Designing with Patterns

John Vlissides

IBM T.J. Watson Research

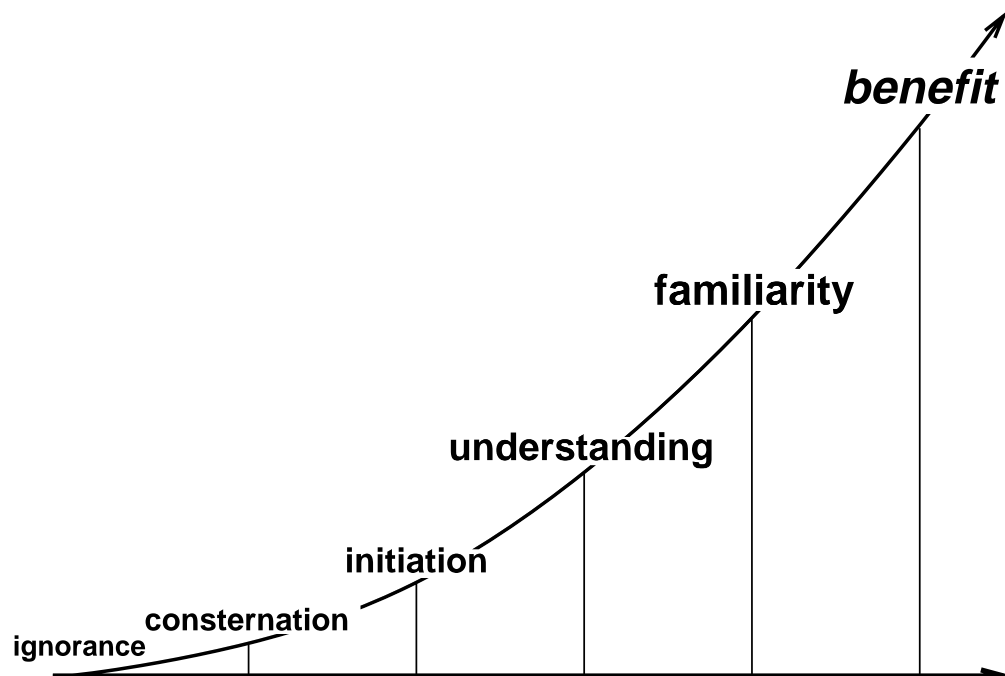
vlis@watson.ibm.com

© 1996–1999 by John Vlissides. All rights reserved.

Diagrams from *Design Patterns: Elements of Reusable Object-Oriented Software* © 1995 by Addison-Wesley Publishing Company. All rights reserved. Diagrams may not be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Introduction

Stages of Design Pattern awareness:



Objectives

Learn to apply design patterns to the design process

- find the right patterns
- understand (un)applicability
- see when and how to bend a pattern
- evaluate design trade-offs effectively

Learn by (counter)example

Designing a File System

Running example

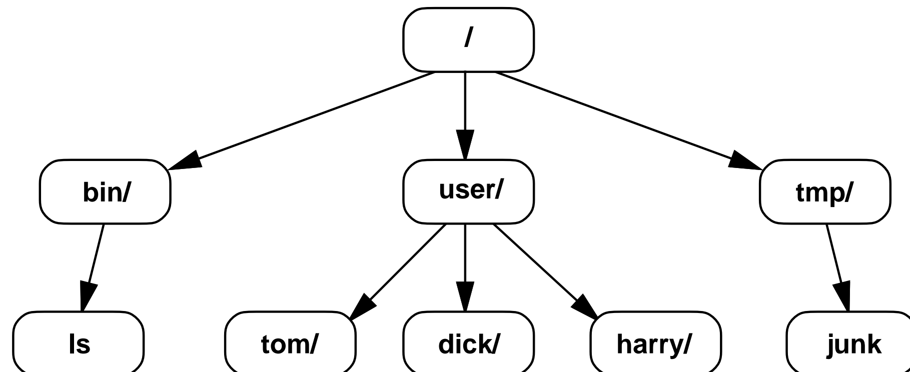
Design problems:

1. Structure
2. Symbolic links
3. Open-ended functionality
4. Single-user protection
5. Multi-user protection
6. Notification

File System Structure

Problem:

- represent file system elements (files, directories)
- *for end-user:* file system of arbitrary size and complexity
- *for programmer:* easy to deal with and extend



File System Structure (cont'd)

Tree structures → COMPOSITE pattern

Rule of thumb
Choice hinges on importance of uniformity and extensibility

Application issues:

- choosing participants: Component, Leaf, and Composite
- choosing operations to treat uniformly

File System Structure (cont'd)

Composite — object structural

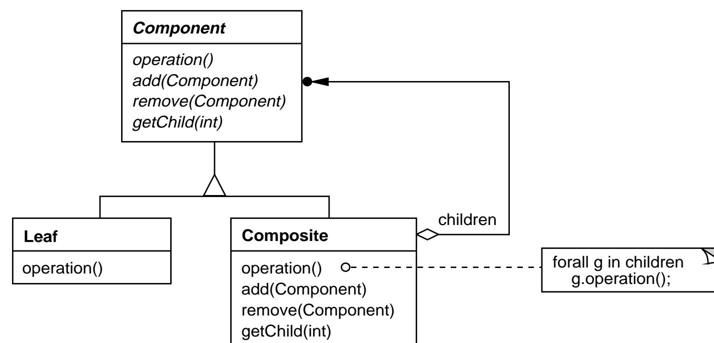
Intent

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



File System Structure (cont'd)

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

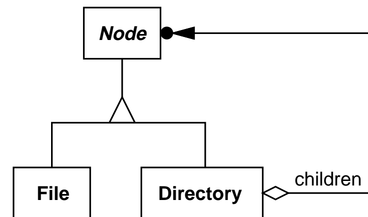
File System Structure (cont'd)

Mapping COMPOSITE participants to file system classes:

- Leaf, for objects that have no children
  → `File`, the file object

- Composite, for objects that have children
  → `Directory`, the directory object

- Component, the uniform interface
  → `Node`



File System Structure (cont'd)

What can `File` objects do?

- return their *name* and *protection*
- *stream* their contents in and out

What can `Directory` objects do?

- return their *name* and *protection*
- *enumerate* their children
- *adopt* and *orphan* children

File System Structure (cont'd)

What *uniform interface* does `Node` define?

- get name/protection

  Obviously common

- stream in/out

  Less-obviously common

- enumerate children

  Needed for recursion, hiding internal data structure
  Could apply ITERATOR instead

- adopt & orphan

  Trade-off between type safety and uniformity

File System Structure (cont'd)

Uniform `adopt`/`orphan` interface simplifies clients

As long as Leaf objects can handle them gracefully

Example: `mkdir`

- "`mkdir newsubdir`"

  → new `newsubdir` subdirectory

- "`mkdir subdirA/subdirB/newsubdir`"

  → new `newsubdir` subdirectory of `subdirB`

File System Structure (cont'd)

Naive `mkdir` implementation

```
public void mkdir (Directory current, String path) {
    String subpath = subpath(path);

    if (subpath == null) {
        if (find(path, current) == null) {
            current.adopt(new Directory(path));
        } else {
            System.err.println(path + " exists.");
        }
    } else {
        String name = head(path);
        Node child = find(name, current);

        if (child != null) {
            mkdir(child, subpath);
        } else {
            System.err.println(name + " nonexistent.");
        }
    }
}
```


File System Structure (cont'd)

`find` searches for a child with the given name

Must return a `Node`

```
public Node find (String name, Directory current) {
    Node child = null;

    for (int i = 0; child = current.getChild(i); ++i) {
        if (name.equals(child.getName())) {
            return child;
        }
    }
    return null;
}
```

File System Structure (cont'd)

But `mkdir` won't compile!

- `mkdir` takes a `Directory`, not a `Node`
- Using `instanceof` adds a control path

```
// ...
Node node = find(name, current);

if (node != null) {
    if (node instanceof Directory) {
        mkdir((Directory) node, subpath);
    } else {
        System.err.println(getName() + " is not a directory.");
    }

} else {
    // ...
```


File System Structure (cont'd)

Solution: Treat `adopt` and `orphan` uniformly

- declare them in `Node` interface
- define default behavior

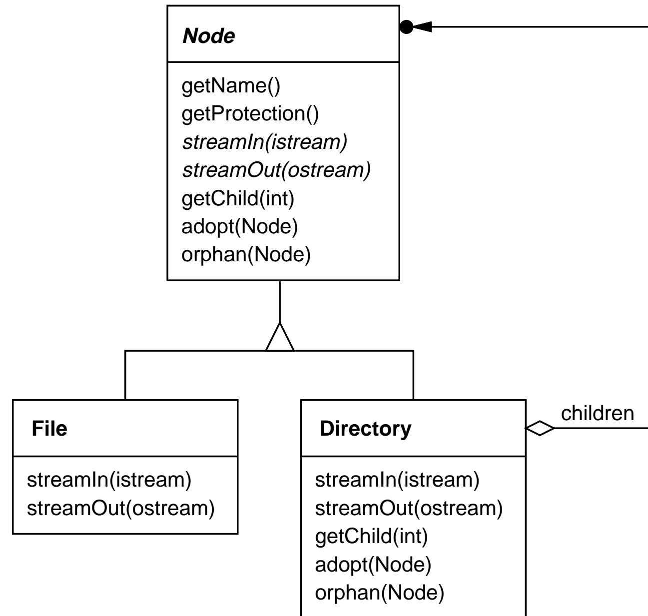
```
public abstract class Node {
    public void adopt (Node child) {
        System.err.println(getName() + " is not a directory.");
    }

    public void orphan (Node child) {
        System.err.println(child.getName() + " not found.");
    }
    // ...
```

Only change to `mkdir` is its signature:

```
void mkdir (Node current, String path) { ... }
```

File System Structure (cont'd)

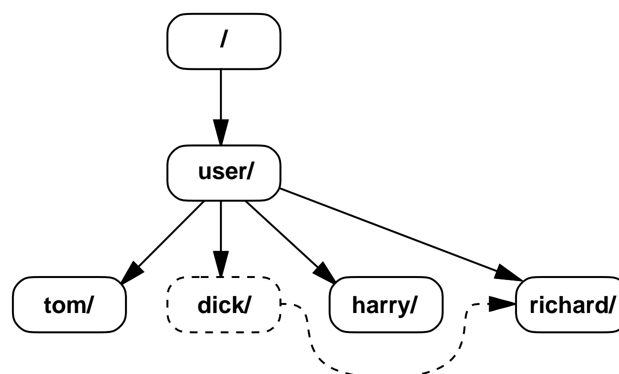
Result:



Symbolic Links

Problem:

- add symbolic link capability uninvasively
- should work across directories, file systems, even machines



Symbolic Links (cont'd)

Finding the right pattern

- consider how design patterns solve design problems
  IOW, study section 1.6—no time today
- scan intent sections
  brute-force
- study how patterns interrelate ("spaghetti diagram," etc.)
  still too involved, but getting warmer
- ...

Symbolic Links (cont'd)

Finding the right pattern

- ...
- look at patterns of relevant purpose (creational, structural, behavioral)
  symbolic links suggest *structural* purpose
- examine a cause of redesign (listed on p. 24)
  not worried about that just yet
- consider what should be variable in your design (Table 1.2)

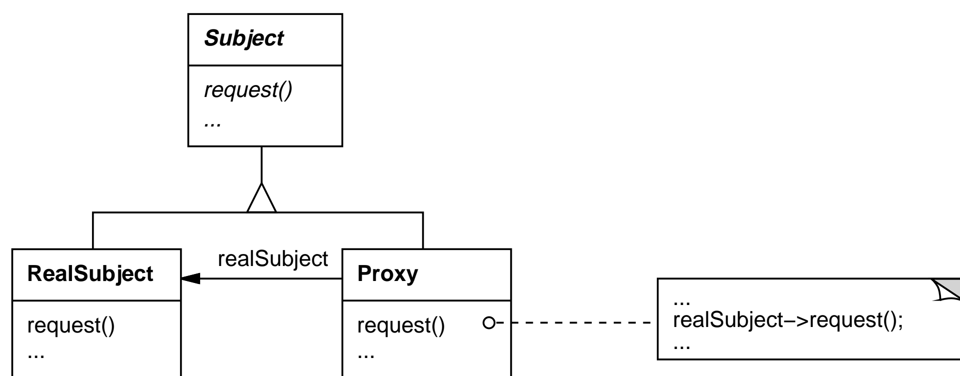
Symbolic Links (cont'd)

Variabilities imparted by structural patterns

- ADAPTER: interface to an object
- BRIDGE: implementation of an object
- COMPOSITE: object structure and composition
- DECORATOR: responsibilities without subclassing
- FACADE: interface to a subsystem
- FLYWEIGHT: storage costs of objects
- PROXY: how an object is accessed and/or its location

Symbolic Links (cont'd)

PROXY structure



- Proxy is a stand-in for RealSubject
- Proxy must match Subject interface

Symbolic Links (cont'd)

Mapping PROXY participants to file system classes:

- Subject, the interface to match
  → `Node`

- Proxy, the stand-in class
  → `Link`, the symbolic link object

- RealSubject, to which the proxy refers
  → ???

Problem:
Don't want to commit RealSubject to either `File` or `Directory`

Symbolic Links (cont'd)

Solution: look at PROXY's description of the Proxy participant:

> [Proxy] maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.

→ `Node` is the RealSubject

N.B.: this couldn't work without COMPOSITE's uniform interface!

Symbolic Links (cont'd)

`Link` implementation: delegate all operations to `RealSubject`

Example:

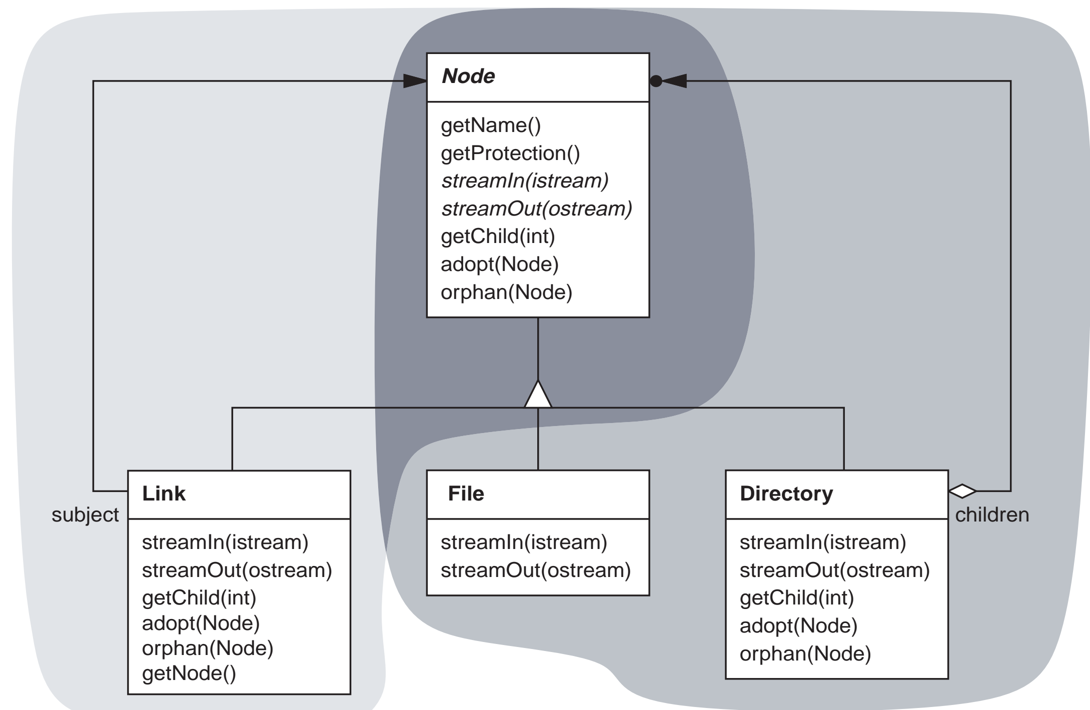
```
public class Link extends Node {
    public Node getChild (int n) {
        return _subject.getChild(n);
    }
    // ...
```

Additional Link-specific operation:

```
    Node getNode () { return _subject; }
```

(for clients who know they are dealing with a `Link`)

Interlude



Open-Ended Functionality

Problem:

- clients want to do arbitrarily many operations on file system objects

  `cat`, `ls`, `du`, `chmod`, `chown`, ...

- must avoid treating `Node` interface as a dumping ground

Open-Ended Functionality (cont'd)

Externalizing operations → VISITOR pattern

Choice hinges on stability of Element class hierarchy

Consequences:

+ recovers type information without downcasts

+ consolidates and encapsulates functionality in Visitor object

– new ConcreteElements may require changing Visitor interface

– circular dependency between Visitor and Element interfaces

Open-Ended Functionality (cont'd)

Visitor object behavioral

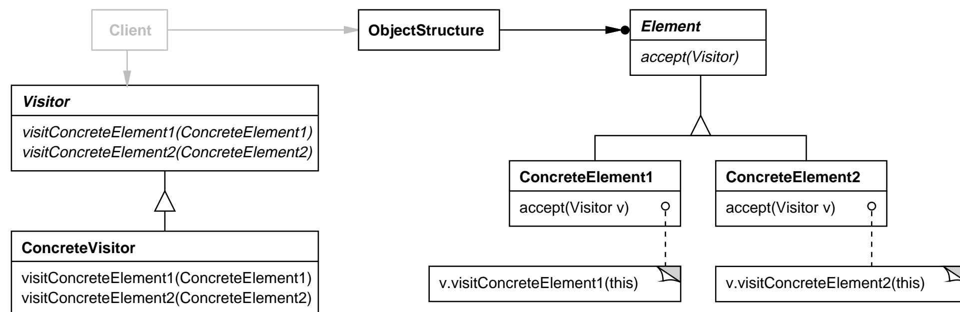
Intent

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



Open-Ended Functionality (cont'd)

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
- general interface to elements of object structure

Known Uses

ProgramNodeEnumerator in Smalltalk-80 compiler
IRIS Inventor scene rendering

Open-Ended Functionality (cont'd)

Defining a `CatVisitor` (lists a file)

1. Define `NodeVisitor` base class

2. Add `accept` operation to `Node` base class and subclasses

3. Define `CatVisitor` subclass of `NodeVisitor`

Open-Ended Functionality (cont'd)

```
public abstract class NodeVisitor {
    public void visit(Node n)           { }  // default

    public void visit(File f)           { visit((Node) f); }
    public void visit(Directory d)      { visit((Node) d); }
    public void visit(Link l)           { visit((Node) l); }
}
```

`accept` operations:

```
public void Node.accept (NodeVisitor v)      { v.visit(this); }
    // => NodeVisitor.visit(Node)

public void File.accept (NodeVisitor v)      { v.visit(this); }
    // => NodeVisitor.visit(File)

public void Directory.accept (NodeVisitor v) { v.visit(this); }
    // => NodeVisitor.visit(Directory}

public void Link.accept (NodeVisitor v)      { v.visit(this); }
    // => NodeVisitor.visit(Link)
```

Open-Ended Functionality (cont'd)

`CatVisitor` subclass implementation

```
public class CatVisitor extends NodeVisitor {
    public void visit (File f) {
        f.streamOut(System.out);
    }

    public void visit (Directory d) {
        System.err.println("Can't cat a directory.");
    }

    public void visit (Link l) {
        l.getNode().accept(this);
    }
```

Usage:

```
    CatVisitor cat = new CatVisitor();
    node.accept(cat);
```


Open-Ended Functionality (cont'd)

What if Element hierarchy *isn't* stable?

Example: `HardLink` is a new subclass of `Node`:

```
public class HardLink extends Node {
    public void accept (NodeVisitor v) { v.visit(this); }
        // => NodeVisitor.visit(Node)
    // ...
```

`visit(Node)` acts as catch-all

- No problem if no visitor treats `HardLink` objects specially and/or default behavior adequate
- Otherwise, need RTTI...

Open-Ended Functionality (cont'd)

Example: `CatVisitor` must deal with `HardLink`s specially

1. Use RTTI in the element:

```
public void accept (NodeVisitor nv) {
    if (nv instanceof CatVisitor) {
        CatVisitor cv = (CatVisitor) nv;
        cv.visit(this);
            // => CatVisitor-specific visit(HardLink)

    } else {
        nv.visit((Node) this);  // do the default
    }
}
```

+ `NodeVisitor` subclasses unchanged if you update `NodeVisitor` interface
– new `NodeVisitor` subclasses force change in `Node` subclasses
? # branches proportional to # new `NodeVisitor` subclasses

Open-Ended Functionality (cont'd)

2. Use RTTI in the visitor:

```
public void visit (Node n) {
    if (n instanceof HardLink) {
        // do something special for hard links

    } else {
        super.visit(n);  // do the default
    }
}
```

+ `Node` subclasses unchanged if you update `NodeVisitor` interface (assumes overloading)
– RTTI in potentially every `NodeVisitor` subclass
? # branches proportional to # new `Node` subclasses

Single-User Protection

Problem:

- protect file system objects from inadvertent corruption

  No defense against malicious corruption

- make nodes (un)readable and/or (un)writable

  Approach should work for other protection modes, too

Single-User Protection (cont'd)

How do these protection modes affect a node's behavior?

- an unreadable node can't divulge its contents

  → files shouldn't respond to `streamOut` requests
  → directories shouldn't enumerate their children

- an unwritable node can't be modified

  → shouldn't respond to `streamIn`
  → (C++) must disable destructor

What patterns support these (anti)responsibilities?

Single-User Protection (cont'd)

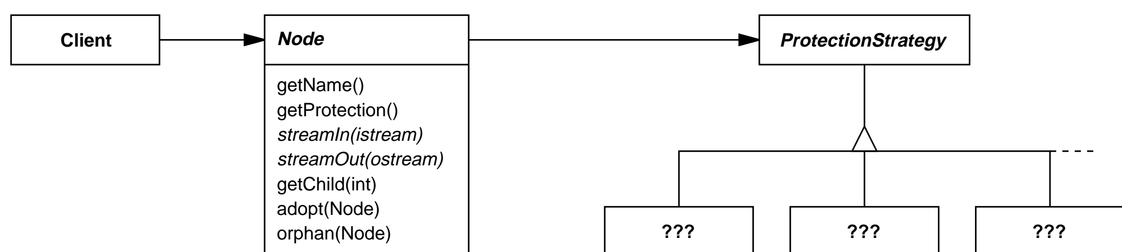
Observation 1: Protection behavior must vary *dynamically*

- user can change protection at any time
- suggests a behavioral pattern, especially STRATEGY or STATE
- DECORATOR is a non-invasive alternative to STRATEGY

Observation 2: Access control suggests PROXY

Single-User Protection (cont'd)

Applying STRATEGY



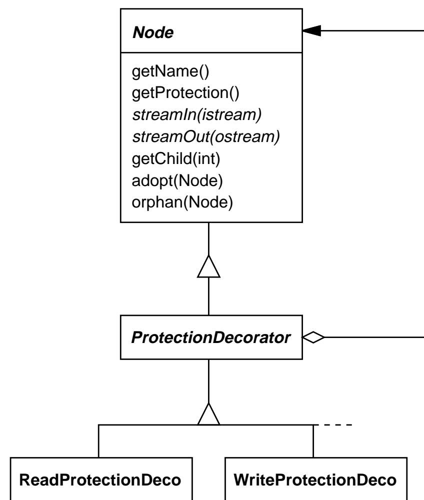
How many strategy objects does a node use?

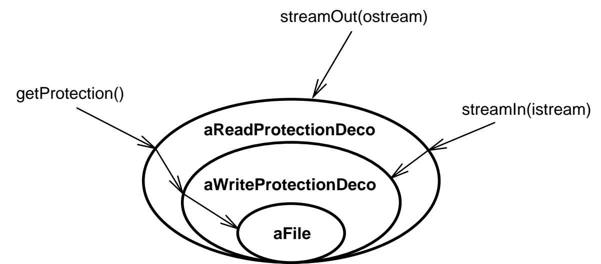
One per node, one per `Node` *operation*, ... ?
If multiple, how do they work together?
What do `Node` subclasses delegate to `ProtectionStrategies`?

STATE poses similar questions

Single-User Protection (cont'd)

Is DECORATOR any better?





Single-User Protection (cont'd)

DECORATOR's drawbacks:

- When a node's protection changes, what happens to existing references to it?

  Object identity problem

- Lots more objects → high overhead

PROXY has similar drawbacks

New observations:

- adding objects isn't helping
  Delegation adds complexity
  Objects add overhead

- Strategy, State, Decorator, and Proxy are all object patterns

How about using *class* pattern(s) to vary behavior?

Template Method

- can vary behavior of each operation independently in subclasses

- no object or delegation overhead

To apply it, determine *variant* and *invariant* parts

- **invariant**: determine current protection (read and/or write)

- **variant**: response (normal operation or nop/error message)

Template Method — class behavioral

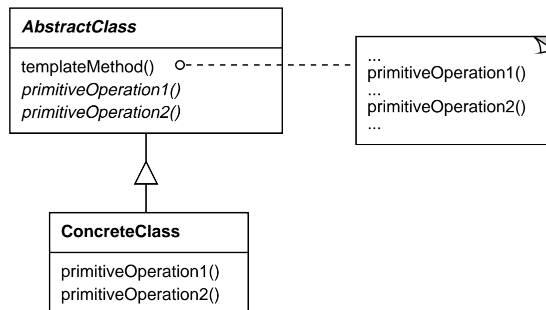
Intent

define the skeleton of an algorithm in an operation, deferring some steps to subclasses

Applicability

- to implement invariant aspects of an algorithm once and let subclasses define variant parts
- to localize common behavior in a class to increase code reuse
- to control subclass extensions

Structure



Template Method (cont'd) — class behavioral

Consequences

\+ leads to inversion of control ("Hollywood principle": don't call us—we'll call you)

\+ promotes code reuse

\+ lets you enforce overriding rules

– must subclass to specialize behavior

Implementation

- virtual vs. non-virtual template method
- few vs. lots of primitive operations
- naming conventions (`do-` prefix)

Known Uses

just about all object-oriented systems (especially frameworks)

Single-User Protection (cont'd)

Typical implementation:

```
public abstract class Node {
    public final void streamOut (OutputStream out) {
        if (isReadable()) {
            doStreamOut(out);
        } else {
            doWarning(unreadableWarning);
        }
    }
    // ...
```

`isReadable`, `doStreamOut`, and `doWarning` are primitive operations

Single-User Protection (cont'd)

Preventing deletion of unwritable node in C++:

1. Protect destructor
2. Define `static void Node::Delete(Node*)` as template method
3. Pass doomed node to primitive operations

   Static doesn't have `this`

```
void Node::Delete (Node* node) {
    if (node->isWritable()) {
        delete node;
    } else {
        node->doWarning(undeletableWarning);
    }
}
```

Multi-User Protection

Problem:

- extend protection scheme to support multiple users
- associate users with files
- support named groups of users

Mimic the Unix file system model

"user," "group," "other" protection modes

Multi-User Protection (cont'd)

Questions:

- how to *model* a user?
- how to *authenticate* a user?
- how does authentication impact node operations?

Multi-User Protection (cont'd)

Assume we model a user as an object

Natural metaphor
Stick with it until it proves unworkable

One `User` instance per "login name" (in Unix sense)

What can we do with a `User` object?

Multi-User Protection (cont'd)

More important: What *can't* we do with a `User`?

- identifies a user to the system
- must prevent masquerade

$\rightarrow$ must control instantiation process

login name $+$ valid password $\Rightarrow$ `User` instance

Encapsulating the instantiation process

A relevant creational pattern?

- ABSTRACT FACTORY: no "families"; not averse to instantiating a concrete class
- BUILDER: we're not *varying* the creation process
- FACTORY METHOD: see ABSTRACT FACTORY
- PROTOTYPE: can't leave prototypes lying around
- SINGLETON: need $>$ 1 **`User`** object...

Singleton object creational

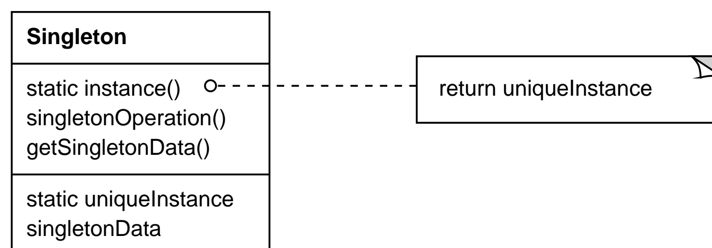
Intent

ensure a class only ever has one instance, and provide a global point of access to it.

Applicability

- when there must be exactly one instance of a class, and it must be accessible from a well-known access point
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Structure



Singleton (cont'd) object creational

Consequences

+ reduces namespace pollution
+ makes it easy to change your mind and allow more than one instance
+ allow extension by subclassing

– same drawbacks of a global if misused
– implementation may be less efficient than a global
– concurrency pitfalls

Implementation

- static `instance` operation
- registering the singleton instance

Known Uses

Unidraw's Unidraw object
Smalltalk-80 ChangeSet, the set of changes to code
InterViews Session object

From SINGLETON's Consequences:

> [SINGLETON] permits a variable number of instances. The pattern makes it easy to change your mind and allow more than one instance of the Singleton class. Moreover, you can use the same approach to **control the number of instances** that the application uses. Only the [Instance] operation that grants access to the Singleton instance needs to change.

We want one and only one `User` instance *per user*

Multi-User Protection (cont'd)

`User` modifies SINGLETON's `Instance` operation a bit:

```
public static final User logIn (String loginName, String password) {
    if (password incorrect for loginName) {
        return null;
    }

    if (a User instance exists for loginName) {
        return it;

    } else {
        return new User instance for loginName;
    }
}
```


Multi-User Protection (cont'd)

Recap of `logIn`'s properties

- globally accessible
- no more than one `User` object per login name
- can return null (or 0) if it fails
- cannot be changed by subclassing

Multi-User Protection (cont'd)

Using a `User`

`Node` operations only work for certain user(s)

Defined by who "owns" the node and its protection mode

How do `Node` operations ascertain the user?

Multi-User Protection (cont'd)

Two approaches:

1. Pass `User` in each call
   - e.g., `void streamOut(OutputStream, User);`
   - "stateless" design
   - can be a nuisance
2. Define an "implicit" `User`
   - signatures unchanged from single-user versions
   - "stateful" design
   - requires global `set/getUser` interface

Multi-User Protection (cont'd)

In C++, default parameters allow both approaches:

```
const char* getName(const User* = 0);
const Protection& getProtection(const User* = 0);

void setName(const char*, const User* = 0);
void setProtection(const Protection&, const User* = 0);

void streamIn(istream&, const User* = 0);
void streamOut(ostream&, const User* = 0);

Node* getChild(int, const User* = 0);
void adopt(Node*, const User* = 0);
void orphan(Node*, const User* = 0);
```

In Java, use overloading ($\rightarrow$ double the operations)

Multi-User Protection (cont'd)

Typical impact on template methods:

```
public void streamOut (OutputStream out, User user) {
    if (isReadableBy(user)) {
        doStreamOut(out);
    } else {
        doWarning(unreadableWarning);
    }
}

public boolean isReadableBy (User user) {
    boolean isOwner = user.owns(this);
        // true iff user's login name matches node's owner

    return
        isOwner && isUserReadable() ||
        !isOwner && isOtherReadable();
}
```

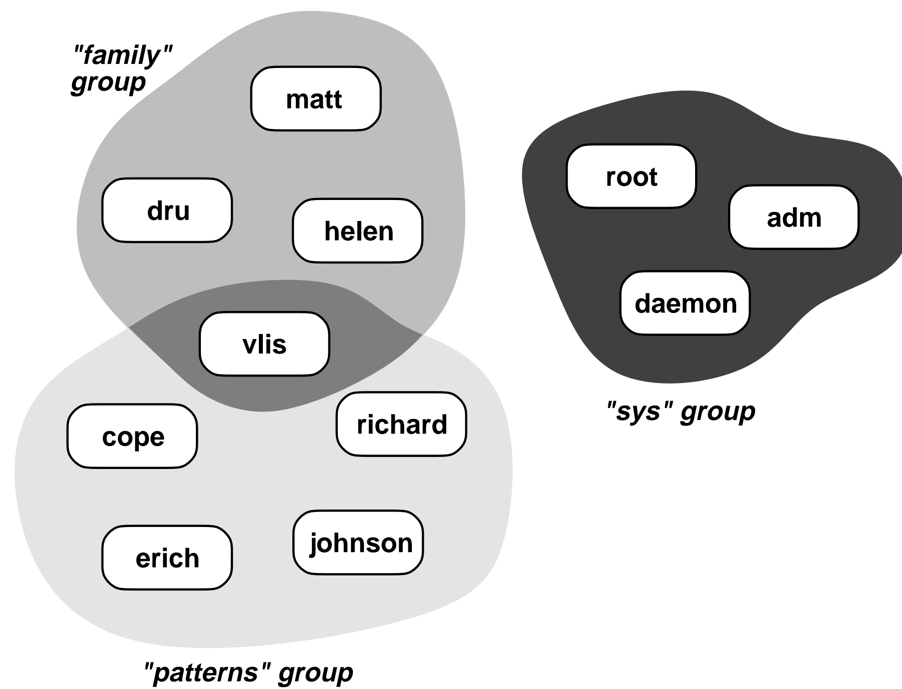
Multi-User Protection (cont'd)

More questions:

- how to model groups of users?
- how to associate users and groups?

Multi-User Protection (cont'd)

Example grouping:



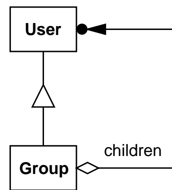
Multi-User Protection (cont'd)

Groups as objects

Subclass from existing hierarchy, or new class (hierarchy)?

Assume `User` subclass

A Composite of users?



Multi-User Protection (cont'd)

Why COMPOSITE is *not* applicable:

1. User-group not a recursive relationship

   Not in Unix, at least

2. Not strictly hierarchical

   A user can belong to more than one group

3. Questionable to treat users and groups uniformly

   Login a group? Pass it around as authentication?

Multi-User Protection (cont'd)

Still need to associate groups and users

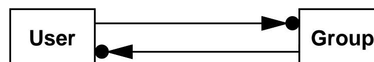
Need a *two-way* mapping for efficiency

- # users $\gg$ # groups
  $\rightarrow$ find all group members without scanning all users

- checks for group membership should be fast, too

Multi-User Protection (cont'd)

One solution:



Drawbacks:

- references hard to change noninvasively

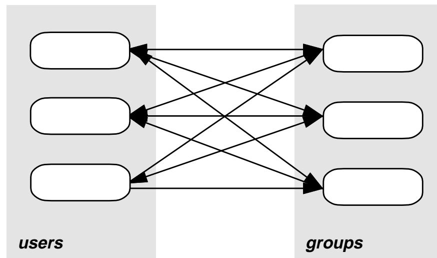
- all objects saddled with cost of a (pointer to a) collection

- tangle of references between `User` and `Group` objects

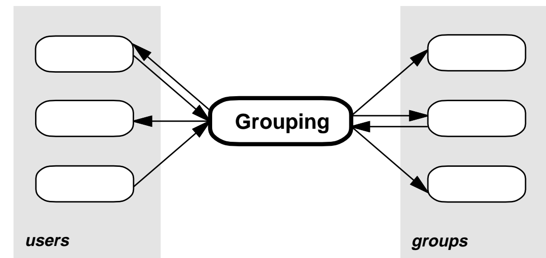
Multi-User Protection (cont'd)

Managing object interconnections → MEDIATOR

Before:



After:



`Grouping` is probably a singleton

Multi-User Protection (cont'd)

Mediator object behavioral

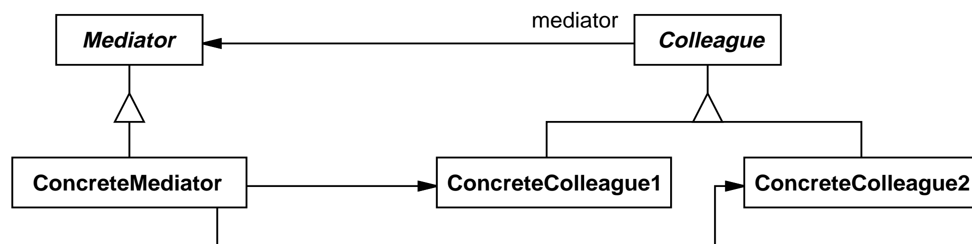
Intent

define an object that encapsulates how a set of objects interact to promote loose coupling and to let you vary their interaction independently

Applicability

- there's cooperative behavior that can't be assigned to an individual object
- a set of objects communicate in well-defined but complex ways
- ordering of operations may change as system evolves

Structure



Mediator (cont'd) — object behavioral

Consequences

+ encapsulates communication
+ simplifies protocols between objects
+ avoids pushing mediation responsibility into one or more colleagues

– Mediator can become complex and monolithic

Implementation

- using static members instead of a separate class
- Singleton mediators

Known Uses

Unidraw's Editor, CSolver
ET++ PrinterManager, DialogDirector

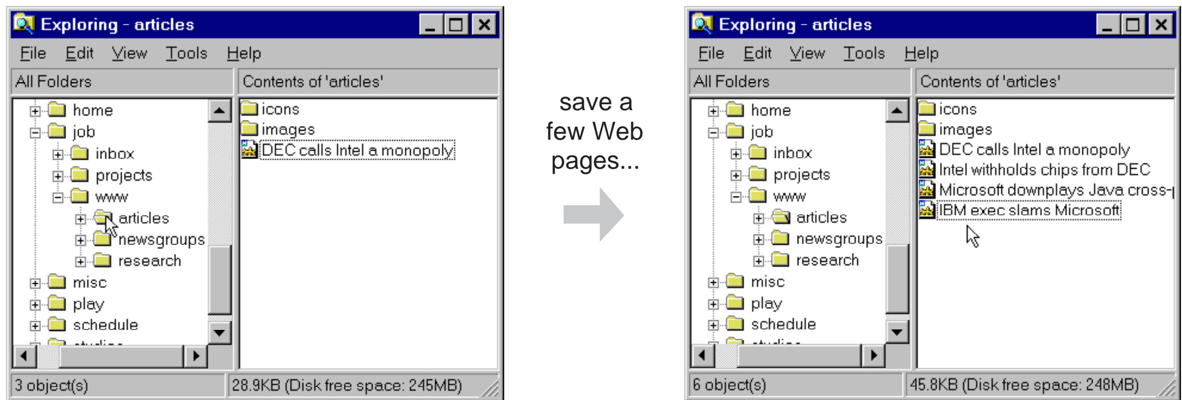
`Grouping` interface

```
public abstract class Grouping {
    public static Grouping getGrouping();    // returns singleton
    public static void setGrouping(Grouping);
    public static void setGrouping(Grouping, User);

    public void register(User, Group);
    public void register(User, Group, User);

    public void unregister(User, Group);
    public void unregister(User, Group, User);

    public Group getGroup(String loginName, int index);
    public String getUser(Group, int index);
}
```

Notification

Problem: Clients sensitive to file system changes

Example: Files created by one application appear in another:



Don't want to hit "Refresh" to see the new files!

Other examples: mail arrival, clipboards, embedded documents

Notification (cont'd)

"Update," "notification," "dependency" portend OBSERVER pattern

A particularly rich one

Application issues:

- choosing participants: (Concrete)Subject, (Concrete)Observer
- the Subject-Observer mapping: how fancy?

Observer — object behavioral

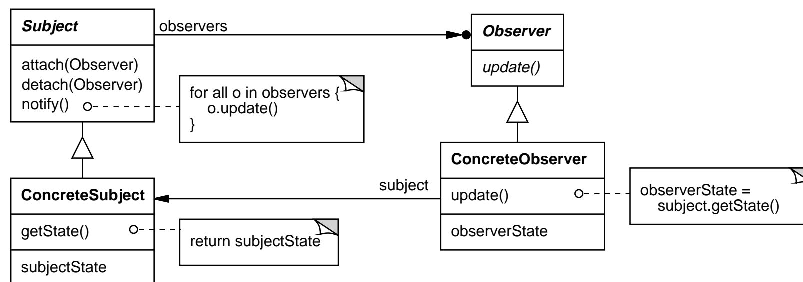
Intent

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



Observer (cont'd) — object behavioral

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)

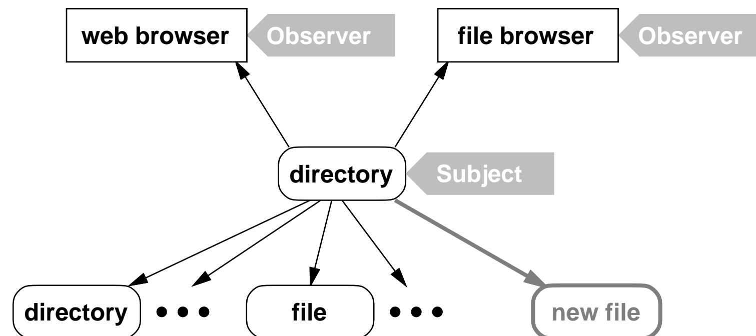
InterViews (Subjects and Views)

Andrew (Data Objects and Views)

Notification (cont'd)

Mapping Observer participants to file system classes

Directory notifies file browser of change in contents:



→ *any* object might observe (= Observer) **and**
any object might be observed (= Subject)

Notification (cont'd)

Define `Subject` and `Observer` as interfaces

Lets you give subject and/or observer characteristics to any class

```
interface Subject {
    void attach(Observer);
    void detach(Observer);
    void notify();
}
```

```
interface Observer {
    void update(Subject);
}
```

```
public abstract class Node implements Subject {
    // ...
}

public class Browser implements Observer {
    // ...
}
```

Notification (cont'd)

Note `Subject` parameter to `update`

Observer may observe multiple subjects

```
public class Browser implements Observer {
    // ...

    public void update (Subject s) {
        if (_openDirectories.contains(s)) {
            Directory d = (Directory) s;
            // update display(s) of d
        }
    }

    private Vector _openDirectories;
}
```


Notification (cont'd)

```
public abstract class Node implements Subject {
    // ...

    public void attach (Observer o) {
        _observers.addElement(o);
    }

    public void detach (Observer o) {
        _observers.removeElement(o);
    }

    public void notify () {
        for (int i = 0; i < _observers.size(); ++i) {
            Observer o = (Observer) _observers.elementAt(i);
            o.update(this);
        }
    }

    private Vector _observers;
}
```

All concrete subjects implement something similar to this...

Notification (cont'd)

Consolidating Subject operations in a **change manager**

Can eliminate them from Subject entirely

Reuse by composition, not inheritance

```
public class ChangeManager {
    public void register (Subject s, Observer o) {
        Vector observers = (Vector) _registry.get(s);

        if (observers == null) {
            observers = new Vector();
            _registry.put(s, observers);
        }
        observers.addElement(o);
    }

    public void unregister (Subject s, Observer o) {
        Vector observers = (Vector) _registry.get(s);

        if (observers != null) {
            observers.removeElement(o);
        }
    }

   // ...
}
```


Notification (cont'd)

```
public class ChangeManager {
    // ...

    public void notify (Subject s) {
        Enumeration e = _registry.elements();

        while (e.hasMoreElements()) {
            Vector observers = (Vector) e.nextElement();

            for (int i = 0; i < observers.size(); ++i) {
                Observer o = (Observer) observers.elementAt(i);
                o.update(s);
            }
        }
    }

    private Hashtable _registry = new Hashtable();
}
```

- yet another Mediator
- may be a Singleton

Summary

Structure: COMPOSITE

Symbolic links: PROXY

Open-ended functionality: VISITOR

Single-user protection: TEMPLATE METHOD

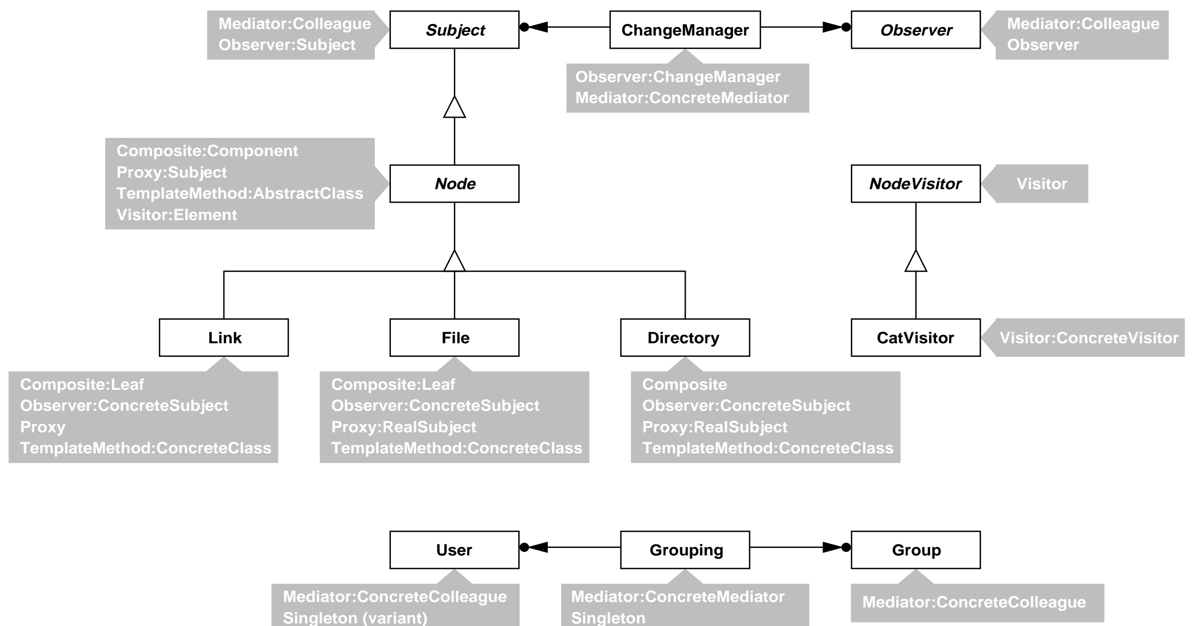
Not STRATEGY, DECORATOR, or PROXY

Multi-user protection:

- SINGLETON: creating users, "implicit" user, `Grouping`
- MEDIATOR: groups
  - Not COMPOSITE

Notification: OBSERVER

Summary (cont'd)



Experiences

Design patterns can't guarantee a good overall architecture

They're just micro-architectures

Creativity still required

- you might never implement a pattern the same way twice
- not all design decisions are covered by patterns

Experiences (cont'd)

Not always obvious which design pattern to apply

- the solutions of some patterns look similar:
  "Just add a level of indirection."

  → STATE, STRATEGY, BRIDGE, ...
- but the problem/intent they address is different

Learning the patterns takes time

You have to experience the problem to appreciate the solution

Pattern Pitfalls

Overenthusiasm

- patterns have costs (indirection, complexity)
- therefore design to be as flexible *as needed*, not as flexible *as possible*

  "Complex systems that work evolved from simple systems that worked."—Booch

  "Start stupid and evolve."—Beck

Overly dense application

E.g., a class that participates in all 23 patterns!

Reducing the world to design patterns

(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