Design Patterns Applied

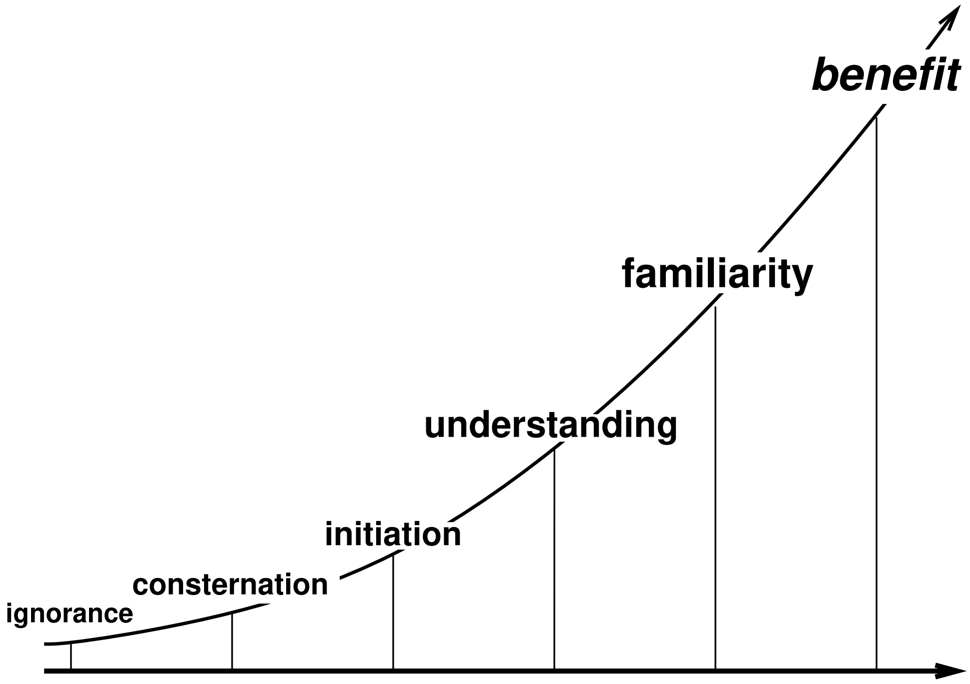
Erich Gamma, IFA Consulting

Richard Helm, IBM Consulting

John Vlissides, IBM Research

Introduction

Stages of Design Pattern awareness:



Overview

Introduction

Part I: *Designing* with Patterns

Designing a file system

Part II: *Redesigning* with Patterns

Redesigning a hardware configuration management system

Part III: *Refining* with Patterns

Debugging, testing, tuning

Part IV: *Communicating* with Patterns

Discussion, exposition, documentation

Conclusion

Part 1: **Designing with Patterns**

Designing a file system

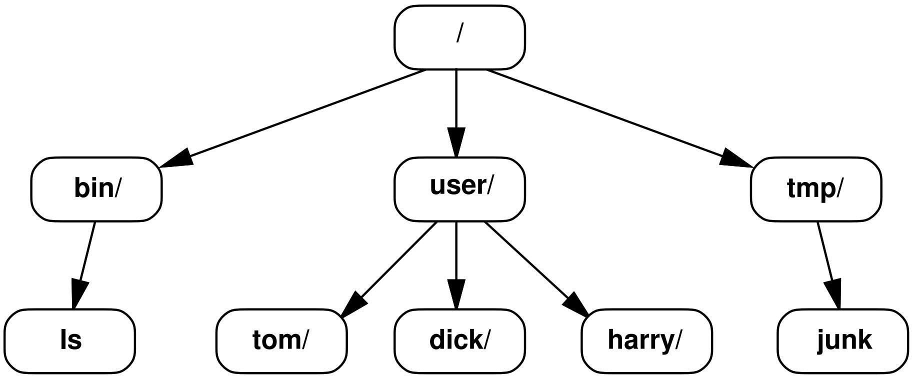
Design problems:

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

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

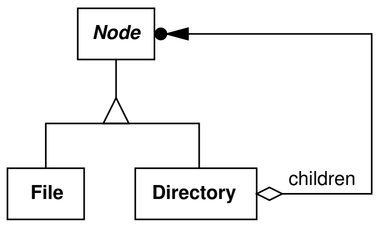
File System Structure (cont'd)

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 `newsubdirB`

File System Structure (cont'd)

Naive `mkdir` implementation

```
void Mkdir (Directory* current, const char* path) {
    const char* subpath = Subpath(path);

    if (subpath == 0) {
        current->Adopt(new Directory(path));

    } else {
        const char* name = Head(path);
        Node* child = Find(name, current);

        if (child) {
            Mkdir(child, subpath);
        } else {
            cerr << name << " nonexistent." << endl;
        }
    }
}
```


File System Structure (cont'd)

`Find` searches for a child with the given name

Must return a `Node*`

```
Node* Find (const char* name, Directory* current) {
    Node* child = 0;

    for (int i = 0; child = current->GetChild(); ++i) {
        if (strcmp(name, child->GetName()) == 0) {
            return node;
        }
    }
    return 0;
}
```

File System Structure (cont'd)

But `Mkdir` won't compile!

- `Mkdir` takes a `Directory*`, not a `Node*`
- Using `dynamic_cast` adds a control path

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

if (node) {
    Directory* child = dynamic_cast<Directory*>(node);

    if (child) {
        Mkdir(child, subpath);
    } else {
        cerr << GetName() << " is not a directory." << endl;
    }
} else {
    // ...
```


File System Structure (cont'd)

Solution: Treat `Adopt` and `Orphan` uniformly

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

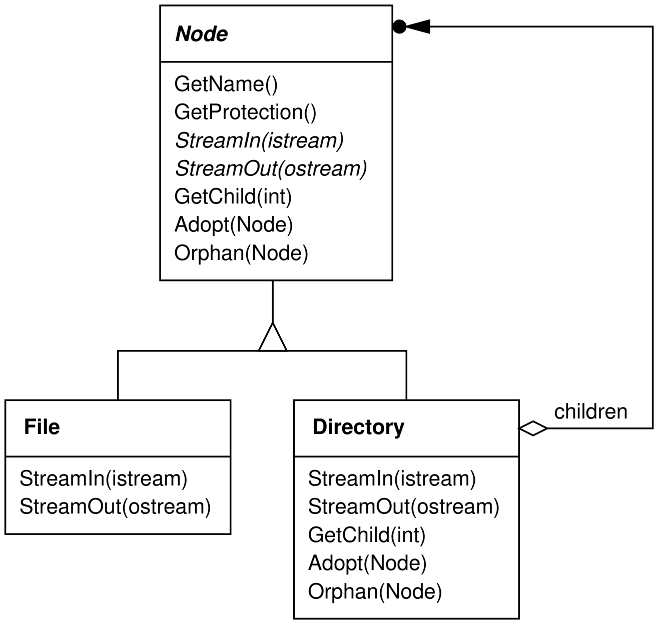
```
virtual void Node::Adopt (Node*) {
    cerr << GetName() << " is not a directory." << endl;
}

virtual void Node::Orphan (Node* child) {
    cerr << child->GetName() << " not found." << endl;
}
```

Only change to `Mkdir` is its signature:

```
void Mkdir (Node* current, const char* path) { ... }
```

File System Structure (cont'd)

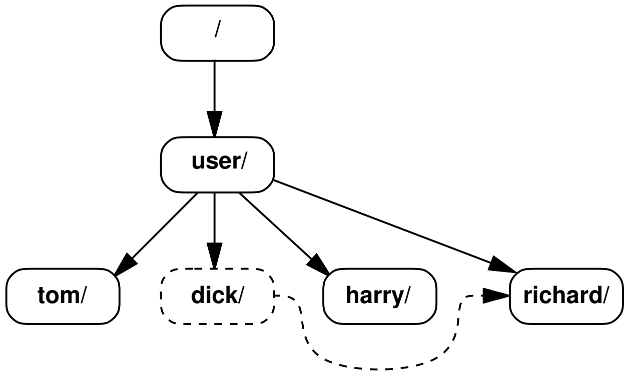
Result:



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)

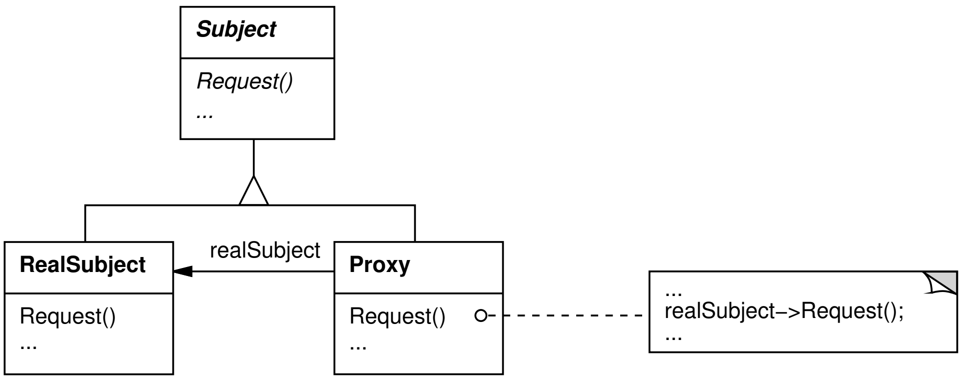
Symbolic Links (cont'd)

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:

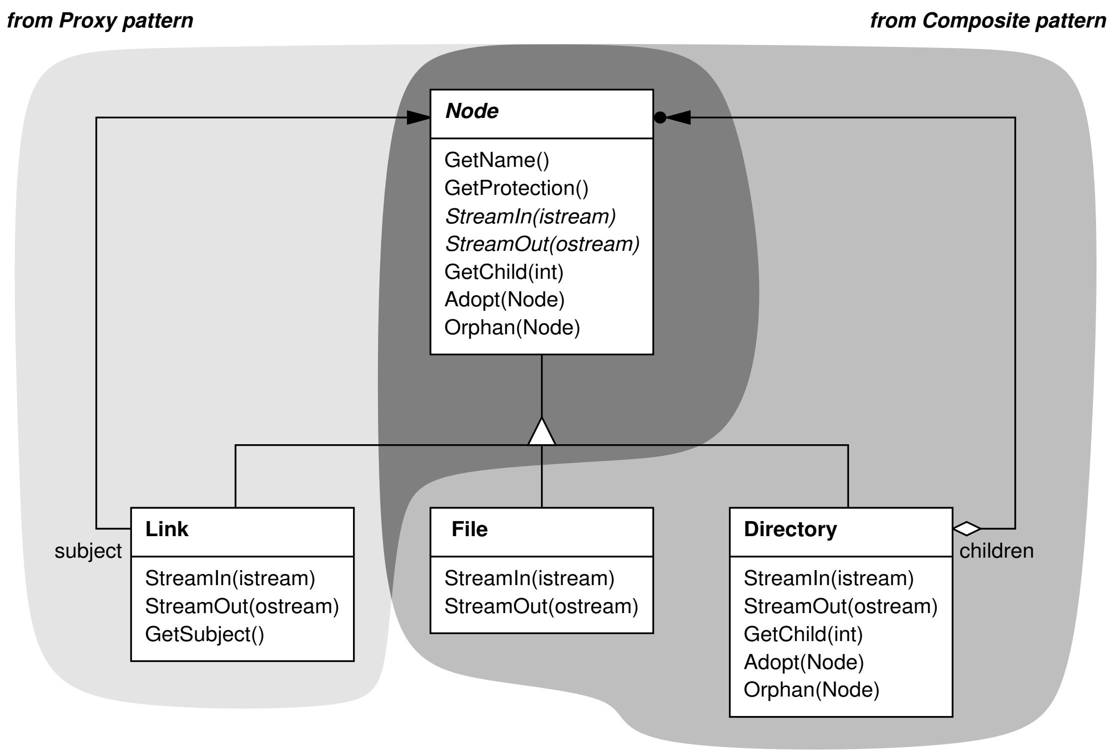
```
Node* Link::GetChild (int n) {
    return _subject->GetChild(n);
}
```

Additional Link-specific operation:

```
Node* Line::GetNode () { return _subject; }
```

(for clients who know they 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

Benefits:

- recovers type information without downcasts
- consolidates and encapsulates functionality in Visitor object

Drawbacks:

- new ConcreteElements may require changing Visitor interface
- circular dependency between Visitor and Element interfaces

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)

```
class NodeVisitor {
public:
    virtual void VisitNode(Node*)             { };  // default

    virtual void VisitFile(File* f)           { VisitNode(f); }
    virtual void VisitDirectory(Directory* d) { VisitNode(d); }
    virtual void VisitLink(Link* l)           { VisitNode(l); }

protected:
    NodeVisitor() { }
};
```

(avoid overloading to get around C++ quirk)

`Accept` operations:

```
void Node::Accept (NodeVisitor& v)      { v.VisitNode(this); }
void File::Accept (NodeVisitor& v)      { v.VisitFile(this); }
void Directory::Accept (NodeVisitor& v) { v.VisitDirectory(this); }
void Link::Accept (NodeVisitor& v)      { v.VisitLink(this); }
```

Open-Ended Functionality (cont'd)

`CatVisitor` subclass implementation

```
void CatVisitor::VisitFile (File* f) {
    f->StreamOut(cout);
}

void CatVisitor::Visit (Directory* d) {
    cerr << "Can't cat a directory." << endl;
}

void CatVisitor::Visit (Link* l) {
    l->GetNode()->Accept(*this);
}
```

Usage:

```
CatVisitor cat;
node->Accept(cat);
```


Open-Ended Functionality (cont'd)

What if Element hierarchy *isn't* stable?

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

```
void HardLink::Accept (NodeVisitor& v) { v.VisitNode(this); }
```

`VisitNode(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

Use RTTI:

1. In the element:

```
void HardLink::Accept (NodeVisitor& nv) {
    CatVisitor* cv = dynamic_cast<CatVisitor*>(&nv);

    if (cv) {
        cv->VisitHardLink(this);  // CatVisitor-specific op
    } else {
        nv.VisitNode(this);  // do the default
    }
}
```

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

Open-Ended Functionality (cont'd)

2. In the visitor:

```
void CatVisitor::VisitNode (Node* n) {
    HardLink* hl = dynamic_cast<HardLink*>(n);

    if (hl) {
        // do something special for hard links
    } else {
        NodeVisitor::VisitNode(n);  // do the default
    }
}
```

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

Single-User Protection

Problem:

- protect file system objects from inadvertent/malicious corruption
- make nodes (un)readible 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`
  → must disable destructor

What patterns support these (anti)responsibilities?

Single-User Protection (cont'd)

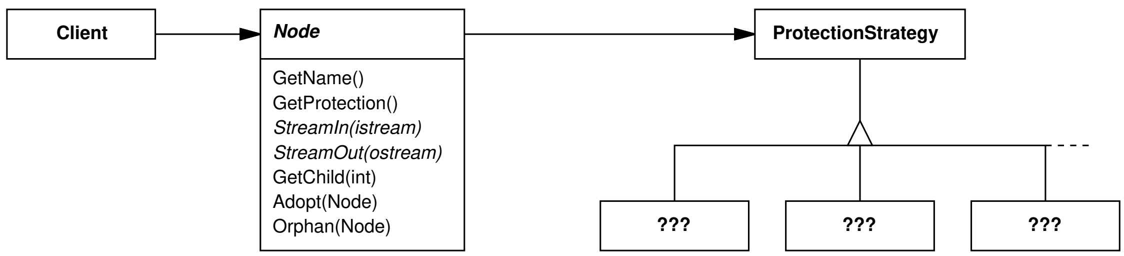
Observation 1: Protection behavior must vary *dynamically*

- 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



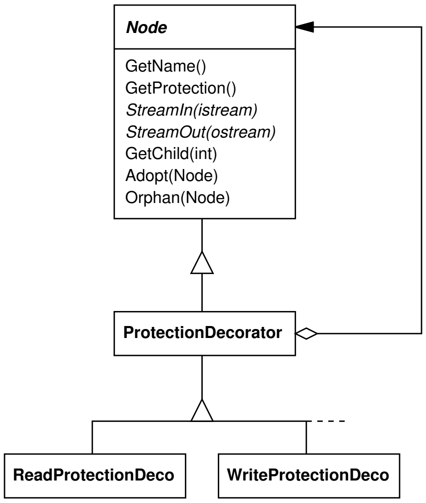
How many strategy objects does a node use?

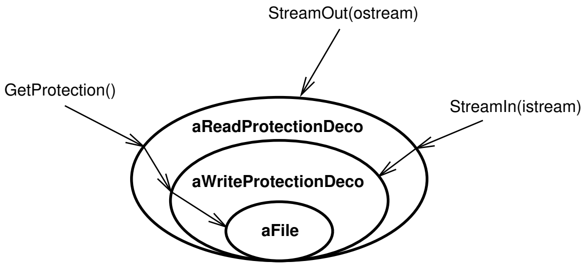
One per node, one per `Node` *operation*, ... ?
If multiple, how do they work together?
What do `Node` subclasses delegate to `ProtectionStrategies`?

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)

Single-User Protection (cont'd)

Typical implementation:

```
void Node::StreamOut (ostream& 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:

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...

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)

Modify SINGLETON's **Instance** operation a bit:

```
static const User* User::LogIn(
    const char* loginName, const char* password
) {
    if (password incorrect for loginName) {
        return 0;
    }

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

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


Multi-User Protection (cont'd)

Recap of `User::LogIn`'s properties

- globally accessible
- no more than one `User` object per login name
- can return 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 `const User*` in each call
   - e.g., `void SteamOut(ostream&, const User*);`
   - "stateful" design
   - can be a nuisance
2. Define an "implicit" `User`
   - signatures unchanged from single-user versions
   - "stateless" design
   - requires global `Set/GetUser` interface

Multi-User Protection (cont'd)

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);
```


Multi-User Protection (cont'd)

Typical impact on template methods:

```
void Node::StreamOut (ostream& out, const User* u) {
    User* user = u ? u : User::GetUser();

    if (IsReadableBy(user)) {
        DoStreamOut(out);
    } else {
        DoWarning(unreadableWarning);
    }
}

bool Node::IsReadableBy (const User* user) {
    bool isOwner = user->Owns(this);
        // true iff user's login name matches node's owner

    return
        isOwner && IsUserReadable() ||
        !isOwner && IsOtherReadable();
}
```

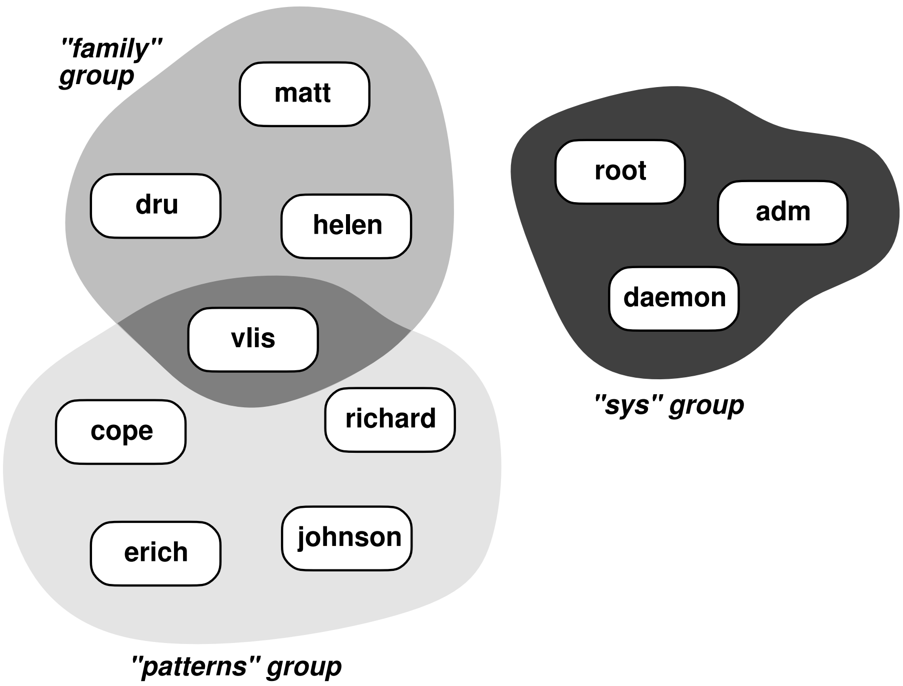
Multi-User Protection (cont'd)

More questions:

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

Multi-User Protection (cont'd)

Example grouping:

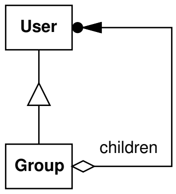


Groups as objects

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

Assume `User` subclass

A Composite of users?



Why COMPOSITE *isn't* 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

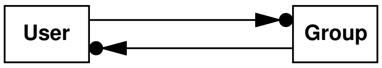
Need a *two-way* mapping for efficiency

- \# users $\gg$ # groups
  $\rightarrow$ find all users 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

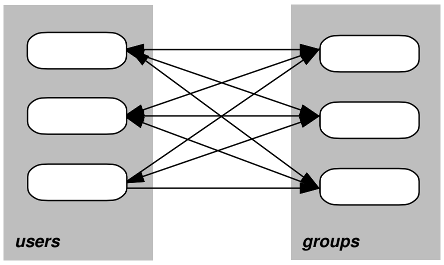
- all objects saddled with cost of a (pointer to a) collection

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

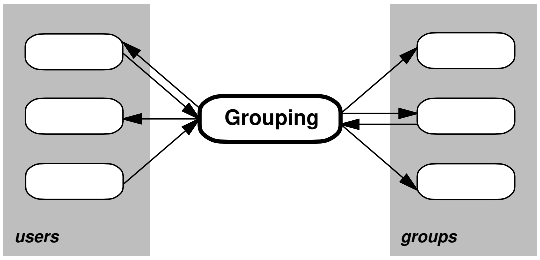
Multi-User Protection (cont'd)

Managing object interconnections → MEDIATOR

Before:



After:



`Grouping` is probably a singleton

Multi-User Protection (cont'd)

`Grouping` interface

```
class Grouping {
public:
    virtual void ~Grouping();

    static const Grouping* GetGrouping();
    static void SetGrouping(const Grouping*, const User* = 0);

    virtual void Register(
        const User*, const Group*, const User* = 0
    ) = 0;
    virtual void Unregister(
        const User*, const Group*, const User* = 0
    ) = 0;

    virtual const Group* GetGroup(const User*, int index = 0) = 0;
    virtual const User* GetUser(const Group*, int index = 0) = 0;
protected:
    Grouping();
};
```

Summary

Structure: COMPOSITE

Symbolic links: PROXY

Open-ended functionality: VISITOR

Single-user protection: TEMPLATE METHOD
Not STRATEGY, DECORATOR, or PROXY

Multi-user protection:

- MEDIATOR: groups
  Not COMPOSITE

- SINGLETON: creating users/ "implicit" user/`Grouping`

Part 2: **Redesigning with Patterns**

```
[1 hr, 30 min, Richard]

        Initial design of an OODB application
            Hardware configuration managment system
            Targets for redesign

        Hardware structure (Composite)

        Database entry point (Singleton)

        Persistent Memory Allocation (Abstract Factory + Strategy)

        Modeling transactions (Commands)

        Dependency management between persistent and transient data
            (Observer)

        View creation (Factory Method + Visitor)

        Configurability (Instance-Type)
```

Summary

Part 3: **Refining with Patterns**

[45 min, Richard]

Debugging, testing, tuning

Intercepting messages
- Decorator
- Mediator + Strategy

Instrumentation
- Events (Command/Courier)
- Transmission across address spaces (Proxy)

Decoupling system from instrumentation
- Observer + Mediator

Part 4: **Communicating with Patterns**

```
[1 hr, 45 min, Erich]

        Exposition

        Java AWT
            Composite
            Bridge
            Decorator
            Chain of Responsibility
            Abstract Factory
            Command

        Financial Engineering
            Strategy
            Composite

        Portability
            TBD

        MFC
            Prototype (Document templates)
```



```
            Factory Method
            Observer

        Web-based documentation
```

Conclusion

```
[5 min, Erich]

        Review, observations, references
```


(Design) Pattern References

Books:

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

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

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides; Addison-Wesley, 1995;
ISBN 0-201-63361-2

Pattern Languages of Program Design, Coplien, Schmidt, eds.; Addison-Wesley, 1995; ISBN 0-201-60734-4

Pattern Languages of Program Design 2, Vlissides, Coplien, Kerth, eds.; Addison-Wesley, 1996; ISBN 0-201-89527-7

Object Models: Strategies, Patterns, and Applications, Coad, North, Mayfield; Prentice-Hall, 1995; ISBN 0-13-108614-6

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 97: Pattern Languages of Programs
Wednesday-Friday, 3-5 September 1997, Monticello, Illinois, USA

EuroPLoP 97
July 1997 (TBD)

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*

`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

URLs:

General
`http://st-www.cs.uiuc.edu/users/patterns/patterns.html`

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

Conferences
`http://st-www.cs.uiuc.edu/users/patterns/Conferences.html`

PLoP Information
`http://www.cs.wustl.edu/~schmidt/jointPLoP-96.html`

Pattern Languages of Program Design
`http://www.research.att.com/orgs/ssr/book/PLoP94/index.html`

Pattern Languages of Program Design 2
`http://st-www.cs.uiuc.edu/users/patterns/PLoPD2/index.html`