Design Patterns
Java/C# Edition

Joseph W. Yoder

The Refactory, Inc.
www.refactory.com

yoder@refactory.com

The Refactory Principals

John Brant
Brian Foote
Ralph Johnson
Don Roberts
Joe Yoder

Refactory Affiliates

Dragos Manolescu
Brian Marick
Bill Opdyke

The Refactory, Inc.

The Refactory principles and affiliates are experienced in software development, especially in object-oriented technology. We've been studying and developing software since 1973. Our current focus has been object-oriented technology, software architecture, and patterns. We have developed frameworks using Smalltalk, C++, and Java, have helped design several applications, and mentored many new Smalltalk, Java and C++ developers. Highly experienced with Frameworks, Software Evolution, Refactoring, Objects, Flexible and Adaptable Systems (Adaptive Object-Models), Testing, Workflow Systems, and Agile Software Development including methods like eXtreme Programming (XP).

Design Patterns

- A new category of knowledge
- Knowledge is not new, but talking about it is
- Make you a better designer
- Improves communication between designers

Why Patterns?

People do not design from first principles.

People design by reusing things they've seen before.

Same techniques appear over and over.

Software industry needs to document what we do.

Patterns

Patterns in solutions come from patterns in problems.

"A pattern is a solution to a problem in a context."

"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."

Christopher Alexander -- *A Pattern Language*

Patterns

A pattern is a balance of forces

Forces: all the issues that affect a problem.

Typical software design forces: efficiency, clarity, maintainability, safety.

Design is the art of making trade-offs.

Patterns should make trade-offs explicit.

Patterns are not

- Patterns are not idioms
- Patterns are not algorithms
- Patterns are not components
- Patterns are not a "*silver bullet*"

Object-Oriented Design Patterns

Repeating organization of classes (objects) and the way they interact

Design Patterns: Elements of Reusable Object-Oriented Software

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

Addison-Wesley, 1995.

Overall Goals

You will be able to:

- describe what patterns are, and why they are important
- recognize all the patterns in "Design Patterns"
- *use patterns to solve specific design problems*
- *use patterns to document a design*
- learn new patterns when you need them

You will not:

- learn everything there is to know about patterns

Class Overview

- Presentation
- *Reading Groups*
- *Exercises*

Patterns in Java and C#

Java and C# frameworks were influenced by the GoF

- black-box
- use patterns (they're *everywhere*)

Java and C# has features that affect how design patterns are applied

- interfaces
- serialization
- distribution
- concurrency
- GUI (AWT, Swing)
- inner classes
- protection

Outline of Course

What are patterns? – Composite, Chain of Responsibility, Template Method

More Patterns – Decorator, Null Object, Strategy

How patterns work together

Abstract Factory, Adapter, Builder, Command, Factory Method, Memento, Observer, Prototype, Singleton, State

Documenting system designs with patterns

Centralized vs. distributed - Interpreter, Visitor, Iterator

Bridge, Facade, Flyweight, Mediator, Proxy

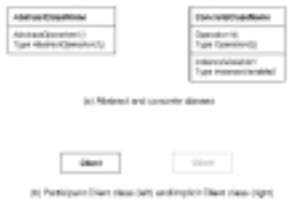
Other Patterns and where to find more information

Notation

Design Patterns Book uses OMT

We use this to show the correlation

Sometimes we use UML which is similar



More Notation

Class Diagrams:

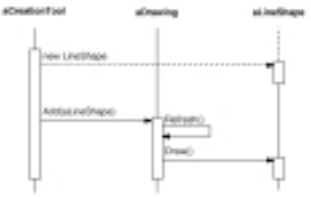
(c) Class relationships

(d) Pseudocode annotation

Yet More Notation

Object Diagrams:

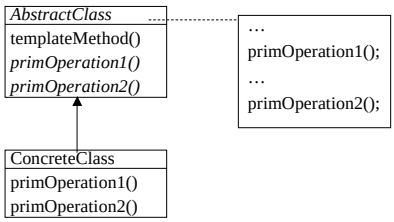
Interaction Diagrams:



Template Method

Problem: Some classes have a similar algorithm, but it is a little different for each class.

Solution: Define the skeleton of the algorithm as a method in a superclass, deferring some steps to subclasses.



Template Method

A template method calls abstract methods.

Usually a template method is created by generalizing several existing methods.

Template Method separates the invariant part of an algorithm from the parts that vary with each subclass.

Template Method (example)

```
public abstract class View {
    public abstract doDisplay();
    public void display() {
            setFocus();
            doDisplay();
            resetFocos();
    }
}
public class ButtonView extends View {
    public void doDisplay() {
            setButtonWidth();
            …}
}
```

```
public class ListView extends View {
    public void doDisplay() {
            setScrollBarWidth();
            …}
}
```


Composite

Context:

Developing OO software

Problem:

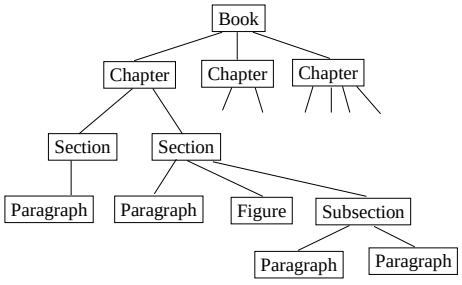
Complex part-whole hierarchy has lots of similar classes.

Example: document, chapter, section, paragraph.

Forces

- simplicity -- treat composition of parts like a part
- power -- create new kind of part by composing existing ones
- safety -- no special cases, treat everything the same

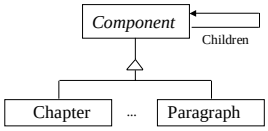
Document as a Tree



Composite

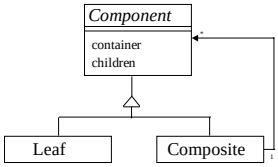
Idea: make abstract "component" class.

Alternative 1: every component has a (possibly empty) set of components.



Problem: many components have no components

Composite Pattern



Composite and Component have the exact same interface.

- interface for enumerating children
- Component implements children() by returning empty set
- interface for adding/removing children?

Two Design Alternatives

Component does not know what it is a part of.

Component can be in many composite.

Component can be accessed only through composite.

Component knows what it is a part of.

Component can be in only one composite.

Component can be accessed directly.

Component Knows its Composite

Rules when component knows its single composite.

A is a part of B if and only if B is the composite of A.

Duplicating information is dangerous!

Problem: how to ensure that pointers from components to composite and composite to components are consistent.

Ensuring Consistency

Solution:

Only public operations that change container are addComponent/removeComponent

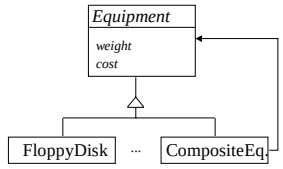
These operations update the container of the component.

There is no other way to change the container.

```
Composite addComponent(Component c) {
  components.add(c);
  c.parent = this;
}
```

In C++, Composite must be friend of component.

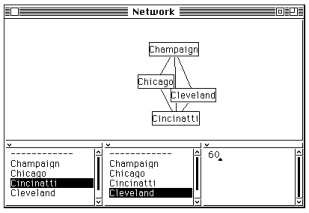
Example: Equipment



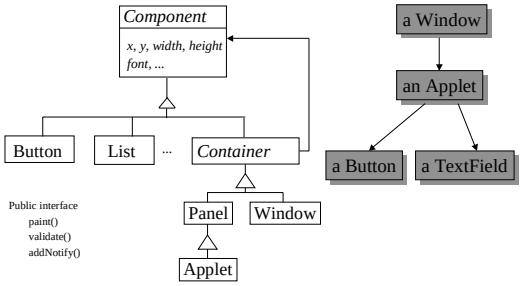

```
class CompositeEquipment {
    int weight() {
        int total = 0; Equipment item;
        for (Enumeration e = children() ; e.hasMoreElements();
                item = (Equipment) e.nextElement()
        {
            total += item.weight;
        }
        return total;
    }
}
```

Example: Views and Figures

Big window can contain smaller windows.



The Java Component Class



Adding Container

Standard questions for adding:

- Where is the collection stored?
- Add at front or rear?
- How do you update back pointer to parent?
- What if component already is in a container?
- Does a component need to know if its position changed?

```
public Component add(Component comp) {
   addImpl(comp, null, -1);
   return comp;
}
```

More of Container add()

```
protected void addImpl(Component comp, Object constraints, int index) { ...
    /* Add component to list; allocate new array if necessary. */
    if (index == -1 || index == ncomponents) {
        component[ncomponents++] = comp;
    } else {
        System.arraycopy(component, index, component,
                         index + 1, ncomponents - index);
        component[index] = comp;
        ncomponents++;
    }
```


More of Container add()

```
/* What do you do if component already has parent? */
    if (comp.parent != null) {
        comp.parent.remove(comp);
    }
    comp.parent = this;

/* How can component know it has a new position?  */
    if (peer != null) {
        comp.addNotify();
    }
```


Painting

If Container used only the Composite pattern, it would implement Paint like:

```
public void paint(Graphics g) {
    for (int i = 0; ++i <= ncomponents; ) {
        component[i] .paint(g);
    }
}
```

But it also uses the Bridge pattern, which changes things.

Summary of Composite

Composite is a kind of Component

Permits arbitrary hierarchies

Add/remove Component from Composite

Operations on Composite iterate over Components

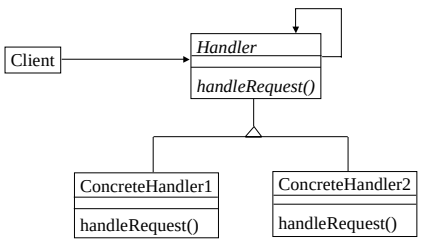
Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Usually found with Composite - chain of parents.

Examples:
- "inheriting" color from car
- event handlers in GUI

Chain of Responsibility



Chain of Responsibility

Usually mixed with other patterns

- Composite often has Chain of Responsibility up the tree.
- Sometimes request is encoded as a Command
- Sometimes request sent to Strategy

Example: GUI System (Windows, Button Widgets, …)

```
onMouseClick() {…
    if hookmethod available handle request
    else parent.onMouseClick();
…}
```


What is a Design Pattern?

Design Pattern: repeating structure of design elements

Pattern is about design, but includes low-level coding details.

Pattern includes both problem and solution.

What is a Design Pattern?

Details of implementing pattern depend on language and environment.

Pattern is often not the most obvious solution.

Pattern can be applied to many kinds of problems.

Parts of a Pattern (Alexander)

Problem - when to use the pattern
Solution - what to do to solve problem
Context - when to consider the pattern

Forces - pattern is a balance of forces

Consequences, positive and negative

Parts of a Pattern

Examples:

Teach both problem and solution

Are the best teacher

Are proof of pattern-hood

Parts of a Pattern (Gamma et. al.)

Intent - brief description of problem and solution
Also Known As
Motivation - prototypical example
Applicability - problem, forces, context
Structure/Participants/Collaborations - solution
Consequences - forces
Implementation/Sample Code - solution
Known Uses
Related Patterns

GoF Design Patterns

Creational patterns
- Abstract factory
- Builder
- Factory method
- Prototype
- Singleton

Structural patterns
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Decorators

Decorators add a responsibility to an object by
- making the object a component
- forwarding messages to component and handling others

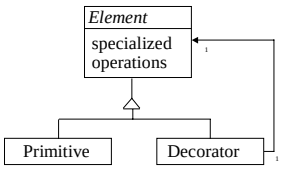
Possible examples from Java
Double, Integer, Float, etc.

Decorators add an attribute to an object.

Decorator forwards operations to the component.

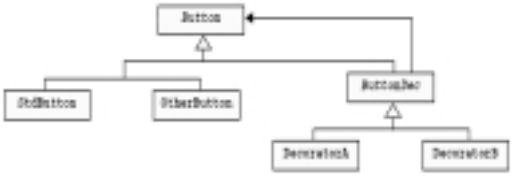
Component gets values from its decorator.

Decorator Structure



Decorator forwards most operations to the object it is decorating.

Decorator Example



Design Patterns in AWT

1.0 Event-handling by Chain of Responsibility
problem, either Mediator or lots of subclasses
1.1 Event-handling by Observer and Adapter

Java uses lot's of Patterns but just because you use a Pattern doesn't necessarily mean a good design!

Strategy Pattern

Define a family of algorithms, encapsulate each one, and make them interchangeable.

Strategy pattern means:

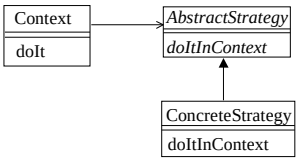
- easy to replace one algorithm with another
- can change dynamically
- can make a class hierarchy of algorithms
- can factor algorithms into smaller reusable pieces
- can encapsulate private data of algorithm
- can define an algorithm in one place

Strategy Pattern

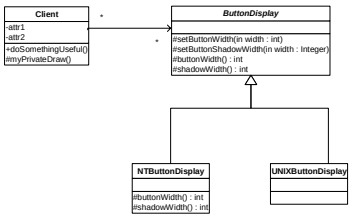
For procedural languages you would have conditional code spread throughout your application for dealing with special cases.

```
onDisplayButton()
   case OS of:
    'NT' : setButtonWidth: 100;
    'UNIX': setButtonWidth: 125;
    ...
onMousePressed()
    case OS of:
    'NT' : setButtonShadowWidth: 10;
    'UNIX': setButtonShadowWidth: 15;
    ...
```


Strategy



Strategy
(an example)



Moving Code "Refactoring"

To move a function to a different class, add an argument to refer to the original class of which it was a member and change all references to member variables to use the new argument.

If you are moving it to the class of one of the arguments, you can make the argument be the receiver.

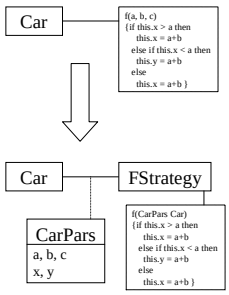
Moving function f from class X to class B

```
class X {
    int f(A anA, B aB){
        return (anA.size + size) / aB.size;
    } ...
class B {
    int f(A anA, X anX){
        return (anA.size + anX.size) / size;
    } ...
```


Moving Code

You can also pass in a parameter object which gives the algorithm all of the values that it will need.

Inner Classes can help by providing access to values that the algorithm may need.



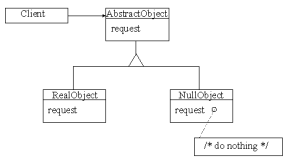
NullObject

Author: Bobby Woolf, PLoPD 3

Intent:

- provide surrogate for another object that shares same interface
- usually does nothing but can provide default behavior
- encapsulate implementation decisions of how to do nothing

Structure:



Design Patterns

Teaching
- help novices learn to act like experts

Design
- vocabulary for design alternatives
- help see and evaluate tradeoffs

Documentation
- vocabulary for describing a design
- describes "why" more than other techniques

Review

Patterns: solutions to recurring problems

OO design patterns: Recurring structures of objects that solve design problems

Stretch from design to code

We have seen: Composite, Chain of Responsibility, Decorator, Null Object, Strategy, Template Method

Goals of Next Session

Be able to recognize the creational patterns: Abstract Factory, Builder, Factory Method, Prototype, Singleton

Be able to describe relationships among creational patterns

Be able to recognize Adapter, Command, Memento, Observer, & State

Learn more about how patterns work together

How Patterns Work Together

Some patterns are commonly used together

Some patterns are alternatives

Some patterns have common context

Creational patterns:

- Some objects have to create other objects.
- How can we parameterize them with the kind of objects that they create?

Creational Patterns

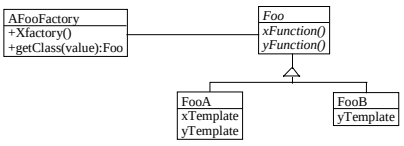
Factory Method

Factory Object

- Abstract Factory
- Builder
- Prototype

Singleton

Factory Method



Factory Method

Don't (call constructor / send message to class) directly.
Make a separate function / method to create object.

Advantages:
- can change class of product in subclass
- can produce easier to read functions

Disadvantages:
- slower, bulkier
- harder to read ALL the code

Factory Object

Problem with factory method -- have to create subclass to parameterize.

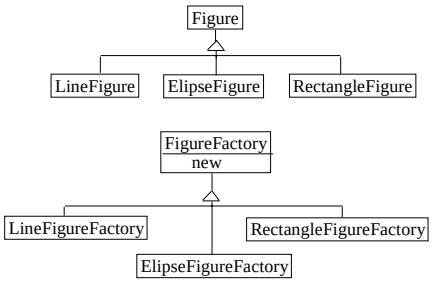
Often end up with parallel class hierarchies.

Example: subclass of Tool for each figure you want to create or a large case statement or many methods.

Alternative: parameterize CreationTool with object that creates figure

(Note: Factory Object is generalization of Abstract Factory, Builder, and Prototype. It is not in the book.)

Example



Applicability

Use factory objects:

- when system creates them automatically
- when more than one class needs to have product specified
- when most subclasses only specialize to override factory method

Prototype

Making a class hierarchy of factories seems wasteful.

The parameters of an object can be as important as its class.

Solution:

Use any object as a factory by copying it to make a new instance.

Advantages

Don't need new factory hierarchy.

Can make new "class" by parameterizing an object

Disadvantages

Requires robust copying

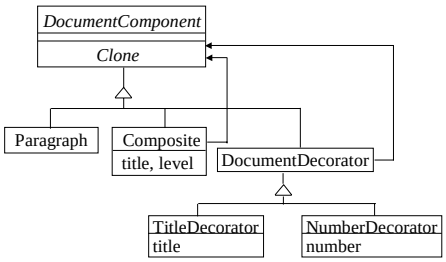
Prototype

Problem: a "chapter" or a "section" is a set of objects, not a single object. Users want to "create a new chapter". How should system create set of objects?

Solution: Specify the kind of objects to create by a prototypical instance, and create new objects by copying the prototype. If object is a composite or decorator then its entire substructure is copied.

Advantage: users can create new objects by composing old ones, and then treat the new object as a "prototype" for a whole new "class".

Prototype

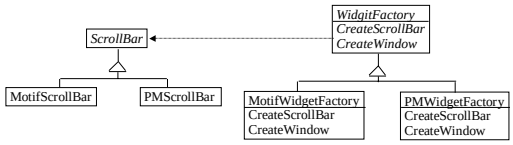


Abstract Factory

Sometimes a group of products are related -- if you change one, you might need to change them all.

Solution:

Make a single object that can make any of the products.



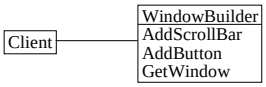
Builder

Complex objects require a lot of work to make.

Solution:

Factory must keep track of partly built product.

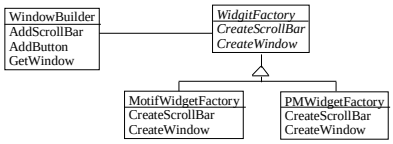
Client specifies product by performing series of operations on factory.



Implementing Builder

Builder can make components using

- Factory Method
- Singleton (to come)
- Abstract Factory, or
- Prototype



Summary of Factory Patterns

Factory method -- use in simple cases so that you can change product

Abstract factory -- use when there is a set of related products

Builder -- use when product is complex

Prototype -- use when Factory Method is awkward and when classes are not objects, or when you want to specify new "classes" by composition

Singleton

What if you want to make sure that a class has only one instance?

One possibility is global variables. Another is using static member functions.

Best solution: store single instance in static member variable.

Singleton in Java

```
abstract public class Singleton
{
   protected Singleton() {}
   abstract protected Singleton makeInstance();
   private static Singleton soleInstance = null;
   public static Singleton Instance() {
        if (soleInstance == null)
                soleInstance = makeInstance();
        return soleInstance;
};
```


Summary

Certain combinations of patterns are common.

- Abstract Factory and Factory Method
- Builder and Singleton

Often one pattern is used to implement an object in another.

A single object will play different roles in different patterns.

Goals of Next Session

Learn State and Observer (Listeners)

Learn Memento

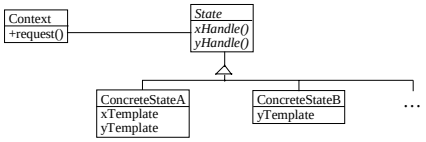
See more about how Patterns work together

State Pattern

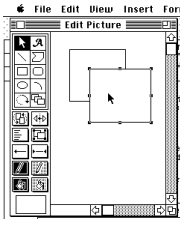
Problem: an object whose behavior changes as its state changes

Solution: make the state be a separate object, and delegate to it.
This results in a new class hierarchy of states.
Design of state is closely coupled to design of object.
Operations on states will change the state of the object.

State Pattern



Toolbar State Example



State

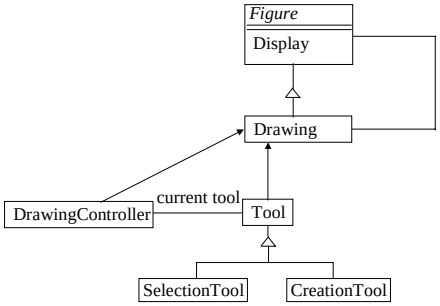
Behavior of drawing editor changes when you select a different tool.

Tools are the "current state" of the DrawingController; it delegates operations to the current state.

Make a class hierarchy of Tools.

DrawingController points to its "current Tool".

State



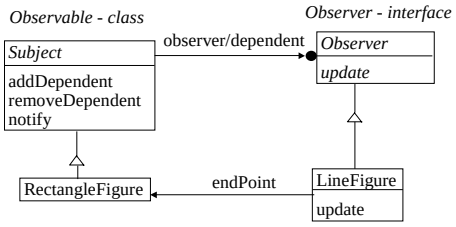
Observer Pattern

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

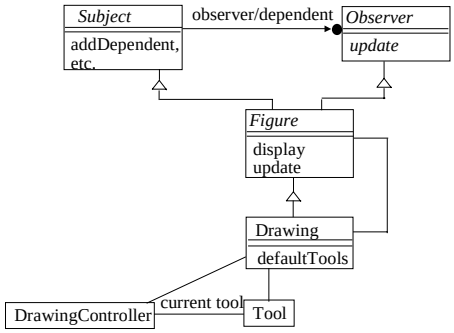
Example: Graphics system - moving box causes connecting lines to move.

Observer Pattern

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Observer Pattern



Event Handling

AWT 1.0 uses Chain of Responsibility

AWT 1.1 uses Observer

Shows the trade-offs between patterns

Shows Patterns != Good

Using Observer

Decide whether object is Subject, Observer, or both

Subjects must call notify() when they change state

Observers must define update()

Observers must register with Subjects

What are the arguments of notify() and update()?

Observer in Java

Original implementation of the Observer pattern:
Observer/Observable.

Observer is an interface.

Observable is a class that implements the ability to keep track of a set of Observers.

More modern implementation is the Listeners.

Listening instead of Observing

EventSource is a subject

EventListener is an observer

Many kinds of EventListeners,
each with their own interface

Different Kinds of Listeners

ActionListener
- actionPerformed(ActionEvent)

ComponentListener
- componentResized(ComponentEvent)
- componentMoved(ComponentEvent)
- componentShown(ComponentEvent)
- componentHidden(ComponentEvent)

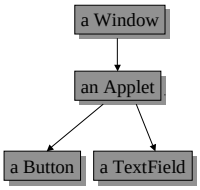
Events in AWT 1.1

Applet is a "Listener"

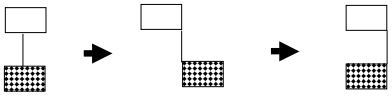
Button has methods
- addActionListener()
- processActionEvent()

Applet registers with Button.

When Button processes action event, it calls applet



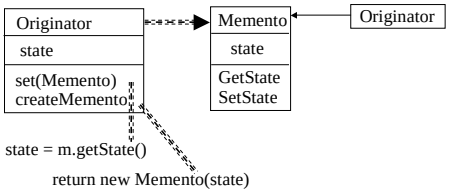
Memento



Undo is not enough in the presence of a constraint system.
Must go back to same state, not just reverse operation.

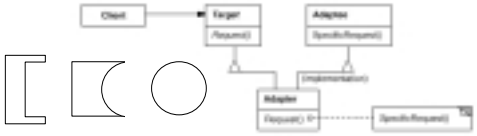
Memento

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



Target **Adapter** **Adaptee**

Command

Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Menus often do this.

Summary

New patterns: Abstract Factory, Builder, Factory Method, Prototype, Singleton, Adapter, Command, Memento, Observer, State

See How Patterns work together

Use Patterns to Document a Design

Goals of next session

Learn Interpreter, Iterator, Visitor

Be able to distribute an algorithm over a class hierarchy, or centralize it

Be able to explain some of the different kinds of trade-offs that patterns can make

Learn more about how patterns work together

Replacing Cases with Subclasses

Advantages

- instead of modifying case statements, add a new subclass
- can use inheritance to make new options

Disadvantages

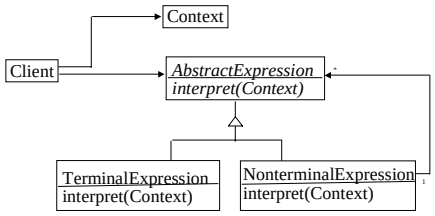
- program is spread out,
  - + harder to understand
  - + harder to replace algorithm
- state of object can change, but class can not

The Interpreter Pattern

To write a little object-oriented interpreter for a language L:

1) make a subclass of LExpression for each rule in the grammar of L
2) for each subclass, define an Interpret method that takes the current context as an argument.
3) define interface for making a tree of LExpression.
4) define a program for L by building a tree.
5) run a program by calling Interpret() on the root of the tree

Interpreter



Spreadsheet Rules

Spreadsheet rules of the form D3 + D4 or Subtotal(D2:D8)

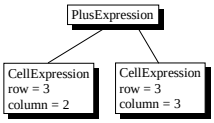
Grammar is

```
expression ::= expression1 '+' expression |
               expression1 '-' expression |
expression1 ::= expression '*' expression |
               expression '/' expression |
               number | cellID |
               'Subtotal(' range ')'
range ::= cellID ':' cellID
```

Spreadsheet Objects

new PlusExpression(new CellExpression(3,2),
new CellExpression(3,3))

Equivalent to “C2 + C3”



Appling the Interpreter Pattern

Step 1: *Make a subclass of Expression for each rule in grammar*

Expression

- BinaryExpression
  - PlusExpression, MinusExpression, TimesExpression, DivideExpression
- ConstantExpression
- CellExpression
- SubtotalExpression

Applying the Interpreter Pattern

Step 2. *Define a* **value(Spreadsheet)** *method for each subclass of Expression*

```
abstract class Expression {
   public Number value(Spreadsheet s);
class PlusExpression extends Expression {
   public Number value(Spreadsheet s) {
        return operand1.value(s) + operand2.value(s);
class  CellExpression extends Expression
   public Number value(Spreadsheet s) {
        return s.cellvalue(row, column);
```

Applying the Interpreter Pattern

Step 3: *Define constructors for making expression tree*

```
Expression(Expression e1, Expression e2) {
  operand1 = e1;
  operand2 = e2;
}
```

Step 4,5: *Build tree and evaluate it.*

```
ss.setExpression(3,4,new PlusExpression( new
   CellExpression(3,2), new CellExpression(3,3)));
ss.cellValue(3,4)
```


Interpreter Pattern Examples

Other examples of Interpreter pattern:

- producing Postscript for a document
- regular expression checker
- figuring out the value of an insurance policy
- compiling a program

In C, the interpreter would be a switch statement.

Easy to add new kinds of expressions to the spreadsheet -- don't have to modify any existing code.

When to Centralize Algorithm

Use centralized algorithm when you need to

- change entire algorithm at once
- look at entire algorithm at once
- work with only a few kinds of components
- change algorithm, but not add new classes of components

Visitor pattern

Visitor lets you centralize algorithm, lets you create a family of algorithms by inheritance, and lets you define a new operation without changing the classes of the elements on which it operates.

Major problem is that adding a new kind of parse node requires adding a new function to each visitor.

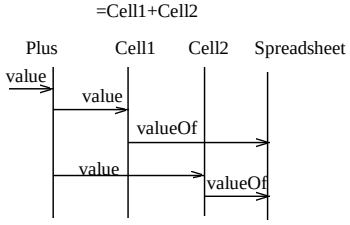
Interpreter To Visitor

- two kinds of classes: parse tree nodes and node visitor
- parse tree nodes have *Accept* function

```
class PlusExpression extends BinaryExpression
  public Object accept(ExpressionVisitor v)
  {
    return v.visitWithPlusExpression(this);
  }
```

- each parse tree node calls a different Visit function
- visitor defines a Visit function for each class of node

Interpreter without Visitor



Double Dispatch and Visitor

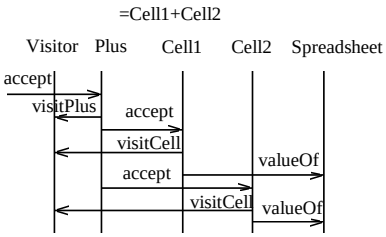
Double Dispatch – Effectively, the Visitor pattern lets you add operations to classes without changing them. Visitor achieves this by using a well known technique called double-dispatch. Double dispatch operation gets executed is dependent upon the kind of request and the type of the receiver. (Search google for Double Dispatch and Java).

1 + 4.5 = 5.5

- Double dispatch can be used to coerce the right type

```
Integer
    + (Number aNumber)
        return aNumber.addFromInteger(this)
```


Interpreter with Visitor

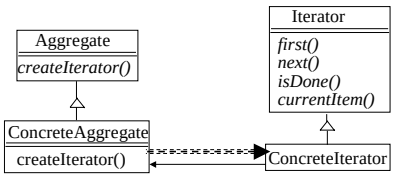


Related Patterns

- Several patterns are often used with Interpreter
  - Visitor - to separate algorithm from tree classes
  - Iterator - to make traversal more abstract
  - Composite - to make tree
  - Template Method - to put reusable code in abstract class

Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying implementation.



Using Enumeration-style Iterators

```
public void printEmployees (Employees emp)
{
   for (e = emp.employees(); e.hasMoreElements(); )
    {
       Employee currentEmployee
           = (Employee) e.nextElement();
      currentEmployee.print();
   }
}
```


Using Iterator-style Iterators

```
public void printEmployees (Employees emp)
{
   for (i = emp.employees(); i.hasNext(); )
    {
       Employee currentEmployee
           = (Employee) i.next ();
      currentEmployee.print();
   }
}
```

The Enumeration Interface

public boolean hasMoreElements();
public Object nextElement();

The old style Java external iterator convention

The Iterator Interface

public boolean hasNext();
public Object next ();
public void remove();

The new style Java external iterator convention

Iterators in Java include Collections, Streams, …

Variations on Iterator

1) Internal iterator - iterate inside the Aggregate
- easy to use, not as powerful as external iterator
- works best with closures (Inner Classes)

2) Combine next() and currentItem()

Smalltalk has Internal and External Iterators
Collections with do: and Streams
Java has also implemented these ideas

An Adapter on an Enumerator

```
class EnumerationAdapter implements Iterator
{
    private Enumeration e;
    public EnumerationAdapter(Enumeration e)
        { this.e = e;}
    public boolean hasNext() { return e.hasNextElement(); }
    public Object next() { return e.nextElement(); }
    public void remove() {}
}
```


Iterator and Composite

Composites usually have an iterator for their components.

Can make Iterator on Component that will iterate over all the components in a tree.

Internal Iterator is easy: here is method on Component:

```
public void preorder(Command c) {
    c.evaluate(this);
    Enumeration e = children();
    for (; e.hasMoreElements();  ) {
        ((Component) e.nextElement()).preorder(c);
    }
```

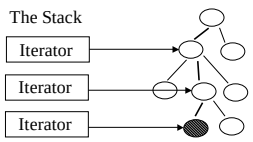

External Tree Iterator

Do a pre-order traversal.

Tree Iterator will have a stack of iterators, one for each ancestor of current node. The currentItem of each Iterator is the ancestor of the current node.

isDone is false for all iterators on the stack.

Tree Iterator is done when stack is empty.



External Tree Iterator

```
class TreeIterator {
   public next() {
         if (stack.isEmpty()) return;
         stack.push(stack.top().currentItem().children());
         while (stack.top().isDone()) {
                  stack.pop();
                  stack.top().Next();}}

   ElementType currentItem() {
         return stack.top().currentItem();}
```


Iterator and Visitor

Who is responsible for the traversal algorithm when you use Visitor and Composite?

- The components? (most common in C++)
- The visitor?
- A separate iterator?

1: The Components

If the component handles traversal, it looks like:

```
public Object accept(Visitor visitor) {
    visitor.visitA(this);
    for (Enumeration e  = children();
            e.hasMoreElements) {
            item = (Item) e.nextElement();
            item.accept(visitor);
```

Otherwise, it looks like

```
    public Object accept(Visitor visitor) {visitor.visitA(this);}
```

2: The Visitor

If the visitor handles iteration, it looks like:

```
public Object visitA(ComponentA  c) {
    // do something with c
    for (Iterator i = c.children();
            !i.isDone();  I.next())
    {
        ((Component) i.currentItem()).accept(visitor);
    }
}
```

Otherwise Visitor visitA just interacts with componentA.

3: An Iterator

If client calls Iterator, it looks like:

```
    visitor = new ConcreteVisitor.
    for (Iterator i = component.iterator();
            !i.isDone();  i.Next())
        {
            ((Component) i.currentItem()).accept(visitor);
        }
    }
```

Otherwise, the client looks like:

```
    ConcreteVisitor visitor;
    treeRoot.accept(visitor);
```


Tradeoffs

1: The Components
2: The Visitor
3: An Iterator

Highlight the tradeoffs and possibly look at a number 4 which is letting the client do it.

Review

New patterns: Interpreter, Iterator, Visitor

Patterns interact:

object can play different roles in different patterns

patterns can be alternatives

one pattern can set up another pattern

Next Session

Learn remaining patterns

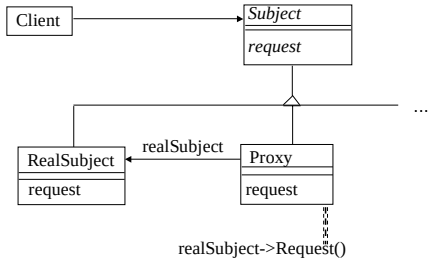
- Proxy
- Bridge
- Facade
- Flyweight
- Mediator

Remember intent of pattern so you can look it up when you need it.

Proxy

- Provide a surrogate or placeholder for another object to control access to it
  - represent an object in a remote address space
  - create expensive objects on demand
  - check access rights
- Proxy has same interface as “real subject”, and forwards operations to it

Proxy



Proxy

Remote proxy - first package arguments, then make remote procedure call.

Virtual proxy - compute objects, then forward request.

Protection proxy - check access rights, then forward request.

Dynamic Proxy Classes

Started in Java 2 1.3

- A *dynamic proxy class* is a class that implements a list of interfaces specified at runtime when the class is created
- A *proxy interface* is such an interface that is implemented by a proxy class
- A *proxy instance* is an instance of a proxy class

How to Hide Information

Hiding the classes in one module from another makes changes easier. Some things that help are:

- Abstract classes
- Builder (or Abstract Factory)

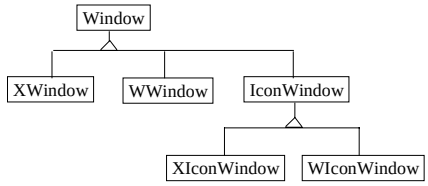
  Hide classes of products that will be used by other module in the builder.

  Example: window builder, code generator
- Adapter

  Hide class being used inside adapter
- Bridge and Facade

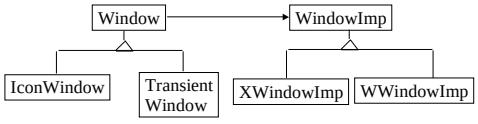
Bridge

What do you do if both an abstraction and its implementation vary?



Bridge

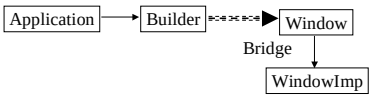
Decouple an abstraction from its implementation so that the two can vary independently.



Bridge and Builder

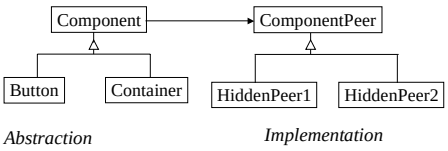
Use Builder to hide Window classes from application.

Use Bridge to hide platform classes from Builder.



Bridge in the AWT

The look of a component depends on the windowing system.

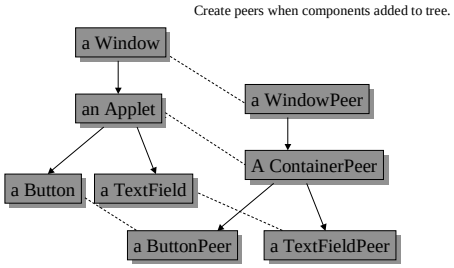


Standard Questions for Bridge

Where is the bridge set up?

When do we cross the bridge
(from abstraction to implementation)?

Building the Bridge



How a Button Creates a Peer

```
/**
    * Creates the peer of the button.  This peer allows us to
    * change the look of the button without changing its functionality.
    */
   public void addNotify() {
   peer = getToolkit().createButton(this);
   super.addNotify();
   }
```


How a List Creates a Peer

```
/**
    * Creates the peer for the list.  The peer allows us to modify the
    * list's appearance without changing its functionality.
    */
public void addNotify() {
   peer = getToolkit().createList(this);
   super.addNotify();
   synchronized (this) {
      visibleIndex = -1;
   }
}
```

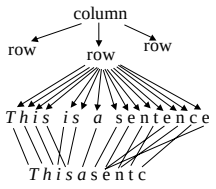
Flyweight

Use sharing to support large numbers of objects efficiently.

Separate intrinsic state (state stored in flyweight) from extrinsic state (state passed in as part of context). Minimize extrinsic state. Share flyweights that have the same intrinsic state.

Usually requires a factory that detects whether a flyweight exists with a particular instrinsic state and returns it.

Flyweights for Text

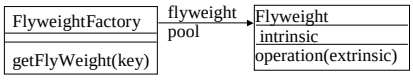


Client of flyweight, which holds context.

Context is location.

Flyweight pool

Flyweight



Flyweight class is usually abstract, with concrete subclasses that define the intrinsic state.

Flyweight for CAD

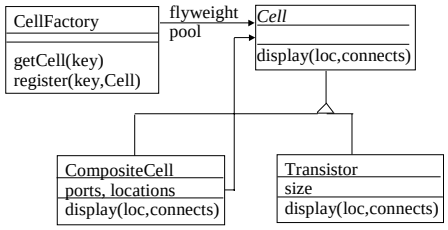
A VLSI design system must model millions of transistors.

This is only possible by sharing structure. Most transistors are part of larger structures (registers, NAND gates, RAM) that designers prefer to think about. Each kind of structure is called a *cell*.

Each cell is interconnected with other cells.

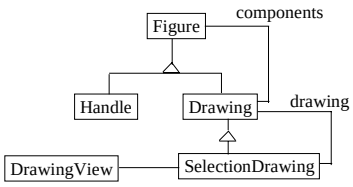
Context is location and interconnections.

Flyweight for CAD



Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

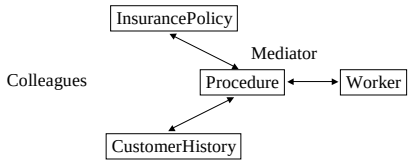


Mediator

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Example: Insurance policies must be approved before they are issued. There is a procedure (which can change over time, and which is different for different kinds of policies) for approving a policy. This procedure must interact with work queues of managers and with the history that is kept on the customer. Instead of putting this procedure in the insurance policy, put it in a separate object that is easy to change. (This is a "business process")

Mediator

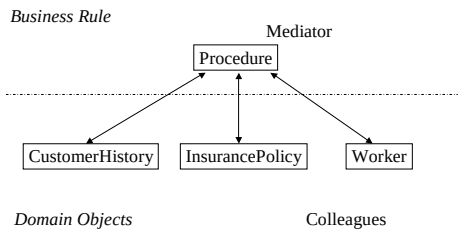


If interaction is main thing that changes, then make the interaction be an object.

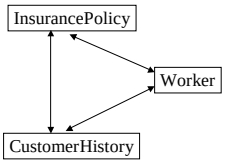
Colleague classes become more reusable.

Mediator is the non-reusable part.

Mediator



Not Mediator



Mediators

Are not reusable, but make other objects reusable

Used to glue together objects from a kit

Tend to be procedural, not object-oriented

Patterns Protect from Change

Rule: if something is going to change, make it an object.

- Strategy: make algorithm an object so it can change
- State: make state-dependent behavior an object so it can change
- Iterator: make the way you iterate over an aggregate an object so it can change
- Facade: make a subsystem an object so it can change
- Mediator: make the way objects interact an object so it can change
- Factory: make the classes of your products an object so it can change

Summary

You have now seen all of the Design Patterns

You should be able to recognize Abstract Factory, Adapter, Bridge, Builder, Chain of Responsibility, Command, Composite, Decorator, Façade, Factory Method, Flyweight, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, Prototype, Proxy, Singleton, State, Strategy, Template Method, Visitor…

Bird on Patterns

Learn the patterns and then forget 'em

-- Charlie Parker

http://www.hillside.net

Silver Buckshot

There are no silver bullets
…...Fred Brooks

But maybe some silver buckshot…

- Objects
- Frameworks
- Patterns
- Architecture
- Process/Organization
- Tools

UIUC Patterns Group
Software Architecture Group
Ralph Johnson's Group

- Objects
- Reuse
- Frameworks
- Adaptive Architecture
- Components
- Refactoring
- Evolution
- Patterns

Our Perspective

Objects, Patterns, Frameworks, and Refactoring really do work, and can lead to the production of better, more durable, more reusable code

To achieve this requires a commitment to tools, architecture, and software evolution, and to people with superior technical skills and domain insight

Next Session

You will be able to
- find new patterns
- learn new patterns

We'll also talk about writing patterns.

Other Patterns

Claim: people always use patterns to solve problems

Corollary: there are a lot of software patterns besides object-oriented design patterns!

patterns for user interface design
patterns for distributed programming
patterns for checking user input
patterns for analysis
patterns for how to manage a software project

User Interface Patterns

Ward Cunningham and Kent Beck
http://c2.com/cgi-bin/wiki?HistoryOfPatterns

- Window per Task
- Few Panes
- Standard Panes
- Nouns and Verbs
- Short Menus

Pattern Language

Set of patterns that tell you how to build something.

Complete -- all the patterns you need.

One pattern leads to another -- language gives order to consider them.

Mapping Objects To Persistence Pattern Language

Pattern Name	Description
Persistent Layer	Provide a layer for mapping your objects to the RDBMS or other data source.
CRUD	All persistent object need, at a minimum, create, read, update, and delete operations.
SQL Code	Defines the actual *SQL Code* that takes the values from the RDBMS or other data source and retrieves them for the object's use and vice-versa. It is where you define the *CRUD* operations.
Attribute Mapping Methods	Maps the values between the database values and attributes. This pattern also handles complex object mappings. Populates the object(s) with the row values.
Type Conversion	Works with *Map Attributes* to translates values from the database to the appropriate object types and vice-versa. Insures data integrity.

Mapping Objects To Persistence Pattern Language

Pattern Name	Description
Change Manager	Keeps track of when an object's values have been changed for maintaining consistency with the database. It determines the need to write the values to a database table or not.
OID Manager	Generates Unique Keys for the Object Ids during an insert.
Transaction Manager	Provides a mechanism to handle transactions while saving objects.
Connection Manager	Gets and maintains a connection to the database.
Table Manager	Manages the mappings from an object to its database table(s).

Joseph W. Yoder, Ralph Johnson and Quince Wilson ***Connecting Business Objects to Relational Databases*** Fifth Conference on Patterns Languages of Programs (PLoP '98) Monticello, Illinois, August 1998. Technical report #wucs-98-25, Dept. of Computer Science, Washington University. Department of Computer Science, September 1998.

Security Patterns

Pattern Name	Description
Single Access Point	Providing a common security module and a single way to log into the system.
Check Point	Organizing security checks and their repercussions.
Roles	Organizing users with similar security privileges.
Session	Localizing global information in a multi-user environment.
Limited View	Allowing users to only see what they have access to.
Full View with Errors	Allowing users to see everything and generate errors.
Secure Access Layer	Integrating application security with low-level security.

Joseph W. Yoder and Jeffrey Barcalow ***Architectural Patterns for Enabling Application Security*** Fourth Conference on Patterns Languages of Programs (PLoP '97) Monticello, Illinois, September 1997. Technical report #wucs-97-34, Dept. of Computer Science, Washington University Department of Computer Science, September 1997. Pattern Languages of Programs Design 4 edited by Neil Harrison, Brian Foote, and Hans Rohnert. Addison Wesley, 2000

Analysis Patterns

David Hay, Data Model Patterns: Conventions of Thought
Dorset House Publishing, 1996 ISBN 0-932633-29-3
Martin Fowler, Analysis Patterns, Addison-Wesley, 1997

Organizational structure	Hay, Fowler
Accountability	Fowler
Quantities	Hay, Fowler
Contracts	Hay, Fowler
Accounting	Hay, Fowler
Products and Inventories	Hay
Material Requirements Planning	Hay

How Patterns Fit Together

- Some patterns naturally fit together
- Real designs use many patterns
- Add patterns to design one or two at a time
- One pattern leads to another
- Some patterns are alternatives
- Some patterns have similar contexts
- You can document a system by a sequence of design patterns, representing the sequence of decisions you made.

Using Patterns in Documentation

How do you tell which patterns are in a design?
- use names to give hints
- describe design as a sequence of patterns
- include in CASE tool

Methods and Patterns

Patterns fill a hole ignored by analysis and design methods.

Methods give language for modeling, patterns give models.

Patterns are a layer on top of methods.

But patterns tell you what to do, too. Does this contradict method?

Methods vs. Patterns

Methods try to be general-purpose, patterns are specific.

Methods try to be domain independent, patterns are often domain dependent.

Different communities; people working on patterns tend to be developers who do not use any particular method.

Will methods grow to include patterns, or will patterns engulf methods?

What Can be a Pattern?

Pattern Languages of Program Design (edited by Coplien and Schmidt, Addison-Wesley, 1995, ISBN 0-201-60734-4) has:

- How to make clients in client/server (Wolf and Liu)
- Distributed programming (DeBruler, Aarsten et. al., Meszaros, Berczuk, Schmidt, Ran)
- Decision support systems (Peterson)
- Software process (Coplien, Whitenack, Foote and Opdyke)
- Going from analysis to design (Kerth)
- Standard architectures (Edwards, Meunier, Mularz, Shaw)

Patterns

Let us describe our practices and let others criticize them.

Make it easier to teach software development.

Makes it easier to see when our techniques are no longer applicable.

Are hard to write.

Writing Patterns

You should write patterns because

- you will learn a lot about patterns
- you probably use some patterns that haven't been documented yet
- you meet a lot of good people that way

But writing is hard work, and not everybody has the time or the desire to do it.

Finding Patterns

Patterns come to those who wait -- must have time for reflection.

Patterns come to those who are prepared -- must have experience in domain of problem.

Patterns are refined in fire -- must have readers who criticize.

It is not a pattern until you have more than one example!

How to Find Patterns

Look for a solution and document it.

What is the problem? When should you use the solution?

Why don't you use it all the time?

What are the drawbacks of the solution?

Writers' Workshop

Excellent way to get feedback on pattern.

Author is silent while group discusses pattern. Group pretends author is not there.

Strong moderator ensures that discussion is positive.

Say what you like before you say what you don't like.

Discuss both form and content.

How to Learn New Patterns

Get a set of patterns.

Meet regularly to discuss them with a group. (Brown-bag lunch works well)

Group is best so you develop shared vocabulary.

Use the vocabulary in design reviews and design sessions.

How to Learn New Patterns

A pattern is usually hard to understand if you don't need it and have never used it. Don't worry, just get the big picture.

Learn what patterns are available, then study the pattern when you need it.

It isn't hard!

Further information

http://hillside.net

Pointers to mailing lists, books, ftp archives, on-line patterns, conferences, etc.

gang-of-4-patterns-request@cs.uiuc.edu
patterns-request@cs.uiuc.edu
patterns-discussion-request@cs.uiuc.edu

Frameworks and Patterns

Frameworks are a kind of pattern.

Frameworks contain Design Patterns.

Compared to Design Patterns, frameworks are
- more concrete
- more domain specific

Design Patterns vs. Frameworks

Design patterns are more abstract
- Frameworks are represented by programs, patterns are illustrated by programs.
- Frameworks are specialized to particular domain.

Frameworks contain design patterns

Design patterns are easier to learn

Frameworks have bigger payoff

Problems with Frameworks

Frameworks are hard to buy:
- Most are proprietary
- You can buy frameworks for GUI, distribution, or persistence, but not for accounting, real-time control, or scheduling

Frameworks are hard to learn:
- Many objects working together
- Design patterns make it easier

Frameworks are hard to make:
- Require experience
- Require iteration

Conclusion

Reuse is capital intensive
- Must acquire assets
- Must learn assets

Patterns are cheaper to use than frameworks, and good preparation for frameworks. Frameworks have higher payoffs.

Developing reusable assets is very expensive.

Buy if you can.

Evolution of Object-Oriented Systems

Objects are good abstractions
- put data and behavior together
- Early reuse through subclassing and copying/pasting
- Nouns are objects / Verbs are actions

Patterns come into play with experience
- More reuse through “pluggable” components
- Action / Strategies can be objects as well

Evolution of Object-Oriented Systems

Frameworks evolve as your code becomes more reusable
- White Box vs. Black Box
- Action / Strategies can be objects as well
- Refactoring and Testing becomes very important

Adaptive Object-Models
http://www.adaptiveobjectmodel.com
- Metadata (descriptive data) allows you to evolve the program without writing new code
- Can very quickly adapt to new business rules

Adaptive Object-Models

Separates what changes from what doesn’t.

Architectures that can dynamically adapt to new user requirements by storing descriptive (metadata) information about the business rules that are interpreted at runtime.

Sometimes called a "reflective architecture" or a "meta-architecture ".

Highly Flexible – Business people (non-programmers) can change it too.

PLoP Conferences

www.hillside.net

Summary

You have been introduced and should recognize many design patterns: Abstract Factory, Adaptor, Bridge, Builder, Chain of Responsibility, Command, Composite, Decorator, Façade, Factory Method, Flyweight, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, Prototype, Proxy, Singleton, State, Strategy, Template Method, Visitor

You should also be familiar with some other types of patterns and how to find more information about them.

- Analysis Patterns
- Architectural Patterns
- Coding Patterns
- GUI Patterns
- Process Patterns
- Persistence Patterns
- Security Patterns
- …

"Patterns Generate Architecture"

That's All