

Design Patterns (I)


Jun Hu

Department of Industrial Design
Eindhoven University of Technology
j.hu@tue.nl

<http://id00243.id.tue.nl/ObjectOrientationAndDesignPatterns>

13th March 2005

- Introduction
- Iterator
- Composite
- Factory
- Summary

 Introduction

What? A recurring design.

Why? Reuse solutions

- Definitions
- Types of Pattern
- Pattern elements
- Reuse: Inheritance
- Reuse: Composition
- Designing for Change

 ► Introduction ► Definitions

Alexander et al. “A Pattern Language”, 1977

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over without ever doing it the same way twice”

 ► Introduction ► Definitions

Alexander et al. “A Pattern Language”, 1977

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over without ever doing it the same way twice”

James W. Cooper “The Design Patterns Java Companion”

“ . . . design patterns describe how objects communicate without become entangled in each other’s data models and methods.”

“ . . . design patterns are not just about the design of objects, but about the communication between objects.”

 ► Introduction ► Types of Pattern

- Creational Patterns
 - Create objects
 - E.g., *Factory* method patterns
 - More power over how objects are created

 Introduction ▶ Types of Pattern


- Creational Patterns
 - Create objects
 - E.g., *Factory* method patterns
 - More power over how objects are created
- Structural Patterns
 - Compose objects in complex ways
 - E.g., *Composite*, *Decorator* patterns

 Introduction ▶ Types of Pattern


- Creational Patterns
 - Create objects
 - E.g., *Factory* method patterns
 - More power over how objects are created
- Structural Patterns
 - Compose objects in complex ways
 - E.g., *Composite*, *Decorator* patterns
- Behavioral Patterns
 - Control communication and flow of program
 - E.g., *Iterator*, *Visitor* patterns

 Introduction ► Pattern elements

1. A name
 - E.g., iterator, composite, visitor
2. The problem
 - Where to apply the pattern
3. The solution
 - General implementation of the pattern
4. Consequences
 - Tradeoffs. What benefits/penalties for a particular pattern

 ► Introduction ► Reuse: Inheritance

- Inheritance gives us reuse
 - Reuse code inherited from parent class
- Disadvantages:
 - Inheritance decided at compile time
 - Change in parent implementation can force change in subclasses
 - * A kind of coupling?

 ► Introduction ► Reuse: Composition

- Composition gives us reuse
 - Compose pre-existing structures (objects) into a more complex whole
- Disadvantages:
 - Need careful design of interfaces so that objects may interact
 - Coupling

Gamma, Helm, Johnson, Vlissides:

“Favour object composition over class inheritance”

 ► Introduction ► Designing for Change

- Maximizing reuse
 - Design code that is robust to changes
 - Unanticipated change → redesign expensive!

 ► Introduction ► Designing for Change

- Maximizing reuse
 - Design code that is robust to changes
 - Unanticipated change → redesign expensive!

- We won't look at all patterns, but only
 - Iterator
 - Composite
 - Factory
 - Decorator
 - Visitor

- Far more patterns. This is an introduction only!

Coming next: Iterator

▶ Iterator

- Problem
- Solution
- Example
- Consequences

 ▶ Iterator

- Problem
 - Solution
 - Example
 - Consequences
-
- Iterator pattern is implemented directly in Java

```
public class Zoo {  
    private Collection zooInventory = new LinkedList();  
  
    public void feedAnimals() {  
        Iterator itr = zooInventory.iterator();  
        while (itr.hasNext()){  
            // Feed each animal in turn  
        }  
    }  
}
```

 ► Iterator ► Problem

- Problem
 - Want to traverse an aggregate (e.g., a list) without knowing internal details

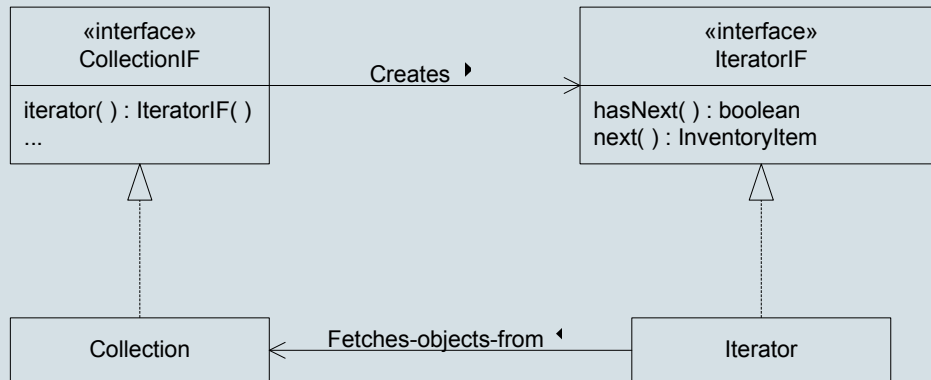
Iterator ▶ Solution


• Solution

- Provide an interface for accessing/traversing elements In Java collections interface:

```
public Iterator iterator ();
```

- I.e., can ask any collection to return an Iterator



 ▶ Iterator ▶ Example

- The Java Iterator hides details
 - Gives us traversal interface no matter what type of collection
- Compare two different implementations of zooInventory:
 - as a **LinkedList**

```
private Collection zooInventory = new LinkedList();

public void feedAnimals() {
    Iterator itr = zooInventory.iterator();
    while (itr.hasNext()){
        // Feed each animal in turn
    }
}
```

- as an **ArrayList**

```
private Collection zooInventory = new ArrayList();

public void feedAnimals() {
    Iterator itr = zooInventory.iterator();
    while (itr.hasNext()){
        // Feed each animal in turn
    }
}
```

 ► Iterator ► Consequences

- Hides traversal details
 - E.g., traversal of trees in pre-order, post-order, etc.
 - Can change traversal algorithm later
 - Information hiding!
- Iterator looks after own traversal state
 - Can use multiple iterators on same structure
 - Could not do this if structure itself held state

Coming next: Composite

Composite

- Problem
- Solution
- Example
- Consequences

 ▶ Composite ▶ Problem

- Problem
 - Part and whole; may have to use different interfaces
 - Want to use the same interface

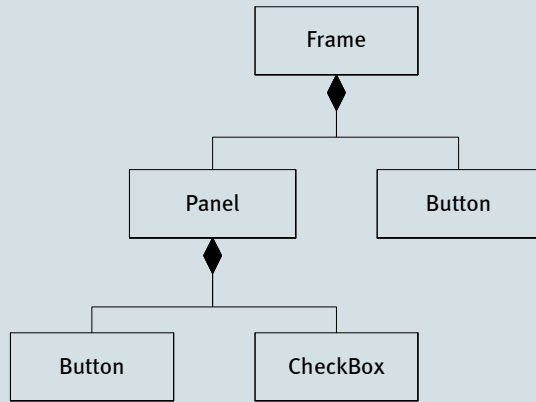
 ▶ Composite ▶ Problem

- Problem
 - Part and whole; may have to use different interfaces
 - Want to use the same interface

- For example,
 - AWT (*Abstract Window Toolkit*) Frames are Containers (e.g., contain Panel, Buttons)
 - repaint() Button
 - repaint() Panel (recursively paint Button)
 - repaint() Frame (recursively paint Panel, Button)

 Composite ▶ Solution

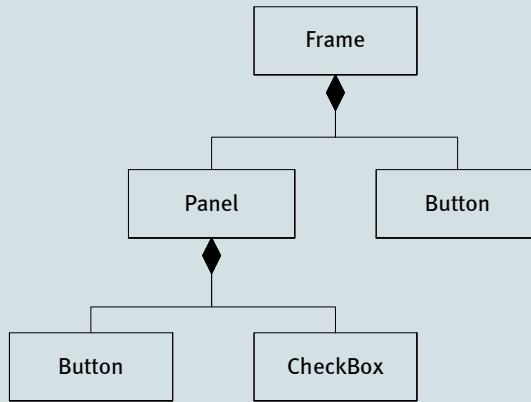
- Many times store a tree structure of objects



- Want one interface
 - For leaf or node

 Composite ▶ Solution

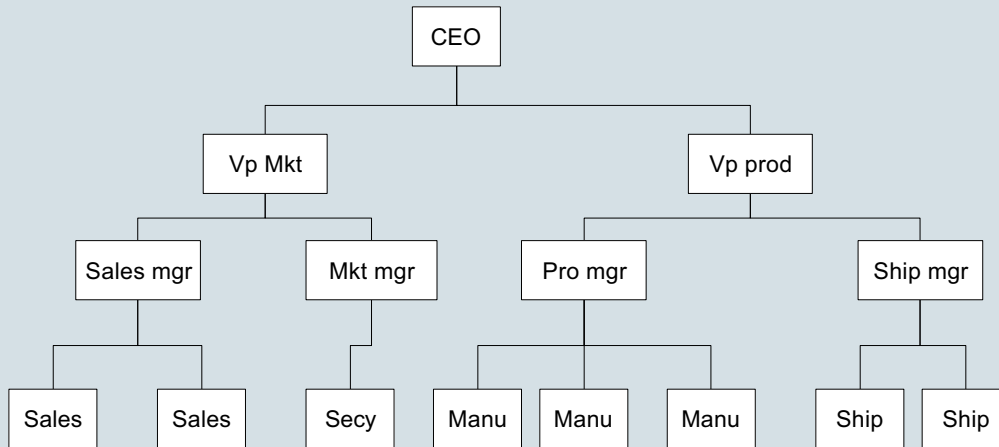
- Many times store a tree structure of objects



- Want one interface
 - For leaf or node
- Solution:
 - Recursion

 Composite Example

- From *The Design Patterns Java Companion*, Employees have subordinates:



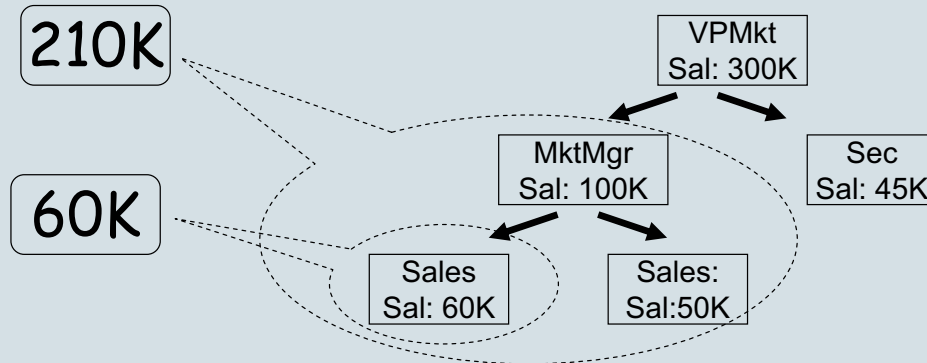
 ▶ Composite ▶ Example

- Employees have
 - A salary
 - A list (vector) of subordinates

```
public class Employee {  
    ...  
    float salary;  
    Vector subordinates;  
    ...  
    float getSalary() {  
        return salary;  
    }  
    void add(Employee e) {  
        subordinates.addElement(e);  
    }  
}
```

Composite ▶ Example

- Want salary function:
 - calculates salary of employee + employee's subordinates



 ▶ Composite ▶ Example

- Recursively descend to the subordinates
 - Sub-trees calculate `getSalaries()`
 - Pass value back up tree

```
public float getSalaries() {  
    float sum = salary; //this one's salary  
  
    //add in subordinates salaries  
    Iterator i = subordinates.iterator();  
    while(i.hasNext())  
        sum += ((Employee)i.next()).getSalaries();  
  
    return sum;  
}
```

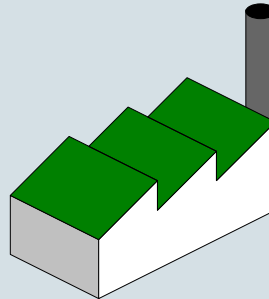
 ► Composite ► Consequences


- Same interface for primitive (i.e. single) object or composite object
 - Simpler access by client code
- Easier to add new components
 - New subclasses work automatically with client code

Coming next: Factory

 ► Factory

- Problem
- Solution
- Example
- Consequences



 ► Factory ► Problem

- If don't know (at compile time) what object type we need
 - I.e., (decide at runtime)
- Can decide to create a new object, or return an existing one
 - Seamless sharing of objects
- When need more control over object creation than `new()`

 ► Factory ► Solution

- Method for controlling the creation of new objects
- Why?
 - Single method can create a variety of different objects
 - Can manage memory better

 ► Factory ► Solution

- Method for controlling the creation of new objects
- Why?
 - Single method can create a variety of different objects
 - Can manage memory better
- Solution:
 - Call
 - * `factoryMethod()` not `new()`
 - * `factoryMethod()` chooses the object to return

 ► Factory ► Example

- Have an applet that allows two forms of name input:

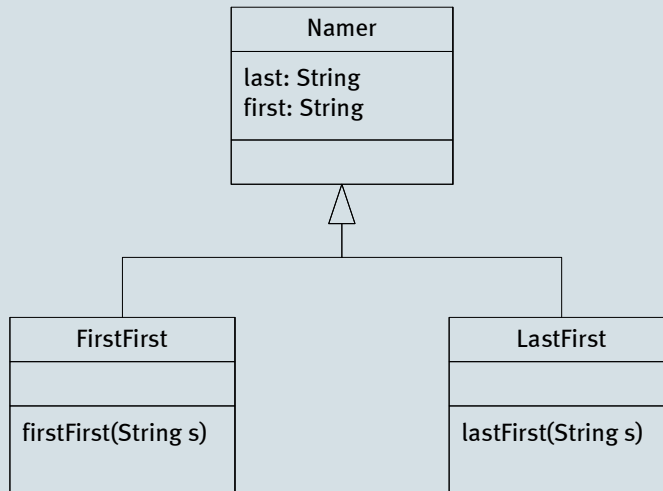
Enter your name:

Enter your name:

- Many implementations . . . try using factory . . .

 ► Factory ► Example

• UML



- Decide to return different object depending on input type

 ► Factory ► Example

• Namer

```
class Namer {  
    protected String last;  
    protected String first;  
  
    public String getFirst() {  
        return first;  
    }  
  
    public String getLast() {  
        return last;  
    }  
}
```

 ► Factory ► Example

● FirstFirst

Enter your name:


```
class FirstFirst extends Namer {  
  
    public FirstFirst(String s) {  
        int i = s.lastIndexOf("_");  
        if (i > 0) {  
            //left is first name  
            first = s.substring(0, i).trim();  
            //right is last name  
            last = s.substring(i+1).trim();  
        }  
        else {  
            first = ""; // put all in last name  
            last = s; // if no space  
        }  
    }  
}
```

 ► Factory ► Example

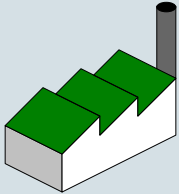
● LastFirst

Enter your name:

```
class LastFirst extends Namer {  
  
    public LastFirst(String s) {  
        int i = s.indexOf(","); //find comma  
        if (i > 0) {  
            //left is last name  
            last = s.substring(0, i).trim();  
            //right is first name  
            first = s.substring(i + 1).trim();  
        }  
        else {  
            last = s; // put all in last name  
            first = ""; // if no comma  
        }  
    }  
}
```

 ► Factory ► Example

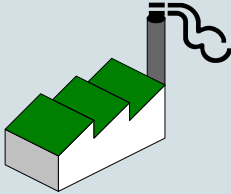
- The factory: `getNamer()` returns object of either subclass



```
class NameFactory {  
  
    public Namer getNamer(String entry) {  
        int i = entry.indexOf(","); //comma determines  
  
        //name order  
        if (i>0)  
            return new LastFirst(entry); //return one class  
        else  
            return new FirstFirst(entry); //or the other  
    }  
}
```


 ► Factory ► Example

- Using the factory




```
private void computeName() {  
  
    Namer namer= nfactory.getNamer(entryField.getText());  
  
    txFirstName.setText(namer.getFirst());  
    txLastName.setText(namer.getLast());  
  
}
```

- . . . without ever knowing which subclass you use


 ► Factory ► Consequences

- Allows dynamic (runtime) choice of subclass in creation
 - Subclass need not be a restriction, factory method can return **Object**
- Dynamic choice in *how* object is instantiated
 - Choose constructor
- Can Choose to return reference to preexisting object
 - Useful for memory management

Coming next: Summary


 ► Summary

- Iterator
 - Standard interface for traversal
 - Behavioral pattern

 ▶ Summary

- Iterator
 - Standard interface for traversal
 - Behavioral pattern

- Composite
 - Standard interface for processing tree-structures
 - Structural pattern

 ► Summary

- Iterator
 - Standard interface for traversal
 - Behavioral pattern
- Composite
 - Standard interface for processing tree-structures
 - Structural pattern
- Factory
 - Power over object creation
 - Creational pattern