Design Patterns
"In Java"

Joseph W. Yoder

The Refactory, Inc.

www.refactory.com

joe@refactory.com

 Day 1 -- Slide 1

The Refactory, Inc.

The Refactory, Inc. was founded in 1998 as a consortium of object-oriented experts dedicated to helping organizations succeed with objects.

Founders and Affiliates have a total of over 120 years of combined software development experience with over 80 years dedicated to Object-Oriented development.

 Day 1 -- Slide 2

The Refactory Principals

John Brant
Brian Foote
Don Roberts
Ralph Johnson
Joe Yoder

Refactory Affiliates

Dragos Manolescu
Brian Marick
Bill Opdyke

 Day 1 -- Slide 3

The Refactory, Inc.

The Refactory principles 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++, 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).

 Day 1 -- Slide 4

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

 Day 1 -- Slide 5

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.

 Day 1 -- Slide 6

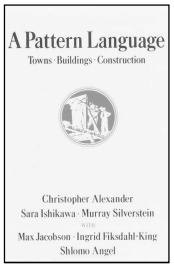
Patterns

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*



 Day 1 -- Slide 7

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.

 Day 1 -- Slide 8

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

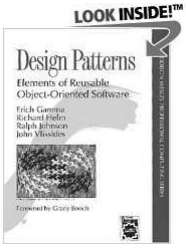
A landmark book that changed the way programmers think about building object-oriented programs

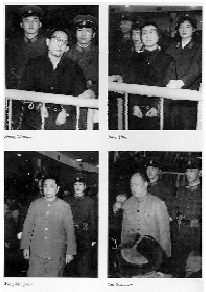
Repeating organization of classes (objects)

Design Patterns: Elements of Reusable Object-Oriented Software

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

Addison-Wesley, 1995. Gang of Four (GOF)





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

 Day 1 -- Slide 11

Class Overview

- Presentation
- Reading Groups
- Exercises

 Day 1 -- Slide 12

Patterns in Java and

Java libraries and frameworks were influenced by the Design Patterns from GoF

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

Java has features that affect implementation of the design patterns

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

Outline of Course

- Day 1

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

  More Patterns –Decorator, NullObject, Strategy, Memento, Observer, State

- Day 2

  How patterns work together

  Abstract Factory, Adapter, Builder, Factory Method, Prototype, Singleton

  Documenting system designs with patterns

  How patterns work together

  Centralized vs. distributed - Interpreter, Visitor, Iterator

Outline of Course

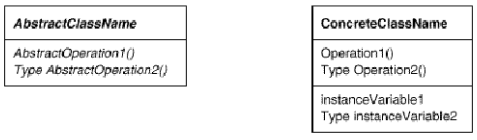
- Day 3
  - The rest of the Design Patterns
    - Bridge, Proxy, Command, Facade, Flyweight, Mediator
  - Frameworks vs Patterns
  - Other kinds of patterns
  - Learning about patterns

Notation

Design Patterns Book uses OMT

We use this to show the correlation

Sometimes we use UML which is similar

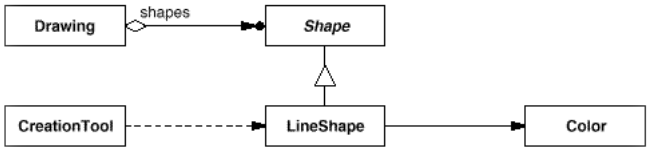


(a) Abstract and concrete classes

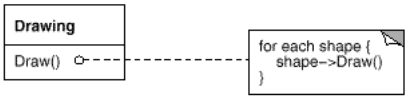
(b) Participant Client class (left) and implicit Client class (right)

More Notation

Class Diagrams:



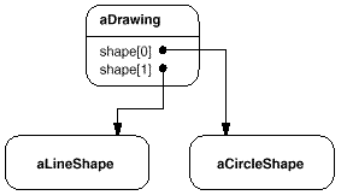
(c) Class relationships



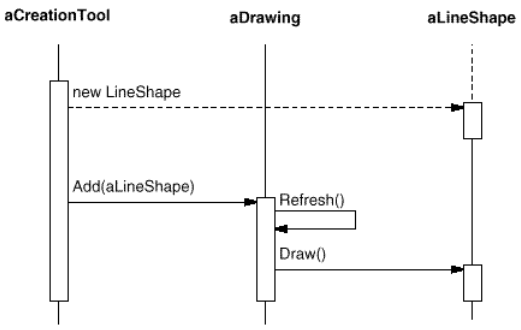
(d) Pseudocode annotation

Yet More Notation

Object Diagrams:



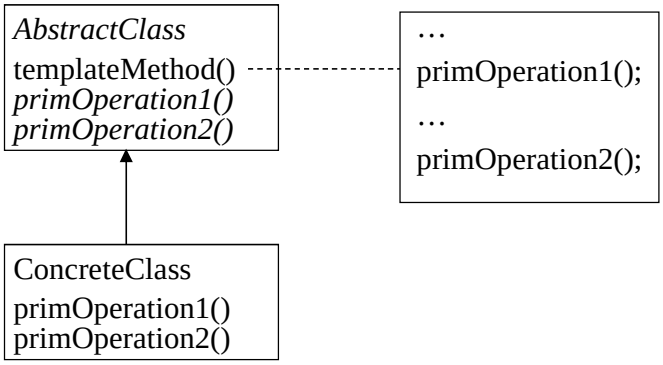
Interaction Diagrams:



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();
            resetFocus();
    }
}
public class ButtonView extends View {
    public void doDisplay() {
            setButtonWidth();
            …}
}
```

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


Composite

Context:

Developing OO software

Problem:

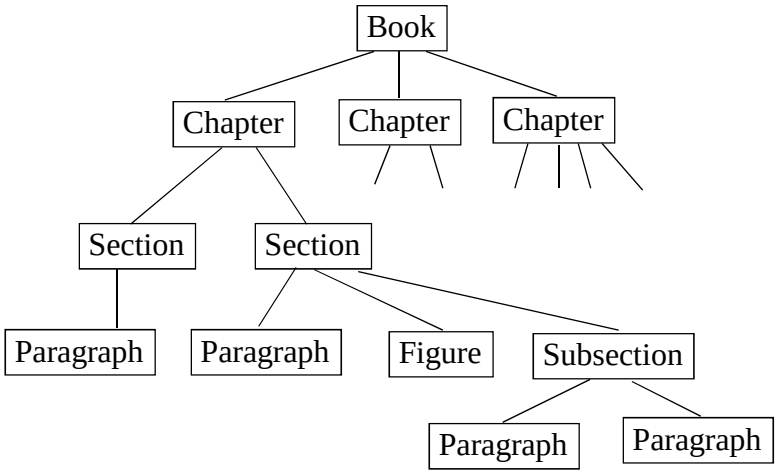
Complex part-whole hierarchy has lots of similar classes.

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

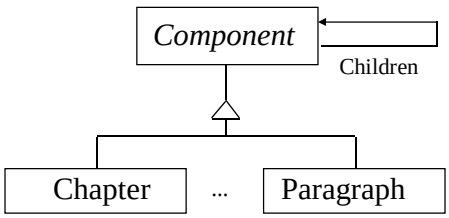


 Day 1 -- Slide 23

Composite

Idea: make abstract "component" class.

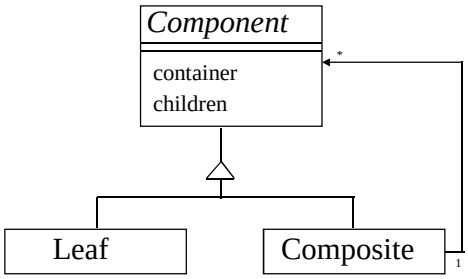
Alternative 1: every component has a (possibly empty) set of components.



Problem: many components have no components

 Day 1 -- Slide 24

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:

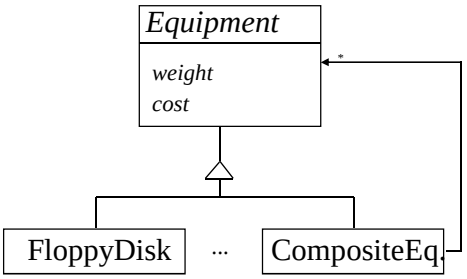
Only public operations that change container are addComponent/removeComponent

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;
}
```

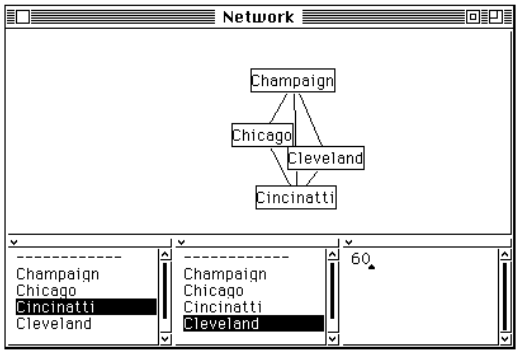

Example: Equipment



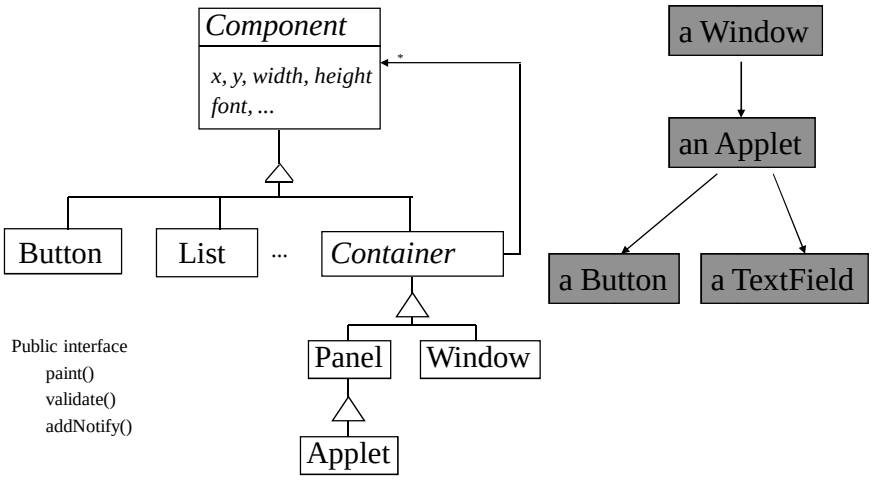

```
class CompositeEquipment {
    int weight() {
            int total = 0; Equipment item;
            for (Enumeration e = children() ; e.hasMoreElements();
                    item = (Equipment) e.nextElement())
            {
                total += item.weight();
            }
            return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

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.

Examples:

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

Chain of Responsibility

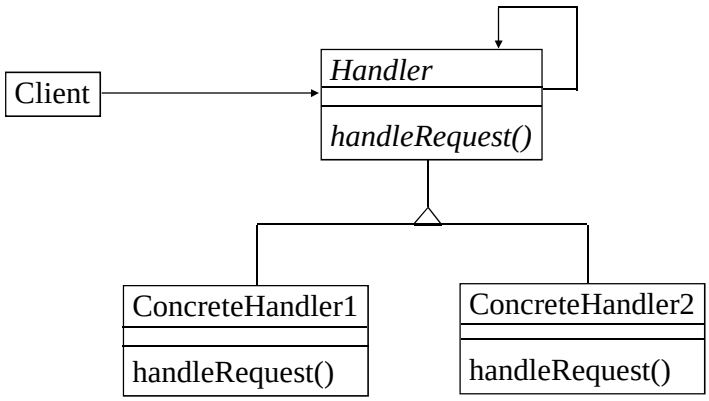
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Usually found with Composite - chain of parents.

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

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


Chain of Responsibility



 Day 1 -- Slide 37

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

 Day 1 -- Slide 38

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

 Day 1 -- Slide 43

GoF Design Patterns

Numbers are the day that the pattern is covered/used.

Creational patterns	
Abstract factory	2
Builder	2, 3
Factory method	2
Prototype	2
Singleton	2
Structural patterns	
Adapter	2, 3
Bridge	3
Composite	1, 2
Decorator	1
Facade	3
Flyweight	3
Proxy	3
Behavioral Patterns	
Chain of Responsibility	1
Command	3
Interpreter	2
Iterator	2
Mediator	3
Memento	1
Observer	1
State	1, 3
Strategy	1, 3
Template Method	1
Visitor	2

 Day 1 -- Slide 44

Goals of this session

Learn Decorator, Null Object, Strategy

Start to learn how patterns work together

- Patterns often share the same context.
- Problems produced by one pattern are sometimes resolved by another.
- A complex design consists of many patterns.

Decorators

"Reading Group"

Decorators add a responsibility to an object by

- making the object a component
- forwarding messages to component and handling others

Possible examples from Java

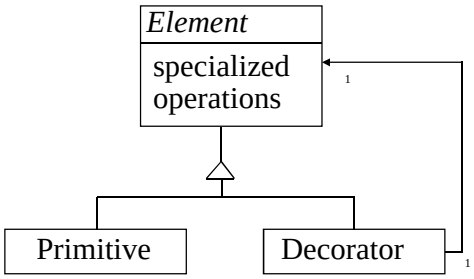
Double, Integer, Float, etc.

Decorators add an attribute to an object.

Decorator forwards operations to the component.

Component gets values from its decorator.

Decorator Structure



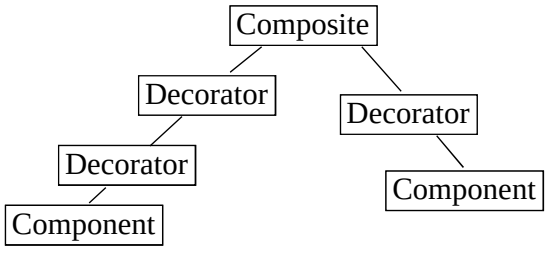
Decorator forwards most operations to the object it is decorating.

Decorator

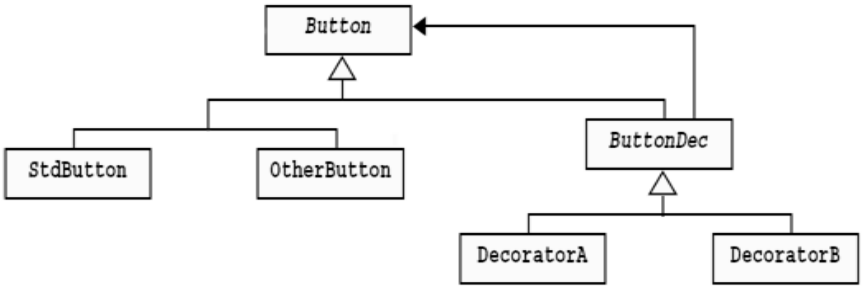
Decorators add an attribute to an object.

Decorator forwards operations to the component.

Component gets values from its decorator.



Decorator Example



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()

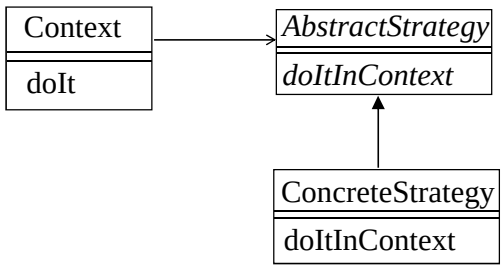
- case OS of:

'NT' : setButtonShadowWidth: 10;

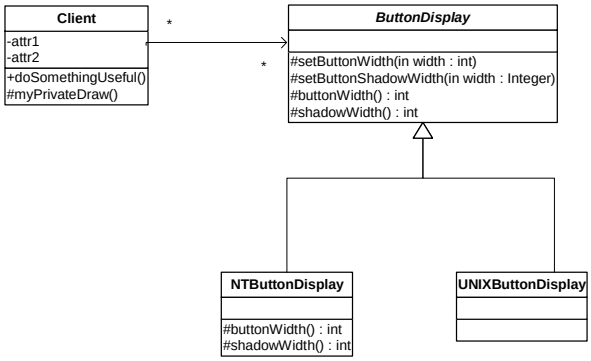
'UNIX': setButtonShadowWidth: 15;

...

Strategy



Strategy
(an example)



 Day 1 -- Slide 53

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

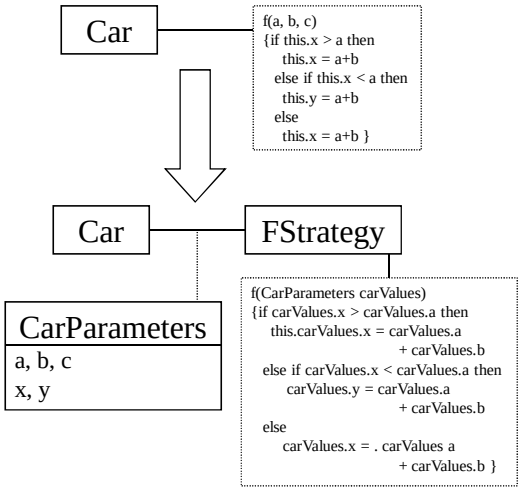
```
class X {
   int f(A anA, B aB){
         return (anA.size + size) / aB.size;
   } ...
class B {
   int f(A anA, X anX){
         return (anA.size + anX.size) / size;
   } ...
```

 Day 1 -- Slide 54

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….you would have to use pointers in C++.



 Day 1 -- Slide 55

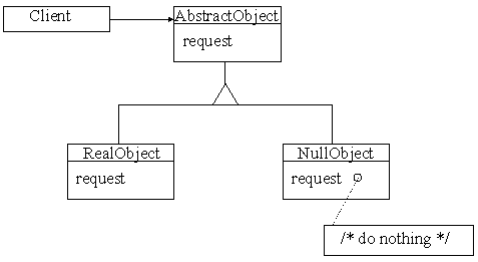
NullObject

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:



 Day 1 -- Slide 56

NullObject
"Example"

```
public class Person {
    private Name name;
    private Address address;
    private Phone phone;
    public void printDetails() {
        if (name == null) this.print("");
        else name printDetails;
        if (address == null) this.print("");
        else address printDetails;
        if (phone == null) this.print("");
        else phone printDetails;
        …}
…}
```

```
public class NullObject {
    public void printDetails() {
        this.print("");}
…}
public class Person {
    private Name name;
    private Address address;
    private Phone phone;
    public void printDetails() {
        name printDetails;
        address printDetails;
        phone printDetails;
        …}
```


Goals of Next Session

Learn State and Observer (Listeners)

Learn Memento

See more about how Patterns work together

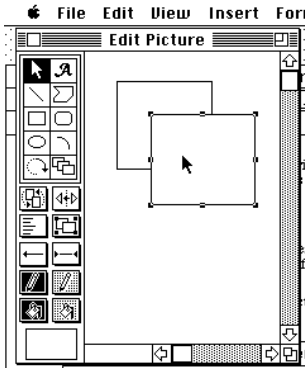
State Pattern

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.

Toolbar State Example



State

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

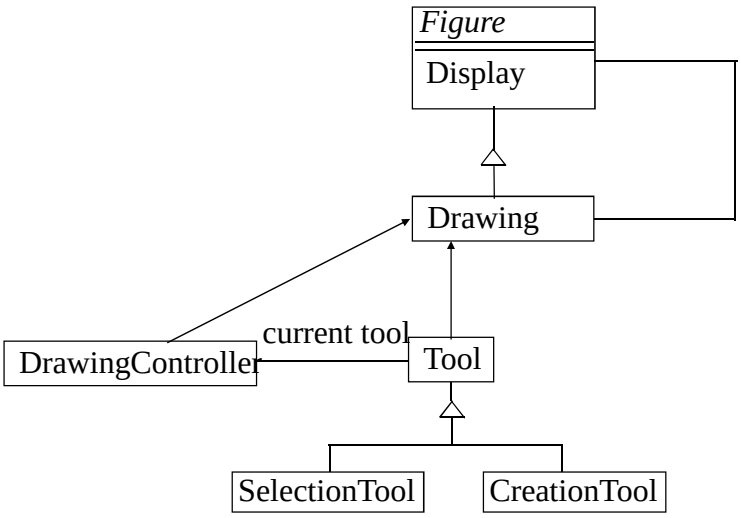
Tools are the "current state" of the DrawingController; it delegates operations to the current state.

Make a class hierarchy of Tools.

DrawingController points to its "current Tool".

 Day 1 -- Slide 61

State

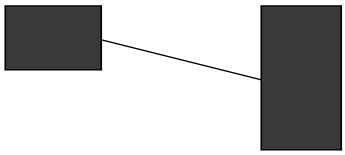


 Day 1 -- Slide 62

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.

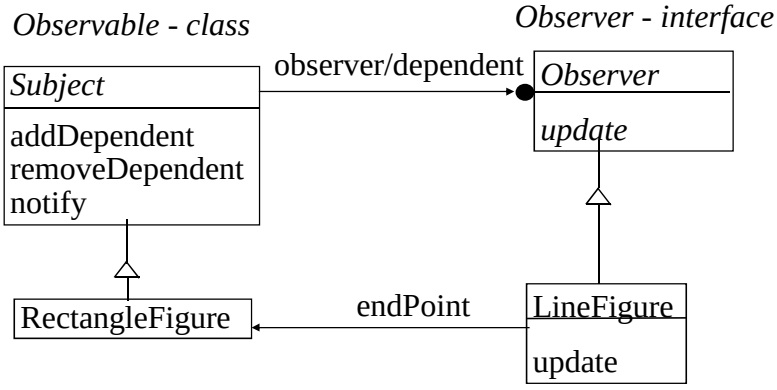
Example: Graphics system - moving box causes connecting lines to move.



 Day 1 -- Slide 63

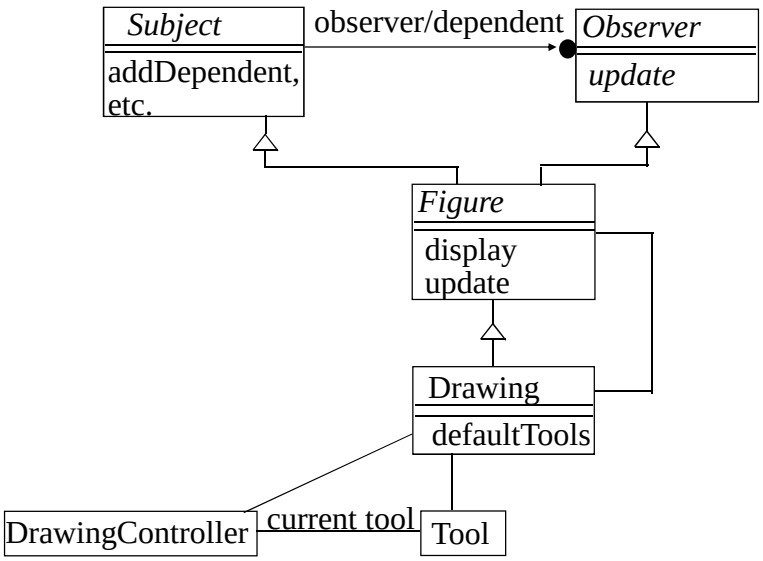
Observer Pattern

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



 Day 1 -- Slide 64

Observer Pattern



 Day 1 -- Slide 65

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

 Day 1 -- Slide 66

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)

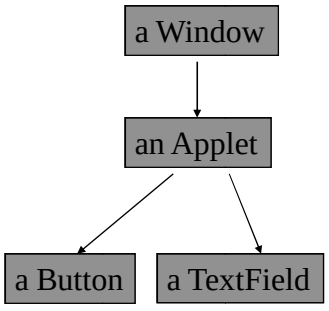
Events in AWT 1.1

Applet is a "Listener"

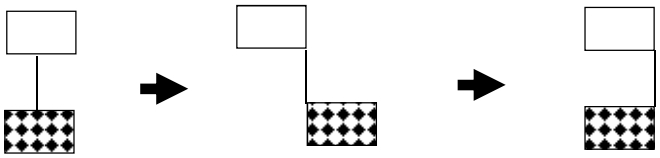
Button has methods
- addActionListener()
- processActionEvent()

Applet registers with Button.

When Button processes action event, it calls applet



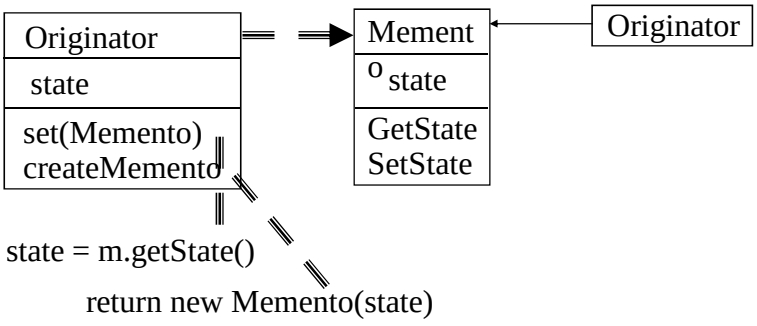
Memento



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.



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

Conclusions

Patterns: solutions to recurring problems

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

Stretch from design to code

We have seen: Chain of Responsibility, Composite, Decorator, Memento, Observer, NullObject, State, Strategy, Template Method

