An Introduction to Design Patterns

John Vlissides

IBM T.J. Watson Research

vlis@watson.ibm.com

Overview

Part I: Motivation and Concept

- the issue
- what design patterns are
- what they're good for
- how we develop and categorize them

Overview (cont'd)

Part II: Application

- use patterns to design a document editor
- demonstrate usage and benefits

Part III: Wrap-Up

- observations, caveats, and conclusion

Part I: Motivation and Concept

OOD methods emphasize design notations

Fine for specification, documentation

But OOD is more than just drawing diagrams

Good draftsmen $\neq$ good designers

Good OO designers rely on lots of experience

At least as important as syntax

Most powerful reuse is *design* reuse

Match problem to design experience

OO systems exploit recurring design structures that promote

- abstraction
- flexibility
- modularity
- elegance

Therein lies valuable design knowledge

Problem: capturing, communicating, and applying this knowledge

A *Design Pattern*

- abstracts a recurring design structure
- comprises class and/or object
  - dependencies
  - structures
  - interactions
  - conventions
- names & specifies the design structure explicitly
- distills design experience

A Design Pattern has 4 basic parts:

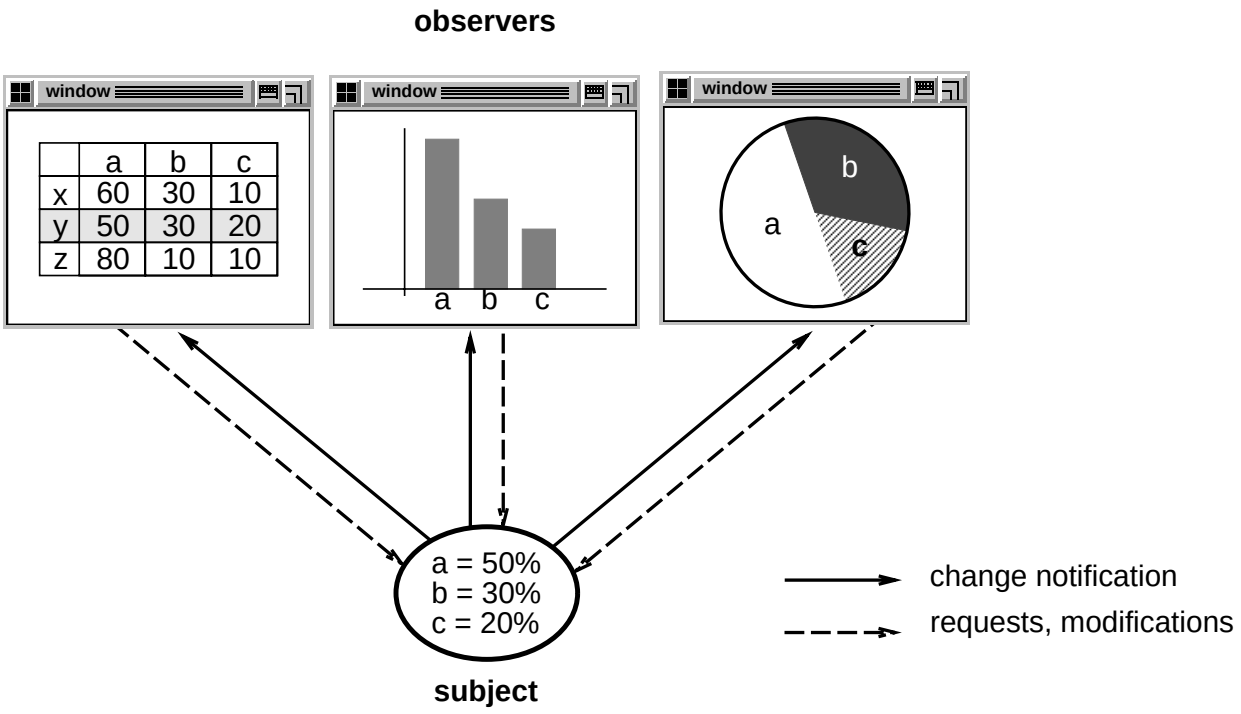
1. Name
2. Problem
3. Solution
4. Consequences and trade-offs of application

Language- and implementation-independent

A "micro-architecture"

Adjunct to existing methodologies (UML/P, Fusion, etc.)

Example: Observer



Goals

Codify good design

- Distill and disseminate experience
- Aid to novices and experts alike
- Abstract how to think about design

Give design structures explicit names

- Common vocabulary
- Reduced complexity
- Greater expressiveness

Capture and preserve design information

- Articulate design decisions succinctly
- Improve documentation

Facilitate restructuring/refactoring

- Patterns are interrelated
- Additional flexibility

Design Pattern Space

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Scope: domain over which a pattern applies

Purpose: reflects what a pattern does

Design Pattern Template (first half)

Name scope purpose

Intent
short description of pattern and its purpose

Also Known As
other names that people have for the pattern

Motivation
motivating scenario demonstrating pattern's use

Applicability
circumstances in which pattern applies

Structure
graphical representation of the pattern using modified OMT notation

Participants
participating classes and/or objects and their responsibilities

...

Design Pattern Template (second half)

...

Collaborations
how participants cooperate to carry out their responsibilities

Consequences
the results of application, benefits, liabilities

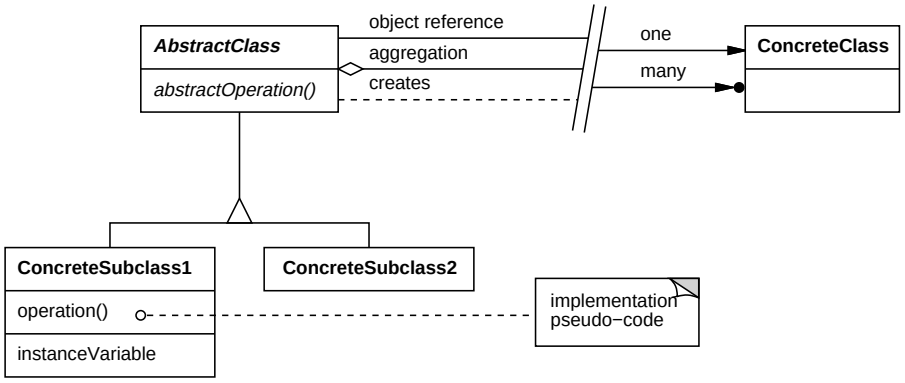
Implementation
implementation pitfalls, hints, or techniques, plus any language-dependent issues

Sample Code
sample implementations in C++ or Smalltalk

Known Uses
examples drawn from existing systems

Related Patterns
discussion of other patterns that relate to this one

Modified OMT Notation



Observer — object behavioral

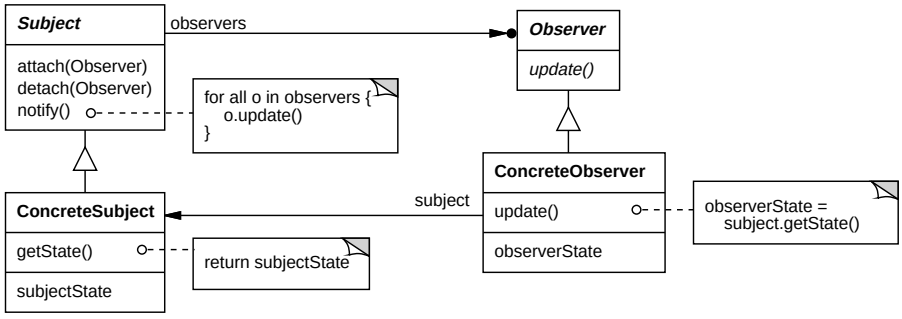
Intent

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

Applicability

- when an abstraction has two aspects, one dependent on the other
- when a change to one object requires changing others, and you don't know how many objects need to be changed
- when an object should notify other objects without making assumptions about who these objects are

Structure



Consequences

+ modularity: subject and observers may vary independently
+ extensibility: can define and add any number of observers
+ customizability: different observers provide different views of subject

− unexpected updates: observers don't know about each other
− update overhead: might need hints

Implementation

- subject-observer mapping
- dangling references
- avoiding observer-specific update protocols: the push and pull models
- registering modifications of interest explicitly

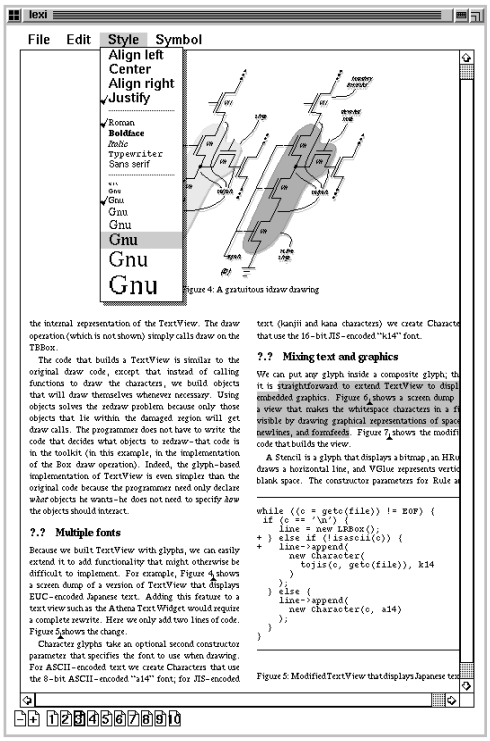
Known Uses

Smalltalk Model-View-Controller (MVC)
InterViews (Subjects and Views)
Andrew (Data Objects and Views)

Benefits

- *design* reuse
- uniform design vocabulary
- enhance understanding, restructuring
- basis for automation

Part II: Application



7 Design Problems:

- document structure
- formatting
- embellishment
- multiple look & feels
- multiple window systems
- user operations
- spelling checking & hyphenation

Document Structure

Goals:

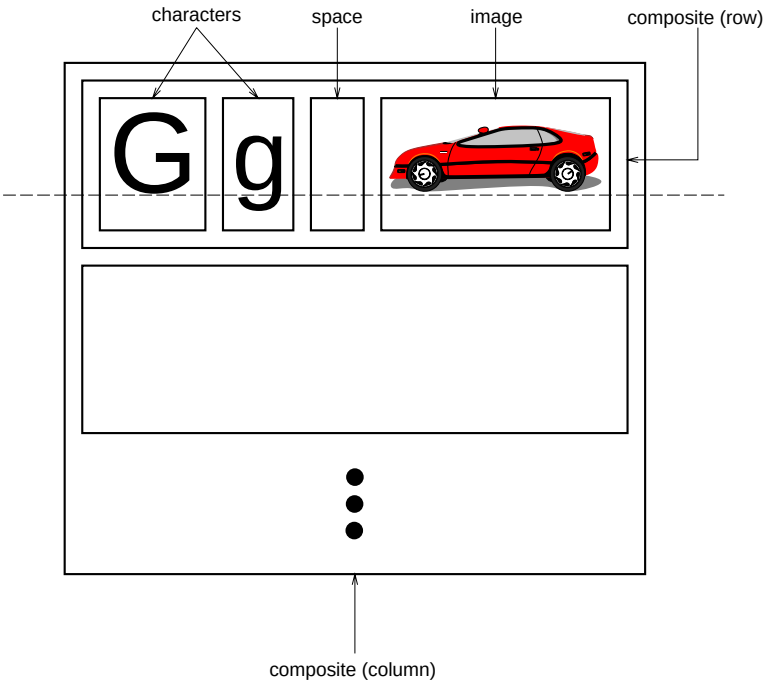
- present document's visual aspects
- drawing, hit detection, alignment
- support physical structure (e.g., lines, columns)

Constraints:

- treat text and graphics uniformly
- no distinction between one and many

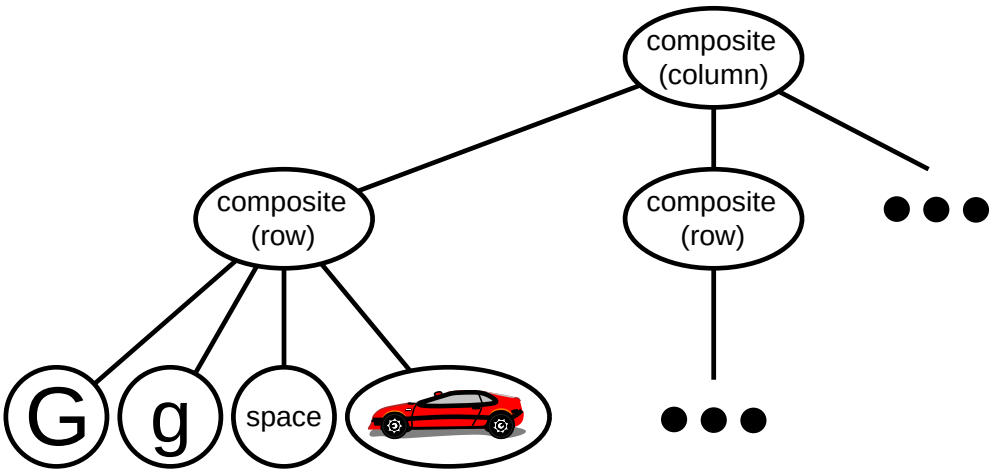
Document Structure (cont'd)

Solution: Recursive composition



Document Structure (cont'd)

Object structure



Document Structure (cont'd)

Glyph: base class for composable graphical objects

Basic interface:

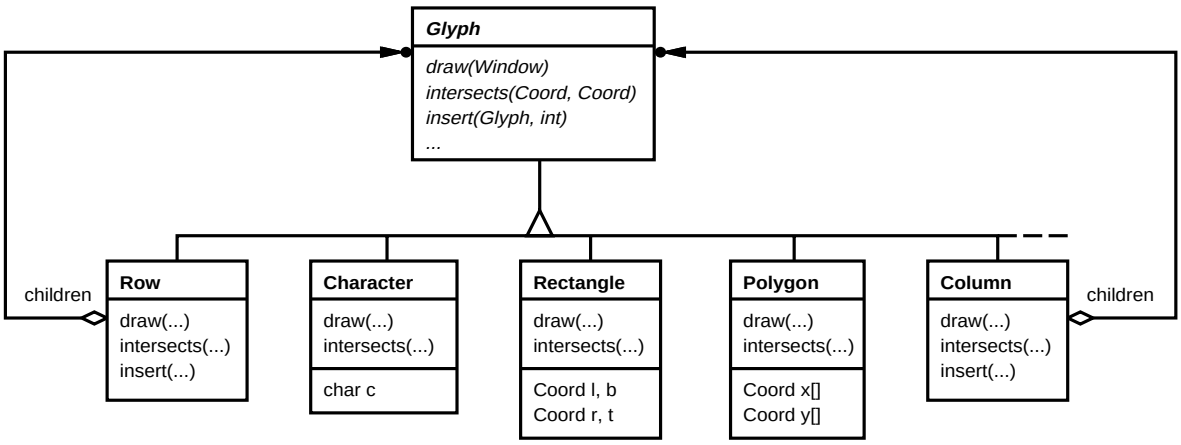
Task	Operations
appearance	`void draw(Window)`
hit detection	`boolean intersects(Coord, Coord)`

structure	`void insert(Glyph)`
	`void remove(Glyph)`
	`Glyph child(int)`
	`Glyph parent()`

Subclasses: Character, Image, Space, Row, Column

Document Structure (cont'd)

Glyph Hierarchy



Composite — object structural

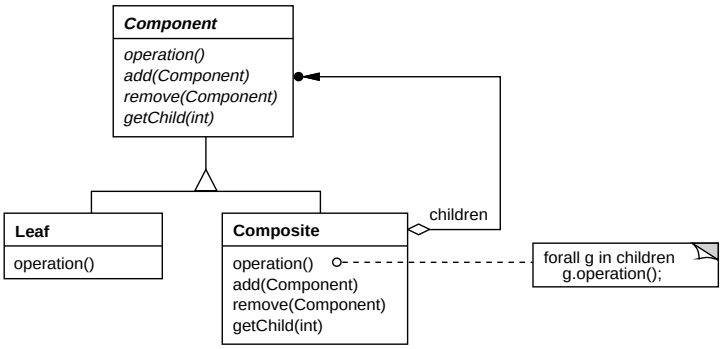
Intent

treat individual objects and multiple, recursively-composed objects uniformly

Applicability

objects must be composed recursively,
and there should be no distinction between individual and composed elements,
and objects in the structure can be treated uniformly

Structure



Composite (cont'd) — object structural

Consequences

+ uniformity: treat components the same regardless of complexity
+ extensibility: new Component subclasses work wherever old ones do

− overhead: might need prohibitive numbers of objects

Implementation

- do Components know their parents?
- uniform interface for both leaves and composites?
- don't allocate storage for children in Component base class
- responsibility for deleting children

Known Uses

ET++ VObjects
InterViews Glyphs, Styles
Unidraw Components, MacroCommands

Questions

What does the pattern let you vary?

Where have you applied this pattern in your designs?

What are the

- objects
- interfaces
- classes
- interactions

etc.?

Formatting

Goals:

- automatic linebreaking, justification

Constraints:

- support multiple linebreaking algorithms
- don't mix up with document structure

Formatting (cont'd)

Solution: Encapsulate linebreaking strategy

Compositor

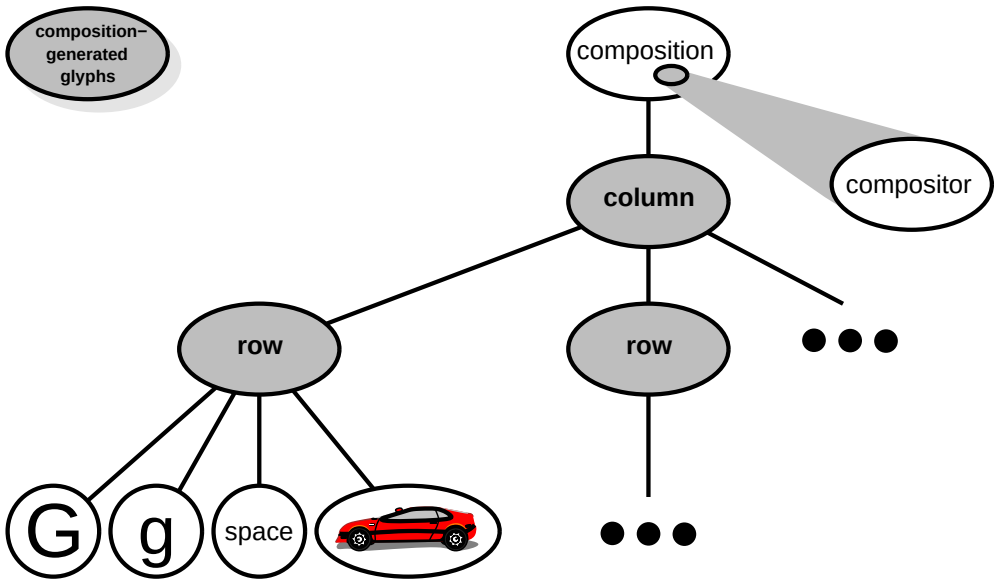
- base class abstracts linebreaking algorithm
- subclasses for specialized algorithms, e.g., **SimpleCompositor**, **TeXCompositor**

Composition

- composite glyph
- supplied a compositor and leaf glyphs
- creates row-column structure as directed by compositor

Formatting (cont'd)

New object structure



Strategy — object behavioral

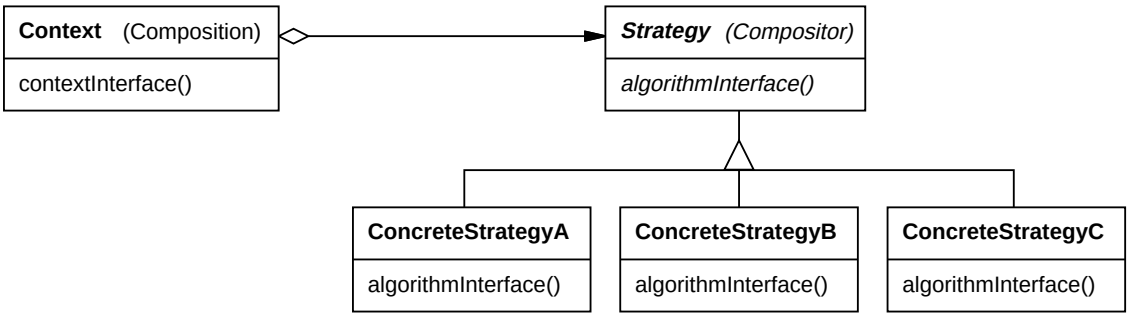
Intent

define a family of algorithms, encapsulate each one, and make them interchangeable to let clients and algorithms vary independently

Applicability

when an object should be configurable with one of several algorithms,
and all algorithms can be encapsulated,
and one interface covers all encapsulations

Structure



Strategy (cont'd) — object behavioral

Consequences

\+ greater flexibility, reuse
\+ can change algorithms dynamically

\– strategy creation & communication overhead
\– inflexible Strategy interface

Implementation

- exchanging information between a Strategy and its context
- static strategy selection via templates

Known Uses

InterViews text formatting
RTL register allocation & scheduling strategies
ET++SwapsManager calculation engines

Embellishment

Goals:

- add a frame around text composition
- add scrolling capability

Constraints:

- embellishments should be reusable without subclassing
- should go unnoticed by clients

Embellishment (cont'd)

Solution: "Transparent" enclosure

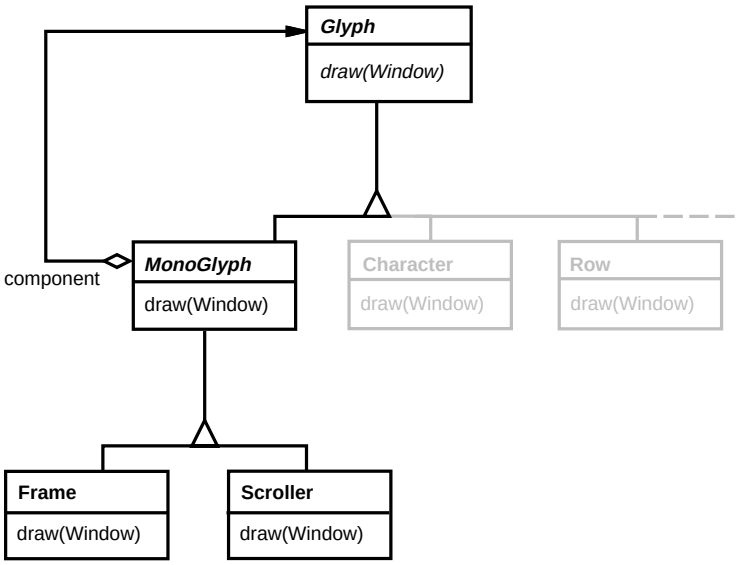
MonoGlyph

- base class for glyphs having **one** child
- operations on MonoGlyph pass through to child

MonoGlyph subclasses:

- **Frame**: adds a border of specified width
- **Scroller**: scrolls/clips child, adds scrollbars

Embellishment (cont'd)

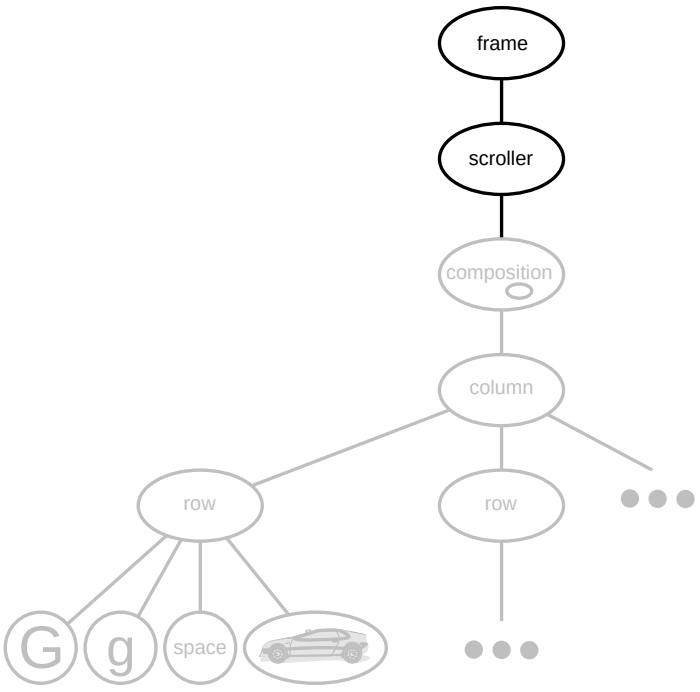



```
void MonoGlyph.draw (Window w) {
    component.draw(w);
}
```

```
void Frame.draw (Window w) {
    super.draw(w);
    drawFrame(w);
}
```


Embellishment (cont'd)

New object structure



Decorator — object structural

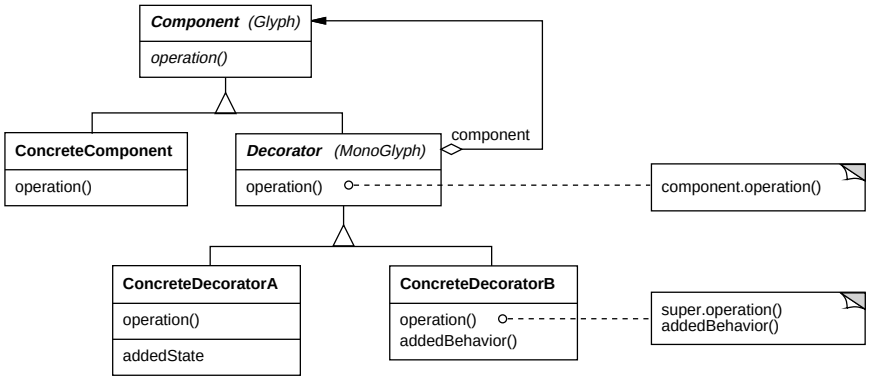
Intent

augment objects with new responsibilities

Applicability

- when extension by subclassing is impractical
- for responsibilities that can be withdrawn

Structure



Decorator (cont'd) — object structural

Consequences

+ responsibilities can be added/removed at run-time
+ avoids subclass explosion
+ recursive nesting allows multiple responsibilities

– interface occlusion
– identity crisis

Implementation

- interface conformance
- use a lightweight, abstract base class for Decorator
- heavyweight base classes make Strategy more attractive

Known Uses

embellishment objects from most OO-GUI toolkits
ParcPlace PassivityWrapper
InterViews DebuggingGlyph

Multiple Look & Feels

Goals:

- support multiple look and feel standards
- generic, Motif, PM, Macintosh, Windows, ...
- extensible for future standards

Constraints:

- don't recode existing widgets or clients
- switch look and feel without recompiling

Multiple Look & Feels (cont'd)

Solution:
Abstract the process of creating objects

Instead of

```
Scrollbar sb = new MotifScrollbar();
```

use

```
Scrollbar sb = factory.createScrollbar();
```

where `factory` is an instance of **MotifFactory**

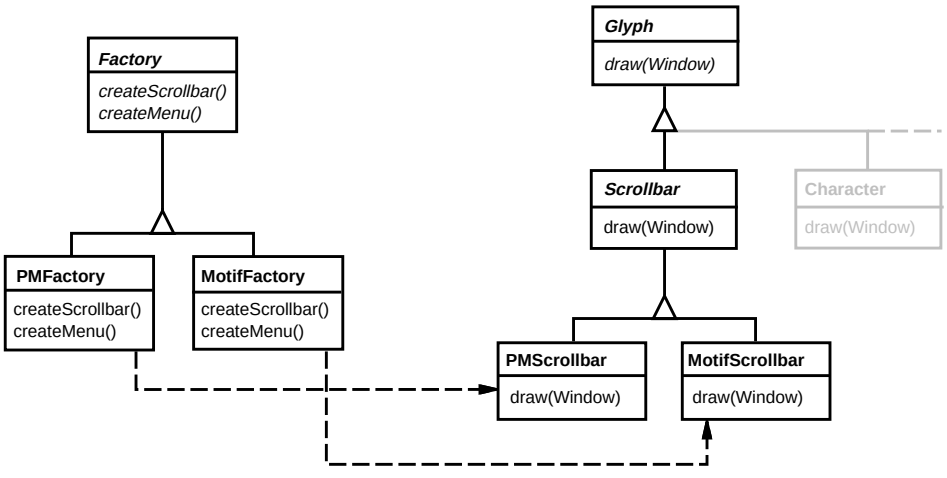
Multiple Look & Feels (cont'd)

Factory interface

- defines "manufacturing interface"
- subclasses produce specific products
- subclass instance chosen at run-time

```
interface Factory {
    Scrollbar createScrollbar();
    Menu createMenu();
    ...
}
```


Multiple Look & Feels (cont'd)




```
Scrollbar MotifFactory.createScrollBar () {
    return new MotifScrollbar();
}

Scrollbar PMFactory.createScrollBar () {
    return new PMScrollbar();
}
```

Abstract Factory — object creational

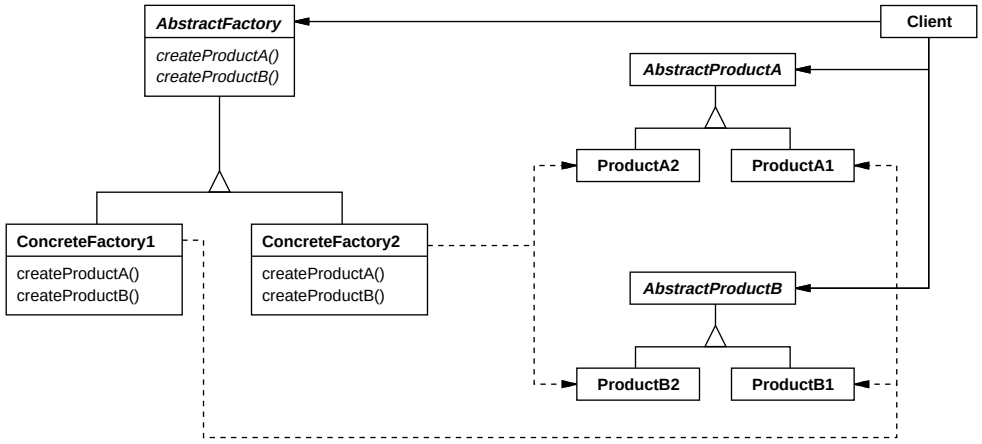
Intent

create families of related objects without specifying class names

Applicability

when clients cannot anticipate groups of classes to instantiate

Structure



Abstract Factory (cont'd) — object creational

Consequences

+ flexibility: removes type dependencies from clients
+ abstraction: hides product's composition

− hard to extend factory interface to create new products

Implementation

- parameterization as a way of controlling interface size
- configuration with Prototypes

Known Uses

InterViews Kits
ET++ WindowSystem

Multiple Window Systems

Goals:

- make composition appear in a window
- support multiple window systems

Constraints:

- minimize window system dependencies

Multiple Window Systems (cont'd)

Solution: Encapsulate implementation dependencies

Window

- user-level window abstraction
- displays a glyph (structure)
- window system-independent
- task-related subclasses (e.g., IconWindow, PopupWindow)

Multiple Window Systems (cont'd)

Window interface

```
interface Window {
    ...
    void iconify();          // window-management
    void raise();
    ...
    void drawLine(...);      // device-independent
    void drawText(...);      // graphics interface
    ...
}
```

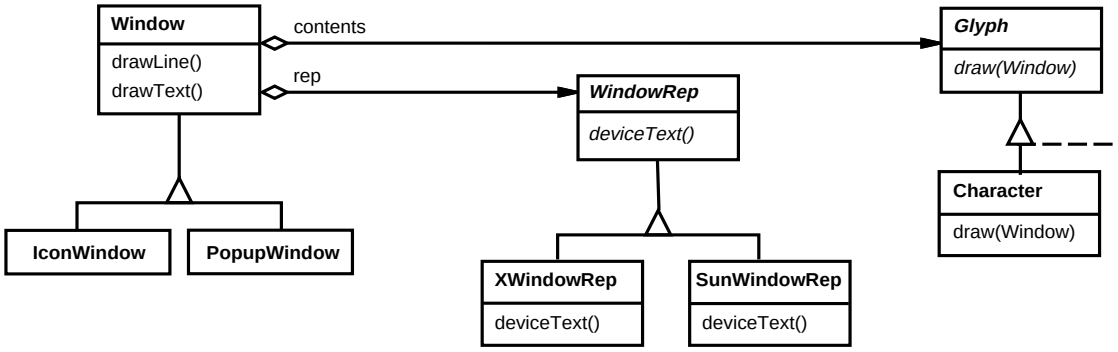

Multiple Window Systems (cont'd)

Window uses a **WindowRep**

- abstract implementation interface
- encapsulates window system dependencies
- window systems-specific subclasses
  (e.g., XWindowRep, SunWindowRep)

An Abstract Factory can produce the right WindowRep!

Multiple Window Systems (cont'd)



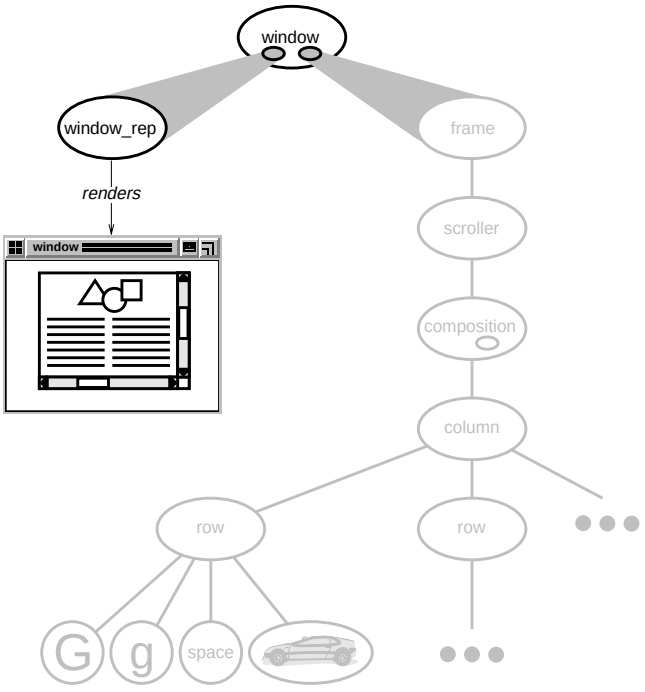

```
void Character.draw (Window w) {
    w.drawText(...);
}

void Window.drawText (...) {
    rep.deviceText(...);
}

void XWindowRep.deviceText (...) {
    XText(...);
}
```


Multiple Window Systems (cont'd)

New object structure



Multiple Window Systems (cont'd)

Bridge — object structural

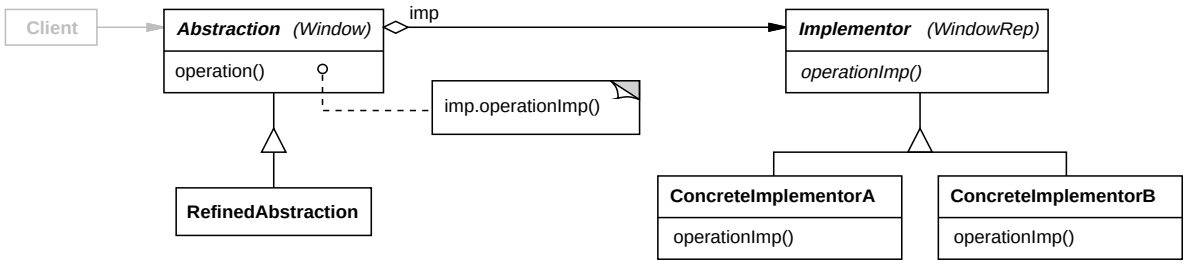
Intent

separate an abstraction from its implementation

Applicability

- when interface and implementation should vary independently
- require a uniform interface to interchangeable class hierarchies

Structure



Multiple Window Systems (cont'd)

Bridge (cont'd) — object structural

Consequences

+ abstraction and implementation are independent
+ implementations may vary dynamically

− one-size-fits-all Abstraction and Implementor interfaces

Implementation

- sharing Implementors
- creating the right implementor

Known Uses

ET++ Window/WindowPort

libg++ Set/{LinkedList,HashTable}

User Operations

Goals:

- support execution of user operations
- support unlimited-level undo

Constraints:

- scattered operation implementations
- must store undo state
- not all operations are undoable

User Operations (cont'd)

Solution: Encapsulate the request for a service

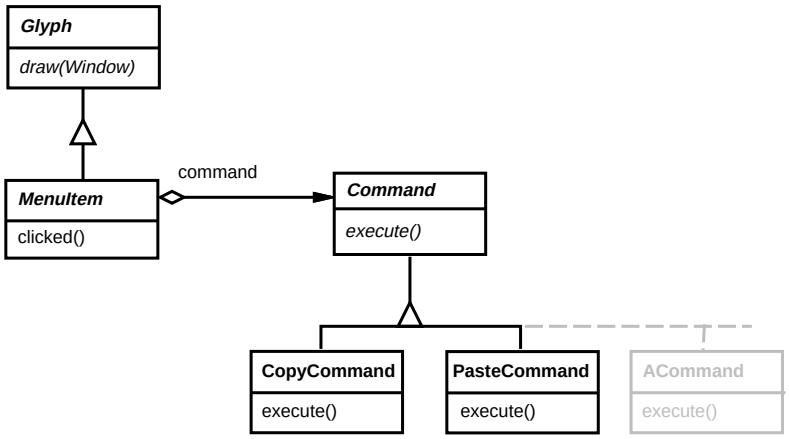
Command encapsulates

- an operation (`execute()`)
- an inverse operation (`unexecute()`)
- a operation for testing reversibility (`boolean reversible()`)
- state for (un)doing the operation

Command may

- implement the operations itself, *or*
- delegate them to other object(s)

User Operations (cont'd)




```
void MenuItem.clicked () {
    command.execute();
}
```

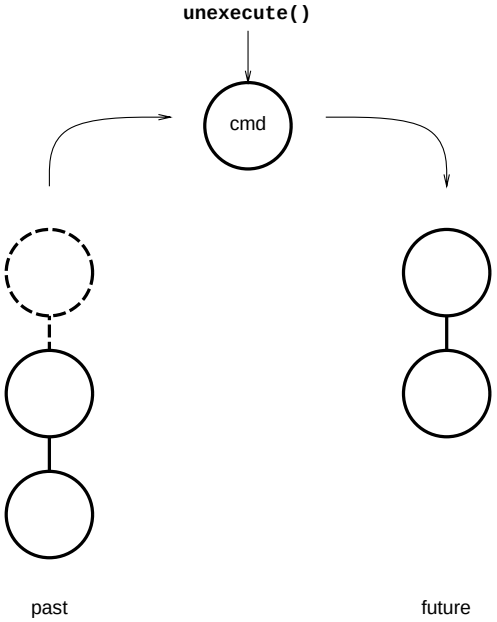
```
void PasteCommand.execute () {
    // do the paste
}

void CopyCommand.execute () {
    // do the copy
}
```

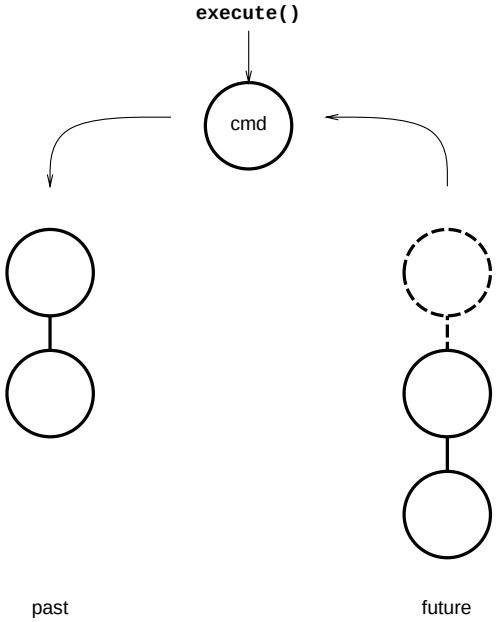

User Operations (cont'd)

List of commands defines execution history

Undo:



Redo:



Command — object behavioral

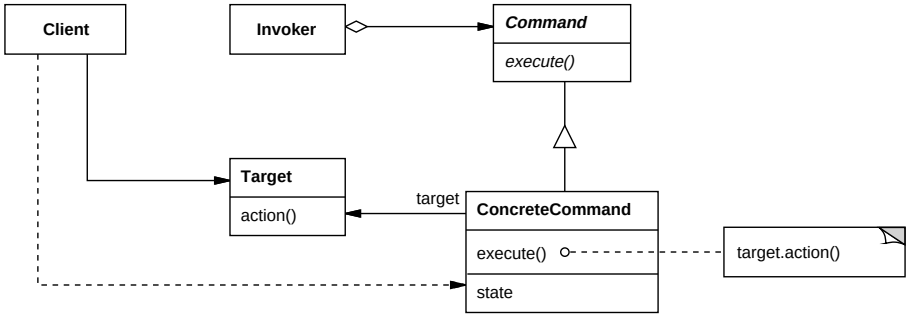
Intent

encapsulate the request for a service

Applicability

- to parameterize objects with an action to perform
- to specify, queue, and execute requests at different times
- for a history of requests
- for multilevel undo/redo

Structure



Command (cont'd) — object behavioral

Consequences

+ abstracts executor of a service
+ supports arbitrary-level undo-redo
+ composition yields macro-commands

– might result in lots of trivial command subclasses

Implementation

- copying a command before putting it on a history list
- handling hysteresis
- supporting transactions

Known Uses

InterViews Actions
MacApp, Unidraw Commands

Spelling Checking and Hyphenation

Goals:

- analyze text for spelling errors
- introduce potential hyphenation sites

Constraints:

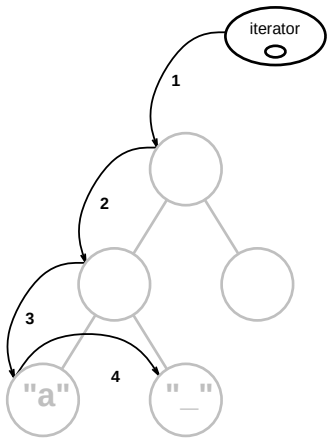
- support multiple algorithms
- don't mix up with document structure

Spelling Checking and Hyphenation (cont'd)

Solution: Encapsulate traversal

Iterator

- encapsulates a traversal algorithm
- uses Glyph's child enumeration operation



Spelling Checking and Hyphenation (cont'd)

Iterator object behavioral

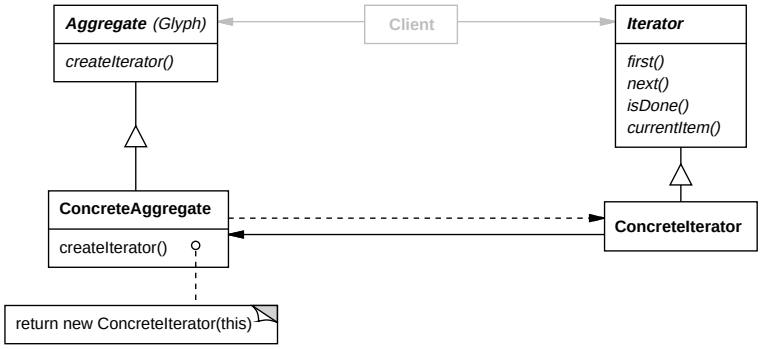
Intent

access elements of an aggregate sequentially without exposing its representation

Applicability

- require multiple traversal algorithms over an aggregate
- require a uniform traversal interface over different aggregates
- when aggregate classes and traversal algorithm must vary independently

Structure



Spelling Checking and Hyphenation (cont'd)

Iterator (cont'd) object behavioral

Consequences

+ flexibility: aggregate and traversal are independent
+ multiple iterators → multiple traversal algorithms

− additional communication overhead between iterator and aggregate

Implementation

- internal versus external iterators
- violating the object structure's encapsulation
- robust iterators

Known Uses

Penpoint traversal driver/slave
InterViews ListItr
Unidraw Iterator

Visitor

- defines action(s) at each step of traversal
- avoids wiring action(s) into Glyphs
- iterator calls glyph's `accept(Visitor)` at each node
- `accept` calls back on visitor

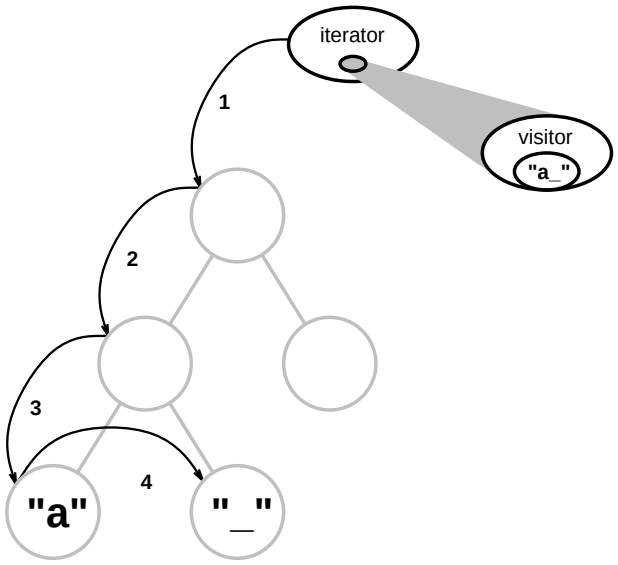
```
void Character.accept (Visitor v) { v.visit(this); }

interface Visitor {
    void visit(Character);
    void visit(Rectangle);
    void visit(Row);
    // etc. for all relevant Glyph subclasses
}
```


SpellingCheckerVisitor

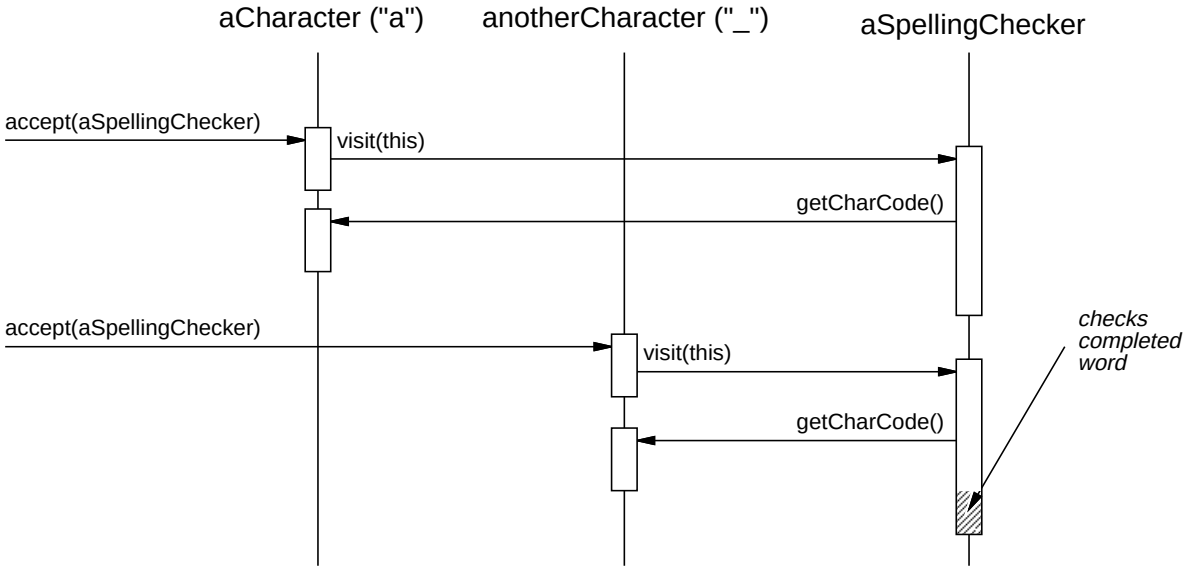
- gets character code from each character glyph
  Can define `getCharCode` operation just on `Character` class
- checks words accumulated from character glyphs
- combine with **PreorderIterator**

Accumulating Words



Spelling check on each non-alphabetic character

Interaction Diagram



Spelling Checking and Hyphenation (cont'd)

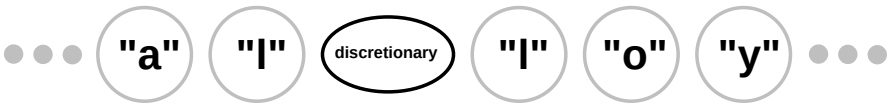
HyphenationVisitor

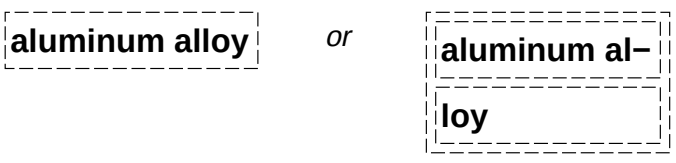
- gets character code from each character glyph
- examines words accumulated from character glyphs
- at potential hyphenation point, inserts a...

Spelling Checking and Hyphenation (cont'd)

Discretionary glyph

- looks like a hyphen when it falls at the end of a line
- has no appearance otherwise
- Compositor considers its presence when determining linebreaks





Visitor — object behavioral

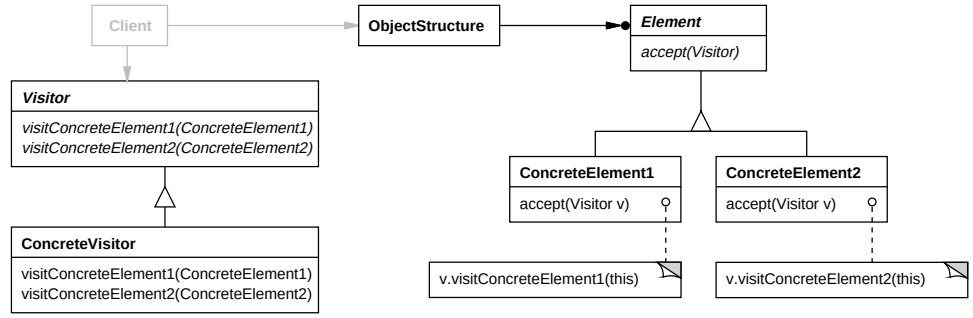
Intent

centralize operations on an object structure so that they can vary independently but still behave polymorphically

Applicability

- when classes define many unrelated operations
- class relationships of objects in the structure rarely change, but the operations on them change often
- algorithms over the structure maintain state that's updated during traversal

Structure



Visitor (cont'd) — object behavioral

Consequences

+ flexibility: visitor and object structure are independent
+ localized functionality

− circular dependency between Visitor and Element interfaces
− Visitor brittle to new ConcreteElement classes

Implementation

- double dispatch
- overloading `visit` operations
- catch-all operation
- general interface to elements of object structure

Known Uses

ProgramNodeEnumerator in Smalltalk-80 compiler
IRIS Inventor scene rendering

Part III: Wrap-Up

Observations

Applicable in all stages of the OO lifecycle

- Design & reviews
- Realization & documentation
- Reuse & refactoring

Permit design at a more abstract level

- Treat many class/object interactions as a unit
- Often beneficial *after* initial design
- Targets for class refactorings

Variation-oriented design

- Consider what design aspects are variable
- Identify applicable pattern(s)
- Vary patterns to evaluate tradeoffs
- Repeat

But...

Resist branding everything a pattern

- Articulate specific benefits
- Demonstrate wide applicability
- Find at least *two* existing examples

Don't apply them blindly

- Added indirection → increased complexity, cost

Pattern design even harder than OOD!

Conclusion

Design patterns promote

- *design* reuse
- uniform design vocabulary
- understanding, restructuring
- automation
- a new way of thinking about design

(Design) Pattern References

The Timeless Way of Building, Alexander; Oxford, 1979; ISBN 0-19-502402-8

A Pattern Language, Alexander; Oxford, 1977; ISBN 0-19-501-919-9

Design Patterns, Gamma, et al.; Addison-Wesley, 1995; ISBN 0-201-63361-2; CD version ISBN 0-201-63498-8

Pattern-Oriented Software Architecture, Buschmann, et al.; Wiley, 1996; ISBN 0-471-95869-7

Analysis Patterns, Fowler; Addison-Wesley, 1996; ISBN 0-201-89542-0

Smalltalk Best Practice Patterns, Beck; Prentice Hall, 1997; ISBN 0-13-476904-X

The Design Patterns Smalltalk Companion, Alpert, et al.; Addison-Wesley, 1998; ISBN 0-201-18462-1

AntiPatterns, Brown, et al.; Wiley, 1998; ISBN 0-471-19713-0

More Books:

Pattern Languages of Program Design (Addison-Wesley)
Vol. 1, Coplien, et al., eds.; 1995; ISBN 0-201-60734-4
Vol. 2, Vlissides, et al., eds.; 1996; ISBN 0-201-89527-7
Vol. 3, Martin, et al., eds.; 1998; ISBN 0-201-31011-2
Vol. 4, Harrison, et al., eds.; 2000; ISBN 0-201-43304-4

Concurrent Programming in Java, Lea; Addison-Wesley, 1997; ISBN 0-201-69581-2

Applying UML and Patterns, Larman; Prentice Hall, 1997; ISBN 0-13-748880-7

Pattern Hatching: Design Patterns Applied, Vlissides; Addison-Wesley, 1998; ISBN 0-201-43293-5

Future Books:

The Pattern Almanac, Rising; Addison-Wesley, 2000; ISBN 0-201-61567-3

Early Papers:

"Object-Oriented Patterns," P. Coad; Comm. of the ACM, 9/92

"Documenting Frameworks using Patterns," R. Johnson; OOPSLA '92

"Design Patterns: Abstraction and Reuse of Object-Oriented Design," Gamma, Helm, Johnson, Vlissides, ECOOP '93.

Columns:

C++ Report, Dr. Dobbs Sourcebook, JOOP, ROAD

Conferences:

PLoP 2000: Pattern Languages of Programs
September 2000, Monticello, Illinois, USA

EuroPLoP 2000
July 2000, Kloster Irsee, Germany

ChiliPLoP 2000
March 2000, Wickenburg, Arizona, USA

KoalaPLoP 2000
May 2000, Melbourne, Australia

See http://hillside.net/patterns/conferences for up-to-the-minute information.

Mailing Lists:

`patterns@cs.uiuc.edu`: present and refine patterns

`patterns-discussion@cs.uiuc.edu`: general discussion on patterns

`gang-of-4-patterns@cs.uiuc.edu`: discussion on *Design Patterns*

`siemens-patterns@cs.uiuc.edu`: discussion on *Pattern-Oriented Software Architecture*

`ui-patterns@cs.uiuc.edu`: discussion on user interface design patterns

`business-patterns@cs.uiuc.edu`: discussion on patterns for business processes

`ipc-patterns@cs.uiuc.edu`: discussion on patterns for distributed systems

See http://hillside.net/patterns/Lists.html for an up-to-date list.

URLs:

General
`http://hillside.net/patterns`
`http://www.research.ibm.com/designpatterns`

Conferences
`http://hillside.net/patterns/conferences/`

Books
`http://hillside.net/patterns/books/`

Mailing Lists
`http://hillside.net/patterns/Lists.html`

Portland Patterns Repository
`http://c2.com/ppr/index.html`