

---

# Design

Methods  
Patterns

# Method

---

Method: A systematic  
way of doing  
something.

Concepts  
Notations  
Process  
Heuristics

(a book)

## Responsibility Driven Design

---

### Concepts

Class  
Responsibility  
Collaboration  
Inheritance  
Subsystem  
Contracts

Book: Object-  
Oriented Design by  
Wirfs-Brock,  
Wilkerson and  
Wiener

## Object Modeling Technique (OMT)

---

### Concepts

Objects  
Classes  
Associations /  
Cardinality  
Attributes  
Inheritance  
Events

Book: Object-  
Oriented Modeling  
and Design

by Rumbaugh, Blaha,  
Premerlani, Eddy and  
Lorenson

# UML

---

Unified Modeling Language

Booch, Rumbaugh, Jacobson

UML = concepts and notation

Unified Process = UML + a process

# Objects and Associations

---

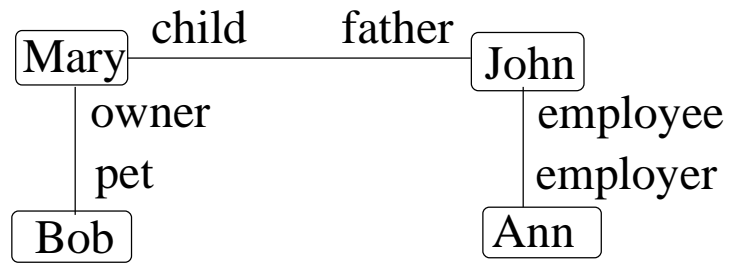
John is Mary's father. Mary is John's daughter.

Bob is Mary's dog. Mary is Bob's owner.

Ann is John's employer. John is Ann's employee.

## Objects and Associations

---

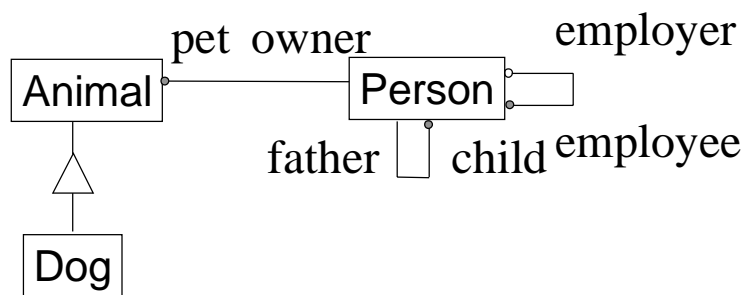


*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

299

## Classes and Associations

---



*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

300

## Objects and Attributes

---

John's name is "John Patrick  
O'Brian".

John's age is 27.

John's address is 987 N. Oak St,  
Champaign IL 61820

What about John's employer?  
John's wife?

## Objects and Attributes

---

John

name: "John

Patrick O'Brian"

age: 27

address: ...

# Objects and Behavior

---

John can earn money by working for his employer.

John fills out time-cards.

John takes vacations.

Employer gives John a paycheck.

John spends money to gain more possessions.

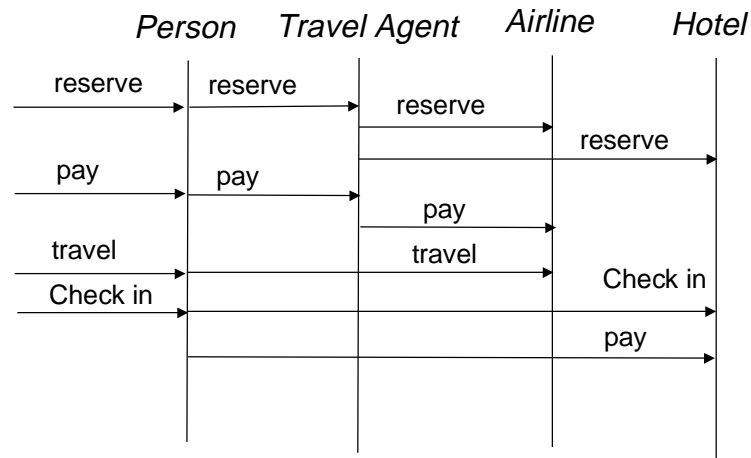
---

John

fill out time-cards  
take vacations  
receive paycheck  
buy/spend  
make reservations  
travel  
check in

## Event Diagrams

---



*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

305

## What Really is an Object?

---

Reservation -- a promise to give service to a customer

Ticket -- record that customer has paid for service in advance

Flight

Payment -- an event (transaction?) in which money is exchanged

*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

306

---

In fact, anything we can talk about can be an object, including associations ("the husband of the first party", "first-born son").

The *important* question is what we are trying to model.

Models should be as simple as possible, but no simpler.

## Why this is important

---

Many of the objects in a good design are NOT problem domain objects.

They can represent operations, processes, states, constraints, relationships, ...

## Design for Reuse

---

Purpose is to make a system flexible, extensible, and easily changed.

Make things objects if they need to be changed and manipulated.

## Design for Reuse

---

Goal is to build an application just by using preexisting objects.

- Compose, don't program.

Relationship between objects as important as class hierarchy.

## To Be or Not To Be --- A Class

---

Analysis model shouldn't contain implementation-specific objects like queues and lists.

Classes should be

- unique
- relevant
- specific

## To Be or Not To Be --- A Class

---

Operations should not be objects unless you need to manipulate them.

- A Call in a telephone billing system has attributes such as date and destination phone number, and must be stored so it can be printed on the monthly bill.

---

The name of a class should represent its intrinsic nature and not a role that it plays in an association.

- A person can have many roles when associated with a car: owner, passenger, driver, lesee.

## What is an OO Model?

---

A OO model is a model expressed in terms of objects, classes, and the relationships between them.

Models can be written in programming languages, graphics, or English.

## OMT

---

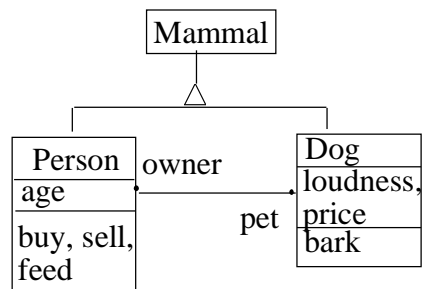
graphical OO modeling notation with three parts

- object model
- dynamic model (event trace/state machines)
- functional model (data flow diagram)

## Object Model

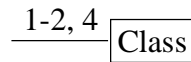
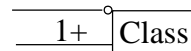
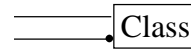
---

Object, classes,  
attributes,  
operations,  
associations,  
inheritance



# Multiplicity of Associations

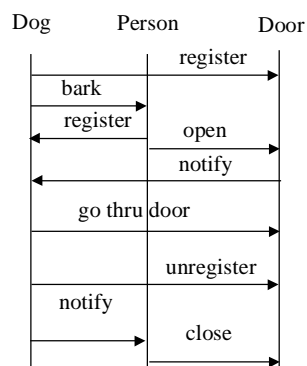
- exactly one
- zero or more
- zero or one
- one or more
- other



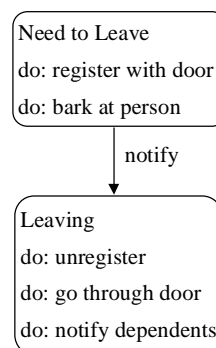
# Dynamic Model

Record order of events, dynamic interaction between objects.

Event Trace



State Machine



## Design vs. Analysis

---

Use of objects integrates design and analysis.

- advantages - easier to move from one to the other
- disadvantages - harder to tell which you are doing

## Analysis vs. Design

---

Analysis produces documents that customer can understand.

Design produces documents that programmers can understand.

# Design

---

Design must be implementable.

- Include detail not needed in analysis model.
- Doesn't include objects outside computer.

# OMT Process

---

Start with requirements

Build object model

- identify objects and classes
- prepare a data dictionary
- identify associations, attributes
- organize using inheritance
- refine, reorganize

(continued)

---

Build dynamic model

- prepare scenarios
- identify events
- build event trace
- build state diagram
- analyze for consistency

RDD Process

---

Start with partial requirements

Brainstorm for objects

Initial design

- Iterate until you find all the objects, responsibilities, and collaborations

Reorganize using inheritance and divide into subsystems

## RDD Initial Design

---

Repeat until nothing is missing.

- Make scenario
- Add missing objects
- Add missing responsibilities (attributes and operations)
- Add missing collaborations (associations)

## Heuristics

---

Model the real world.

Nouns are classes, verbs are operations on classes.

Eliminate cycles of dependences between classes.

Hide implementation details.

A class should capture one key abstraction.

## Heuristics

---

Minimize the number of messages in a class.

Minimize the number of classes which which another class collaborates.

Minimize the amount of collaboration between a class and its collaborator.

## Patterns

---

All designers use patterns.

Patterns in solutions come from patterns in problems.

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

## Christopher Alexander -- A Pattern Language

---

"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."

## 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

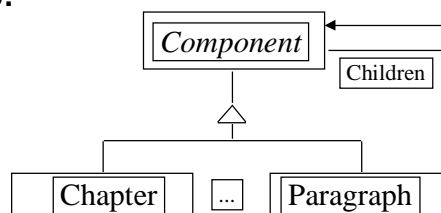
# Composite

---

Idea: make abstract "component" class.

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

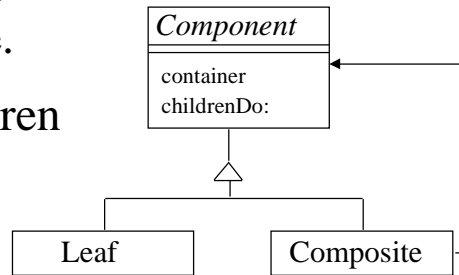
Problem: many components have no components.



# Composite Pattern

Composite and Component have the exact same interface.

- enumerating children
- childrenDo: for Component does nothing
- only Composite adds removes children.



*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

333

## Two design alternatives for Part-of

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.

*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

334

## Part-of

---

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

---

The public operations on components and composites are:

- Composite can enumerate components.
- Component knows its container.
- Add/remove a component to/from the composite.

---

The operation to add a component to a composite updates the container of the component. There should be no other way to change the container of a component.

In C++, make component friend of composite.

Smalltalk does not enforce private methods.

---

*private*

**container: anObject**

container := anObject

*accessing*

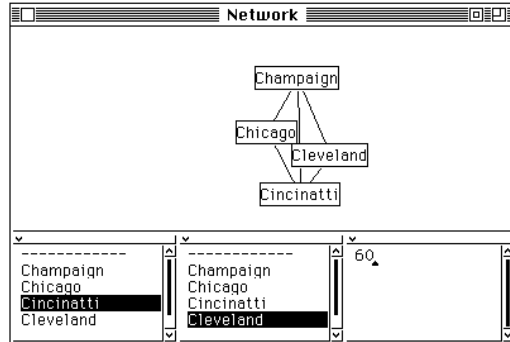
**addComponent: aComponent**

components add: aComponent

aComponent container: self

## Example: Views and Figures

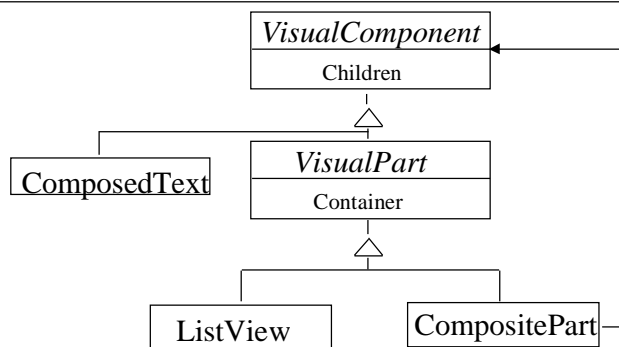
Big window can contain smaller windows.



*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

339

## Composite Pattern in VisualWorks



Most operations in CompositePart iterate over the children and repeat the operation on them.

*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

340

## CompositePart

---

```
addComponent: aVisualComponent
self isOpen
  ifTrue: [ ... ]
  ifFalse: [components addLast:
aVisualComponent.
          aVisualComponent container: self]
```

*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

341

```
displayOn: aContext
  "Display each component."
  | clipBox |
  clipBox := aContext clippingBounds.
  components do:
    [:each | (each intersects: clipBox)
              ifTrue: [each displayOn: aContext
copy]]
```

## Results of a Pattern

---

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

Details of implementing pattern depend on language.

Pattern is often not obvious.

Pattern can be applied to many kinds of problems.

## Kinds of Patterns

---

OO Design Patterns

- Design Patterns: Elements of Reusable O-O Software

Distributed/Concurrent Programming Patterns

User Interface Design Patterns

Architectural Patterns

Software Process Patterns

## Parts of a Pattern

---

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

Examples

- are proof of pattern-hood

## Patterns Work Together

---

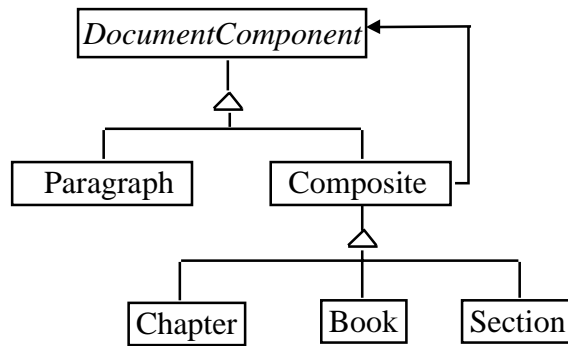
Patterns often share the same context.

Problems produced by one pattern are sometimes resolved by another.

A complex design consists of many patterns.

# Book Example

---



*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

347

# A book

---

Book  
  Chapter  
    Section  
      Paragraph  
      Paragraph  
    Section  
      Paragraph  
  Chapter  
    Section  
      Paragraph

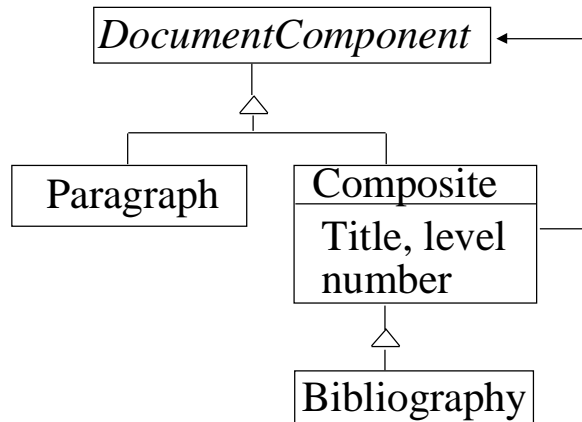
Composite classes  
are simple.  
Too many Composite  
classes.

*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

348

# Distinguishing Components by Attributes, Not Classes

---



Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

349

---

Composite (book title, level 1)  
  Chapter (chapter title, level 2, #1)  
    Composite (section title, level 3, #1)  
      Paragraph  
      Paragraph  
    Composite (section title, level 3, #2)  
      Paragraph  
  Chapter (chapter title, level 2, #2)  
    Composite (section title, level 3, #1)  
      Paragraph

Problem:  
  numbering  
  sections

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

350

## Strategy Pattern

---

A strategy is an algorithm represented as an object.

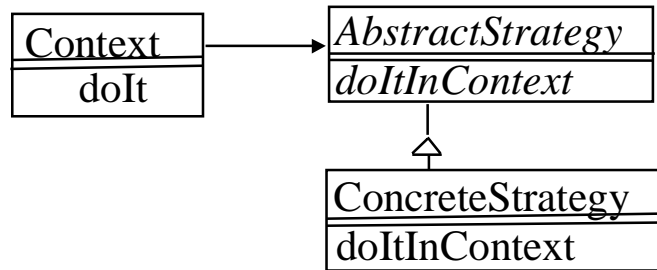
Not normal. Only use when there is a good reason.

## Advantages of Strategy

---

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

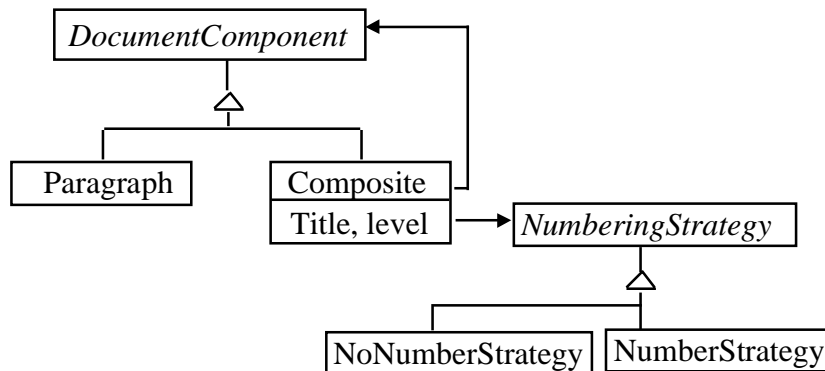
# Strategy



Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

353

# Design with Strategy



Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

354

# Decorators

---

Decorators add an attribute to an object by

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

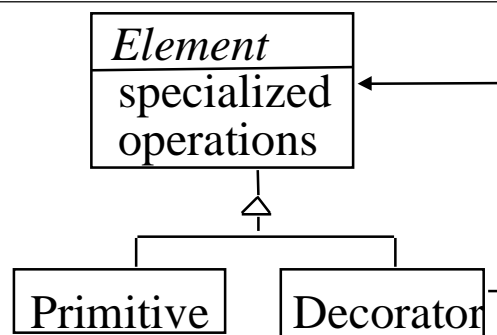
---

Example from VisualWorks: Wrapper is a subclass of VisualComponent that is a decorator.

- 
- TranslatingWrapper adds an offset.
  - BorderedWrapper adds a border and inside color
  - EdgeWidgetWrapper adds a collection of EdgeWidgets

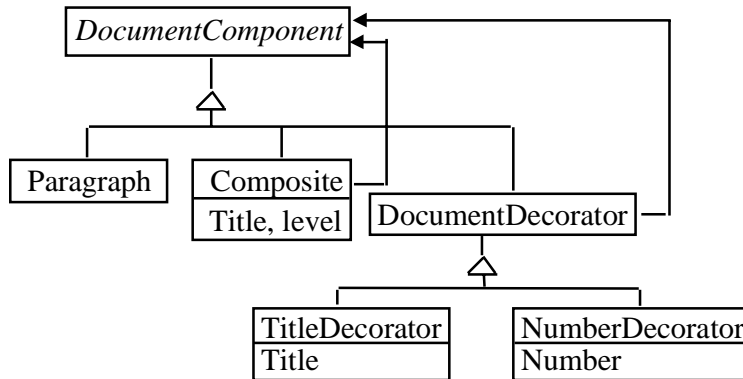
## Decorator Pattern

---



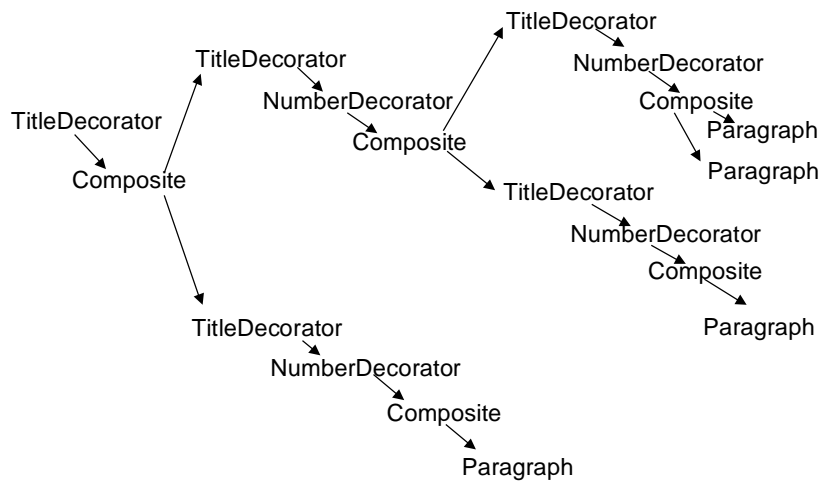
Decorator forwards most operations to the object it is decorating.

# Document with Decorators



Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

359



Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

360

# 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?

# Prototype

---

Solution: create a new object by copying an old one. If object is a composite or decorator then its entire substructure is copied.

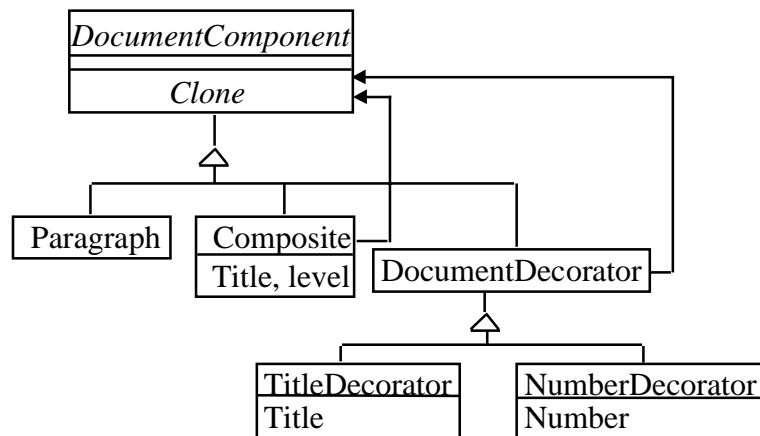
# Prototype

---

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

---



## Conclusion

---

Methods provide language.

Patterns provide content.

Patterns and methods complement each other, and patterns provide something that methods have been missing.