Visitors* can traverse the object structure
- **gathering related operations and separation of unrelated ones**
  - related behavior is localized in a *Visitor*
  - unrelated sets of behavior are partitioned in their own *Visitor* subclasses
- **difficult adding new ConcreteElements**
  - each *ConcreteElement* gives rise to a new operation on *Visitor* and corresponding *ConcreteVisitors*
- **visiting across class hierarchies**
  - unlike *Iterator*, the *Visitor* can visit objects of different classes

by the Gang of Four

Visitor: Consequences (cont.)

- **accumulating state**
  - *Visitors* can accumulate state during the structure traversal
- **breaking encapsulation**
  - pattern often enforces existance often public operations that access an element's internal state

by the Gang of Four

Visitor: Example

```
public class Bank {
   List<BankingProduct> products = new ArrayList<BankingProduct>();

   public List<BankingProduct > doReport(Report report) {
      List<BankingProduct > result
          = new ArrayList<BankingProduct >();

      for (BankingProduct product : products) {
         result.add(product.accept(report));
      }

      return result;
   }
}
```

Visitor: Example

```
public abstract class BankingProduct {
}

public interface Element {
   public BankingProduct accept(Report report);
}

public class Account extends BankingProduct implements Element {
   public BankingProduct accept(Report report) {
      if (isPriviliged(report)) {
         return report.visit(this);
      }

      return null;
   }
}

public class Credit extends BankingProduct implements Element {
   public BankingProduct accept(Report report) {
      if (isPriviliged(report)) {
         return report.visit(this);
      }

      return null;
   }
}
```

Visitor: Example

```
public class Over1000Report implements Visitor {
   public BankingProduct visit(Account acc) {
      if (acc.balance > 1000)
          return this;
      return null;
   }

   public BankingProduct visit(Credit credit) {
      if (credit.draft > 1000 && credit.isActive())
         return this;
      return null
   }
}
```

```
public class PassAllReport implements Visitor {
   public BankingProduct visit(Account acc) {
      return this;
   }

   public BankingProduct visit(Credit credit) {
      return this;
   }
}
```

Q & A