Design Patterns: A Java Programmer's Perspective

Ray Grimmond

Christie Whitesides

Threshold Computer Systems, Inc.

ray@thresholdobjects.com

Agenda

- Introduction to Patterns & Design Patterns
- Importance of Patterns
- Case Study - Consists of 6 individual patterns combined in single solution
- Patterns are Everywhere

History of Patterns

- Where do they come from?

  Christopher Alexander is a building architect and author of the following architecture books. The first book, *Timeless Way*, took 14 years to complete and was published in 1979.

  – *The Timeless Way of Building*
  – *A Pattern Language*
  – *Nature of Order* - latest work, to be published soon

Christopher Alexander's World of Patterns

- **What does it all mean?**

"Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice."

Christopher Alexander

Christopher Alexander's World of Patterns (Continued)

- More definitions
  - Each pattern is a 3 part rule, which expresses a relation between a certain context, a problem, and a solution.
  - Each pattern is at the same time, a thing which happens in the world, and the rule which tells us how to create that thing, and WHEN we must create it.
  - It is both a PROCESS and a THING.

Christopher Alexander, *Timeless Way of Building*, p 247.

Colorado Software Summit

How Did We Go from Buildings to Software?

- Erich Gamma's Ph.D thesis
- OOPSLA '91
- Knuth - *Art of Computer Programming*
- Coplien - *Advanced C++: Programming Styles & Idioms*
- *Design Patterns* - GofF (Gamma, Helm, Johnson, Vlissides)
- PLoP I , 2 , 3, 4 , EuroPlop and Chiliplop Conferences
- Countless Books on Patterns

Colorado Software Summit

Pattern Form

- Pattern can be expressed or written in a variety of different forms. Several pattern proponents have come up with there own literary form.
  - Alexandrian
  - Gang of Four
  - Coplien
  - Portland

Alexandrian Form

- Name
  - A short noun or noun phrase (sometimes a verb phrase)
- Context
  - Alexander's introductory paragraph sets the context of a pattern.
  - Problem and solution apply to context.
- Problem
  - The design challenge
- Solution
  - Instructions to solve the problem
  - Could be accompanied by a sketch

Gang of Four - Design Patterns

- Abstracts a recurring design structure
- Design pattern has 4 basic parts
  - Name
  - Problem
  - Solution
  - Consequences

Gang of Four - Template

- Name
  - What is it
- Intent
  - Description of pattern and purpose
- Motivation
  - Alexander's - Problem, Context, Solution
- Applicablity
  - Circumstances in which pattern applies
- Structure
  - Graphical representation of pattern
- Participants
  - Classes, objects and their responsibilities

Gang of Four - Template (Continued)

- Collaborations
  - How participants carry out their responsibilities
- Consequences
  - The results of application, benefits, liabilities
- Implementation
  - Traps, hints , techniques, plus language dependent issues
- Sample code
  - Sample implementations
- Known uses
  - Examples from existing systems
- Related patterns
  - Discussion of other patterns that relate

Patterns Are Not ...

- Algorithms
  – Pattern-like
  – Takes the functional view
- Idioms
  – Pattern-like
  – Describe language specific techniques
- Frameworks
  – More concrete
  – Only apply in a particular domain

Why Are Patterns Important to Java Programmers?

- Capture, communicate and apply design knowledge
  – Your own or other people's
- Build consensus
  – Patterns are shared by a community
  – Shared vocabulary
  – Effective way of communicating with clients, peers, and customers
- Reflecting more and creating rationales
  – Promotes "thought" rather than "action", working awarely
  – Artifacts and processes
  – Expressions and problem solving
- Allow potential for design re-use
- Build easily adaptable solutions

What Is Important in a Design Pattern to a Java Programmer

- **Name and Intent**
  - Identifies the design pattern and tells us what the pattern does and the design problem it attempts to solve

What Is Important in a Design Pattern to a Java Programmer

- Motivation
  - This section represents the design problem and outlines the solution to the design problem.
  - It can be viewed as the classical Alexander statement of problem, solution, and context. But it also goes further and discusses the classes and objects within the pattern and how they solve the design problem.

What Is Important in a Design Pattern to a Java Programmer

- Applicability
  - Horses for Courses !! - Can the pattern be applied in your situation, can a modified pattern work any better ?
- Forces
  - Understand the forces (or trade-offs) to effectively apply the pattern. If you understand the forces, then you understand the problem and the solution.

What Is Important in a Design Pattern to a Java Programmer

- Structure
  - Keep mental picture of the class diagrams.
  - Look at the class diagrams with the concrete example first.
  - Examine the abstract structure diagram and look for the relationships between the participants, common methods, abstract *vs.* concrete classes, aggregation, differences with other patterns.

Colorado Software Summit

What Is Important in a Design Pattern to a Java Programmer

- Participants
  - Look at their names - lots of meaning is intentionally or unintentionally conveyed.
  - Avoid making too many inferences from the names alone.
- Roles and Responsibilities
  - Examine the roles played by each participant, view them as actors in a play.. "When can they speak and what can they say and to whom."

What Is Important in a Design Pattern to a Java Programmer

- Relationships between participants
  – Closely examine the relationships between participants
  – A relationship that doesn't or shouldn't exist is just as important as one that does or should exist.

What Is Important in a Design Pattern to a Java Programmer

- Consequences
  - This is real important section.
  - It normally examines the trade-offs, benefits and liabilities associated with applying the pattern.
  - Check to see if there are any unacceptable consequences by using this pattern.

How Do I Get Started with Design Patterns?

- Personal Experience - No Silver Bullet
  - The following has worked for me - but hindsight is wonderful.
- Getting Started
  - Remember - Name, Problem, Context, Solution.
  - DON'T be overwhelmed by the amount of information available. Examine 2 or 3 patterns at a time.
  - Quickly review pattern catalog/names, look for one that fits.

Pattern Roadmap

- Scan Names and Intent for something that feels right.
- Look for a pattern with a similar purpose (creational , structural, behavioral)
- Examine redesign cause, and apply the patterns that help avoid it.
- Look at any examples, examine the structure and the participants

A Design Pattern Catalog

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory	families of product objects
	Builder	how a composite object gets created
	Factory Method	subclass of object that is instantiated
	Prototype	class of object that is instantiated
	Singleton	the sole instance of a class
Structural	Adapter	interface to an object
	Bridge	implementation of an object
	Composite	structure and composition of an object
	Decorator	responsibilities of an object without subclassing
	Facade	interface to a subsystem
	Flyweight	storage costs of objects
	Proxy	how an object is accessed; its location

A Design Pattern Catalog

Purpose	Design Pattern	Aspect(s) That Can Vary
Behavioral	Chain of Responsibility	object that can fulfill a request
	Command	when and how a request is fulfilled
	Interpreter	grammar and interpretation of a language
	Iterator	how an aggregate's elements are accessed, traversed
	Mediator	how and which objects interact with each other
	Memento	what private information is stored outside an object, and when
	Observer	number of objects that depend on another object, how the dependent objects stay up to date
	State	states of an object
	Strategy	an algorithm
	Template Method	steps of an algorithm
	Visitor	operations that can be applied to objects(s) without changing their class(es)

Redesign Causes

- Creating objects with explicit class names
- Hard-coded operations
- Hardware and OS dependencies
- Code tied to object reps & implementations
- Algorithmic dependencies
- Tight coupling
- Too many subclasses
- Altering someone else's monolithic mess

Colorado Software Summit

Understanding Design Patterns

- Start concrete and go abstract
  - Get familiar with the pattern Name and Intent, examine the Motivation section (problem and context). Focus on the problem, and the solution to that problem.
- Samples and more samples
  - Review as many samples as you can find for a given pattern (even those in other languages). Understand and review the implementation trade-offs section and what they mean. Learn by example and differences.

Understanding Design Patterns (Continued)

- Check applicability
  - Once you have an idea of the pattern's intent and the problems it solves, see if it is applicable to your context.
  - Does this solution solve your problem ??
- Go abstract
  - Review the Structure, Participants , Collaboration and Consequences sections of the pattern.

Understanding Design Patterns (Continued)

- Go back to being Concrete
  - Take another look at the implementation trade-offs and examples.
  - Then try and apply the pattern and write your code.
  - Not working out ?? Go back to the beginning and start again, maybe check out some new patterns, or try the roadmap approach.

File System API Example

- Simple example common in CS101
  - Write a File system - something we all understand
- Focus on problems
  - Look at the design problems we wish to overcome
- Focus on solution for that problem

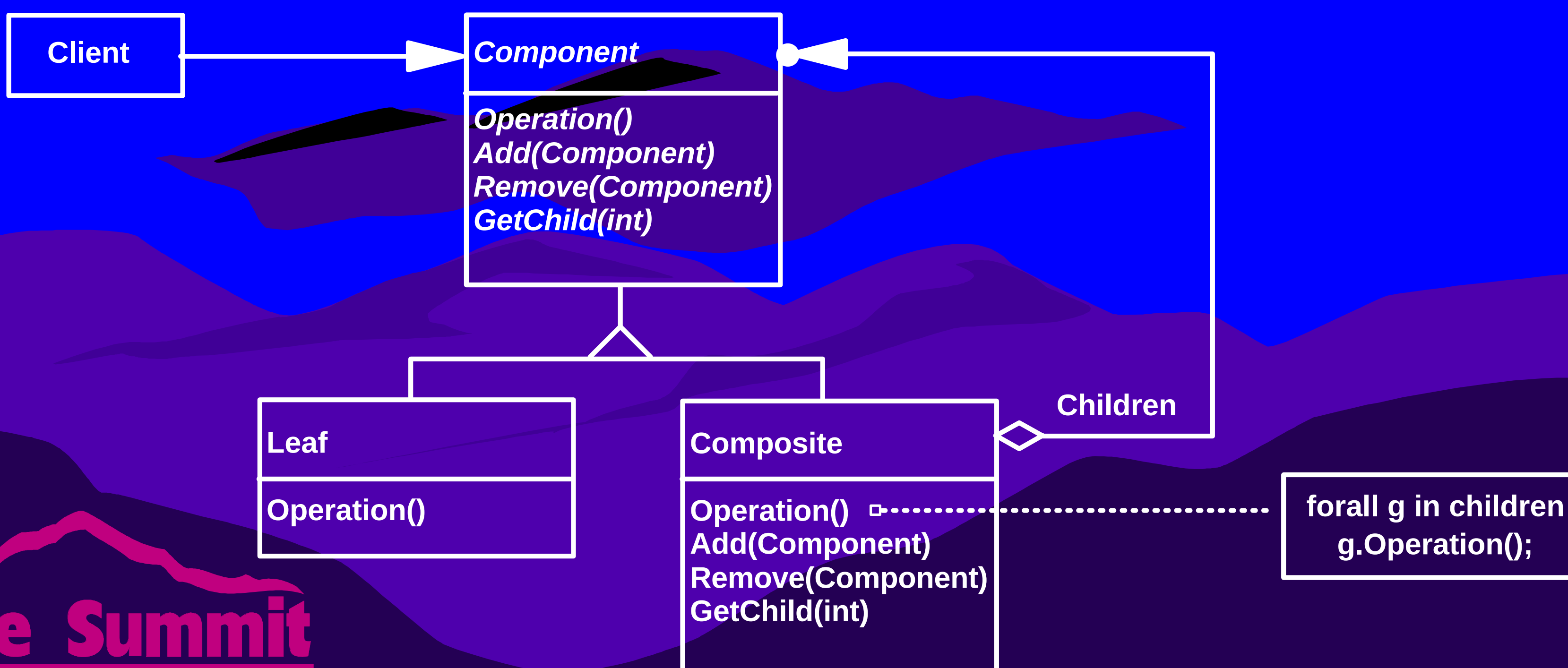
- Remember - there are an infinite number of solutions - applying patterns is discovery

File System API Example - Pattern#1

- Design problem
  - Handle scalable and complex file system structures
  - Easy to maintain
  - Have common properties like size and name
  - Need to treat objects uniformly - allows recursion
- Solution - use Composite pattern

Composite

- Intent
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Structure



Composite Participants

- Component
  - Decares interface for objects in the composition
  - Implements default behavior in the interface commom to all classes
  - Declares an interface for accessing and managing child components (optional) Declares an interface for accessing a component's parent
- Leaf
  - Represents leaf objects; has no children
  - Defines behavior for primitive objects in the composition
- Composite
  - Defines behavior for components having children
  - Stores child components
  - Implements child-related operations in the Component interface
- Client
  - Manipulates objects in the compositon through the Component interface

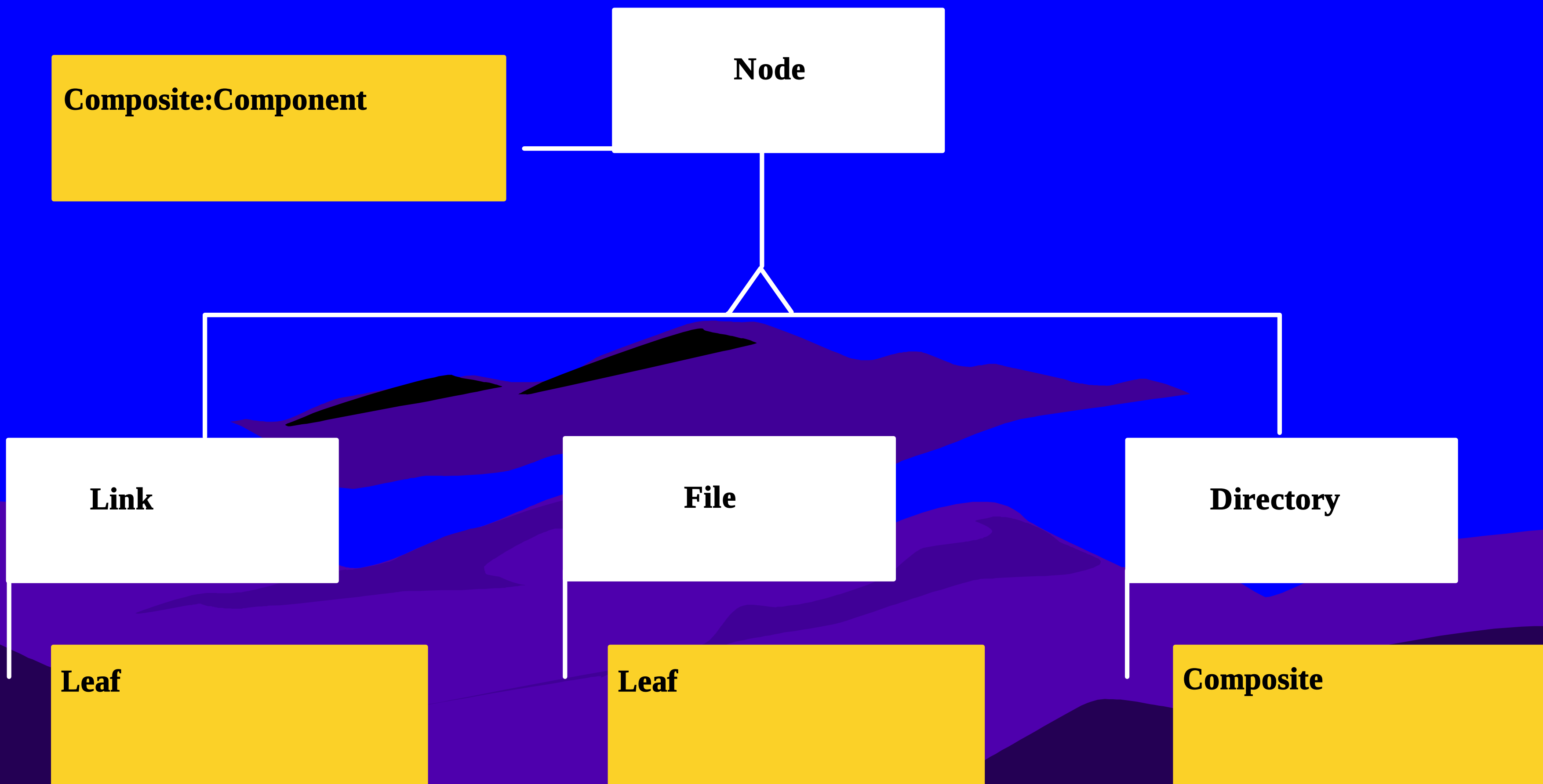
Composite Sample Code

- **Composite.java**

```
interface Component {
     void operation();                //  supply method name
     void add(Component c);           //  ..
     void remove(Component c);        //  ..
     Component getChild(int a );      //  ..
};
class Composite implements Component {
     public Component[] g;
     public void operation() {;}      // g.operation
     public void add(Component c) {;}
     public void remove(Component c) {;}
     public Component getChild(int a ) { return g[a]; }
};
class Leaf implements Component {
     public void operation() { ; }
     public void add(Component c) {;}
     public void remove(Component c) {;}
     public Component getChild(int a ) { return null; }
};
class Client {
     void clientMethod() {
          Component x = new Leaf();
          Component y = new Composite();
     }
}
```

Case Study - FileSystem

- (See FileSys1.java)

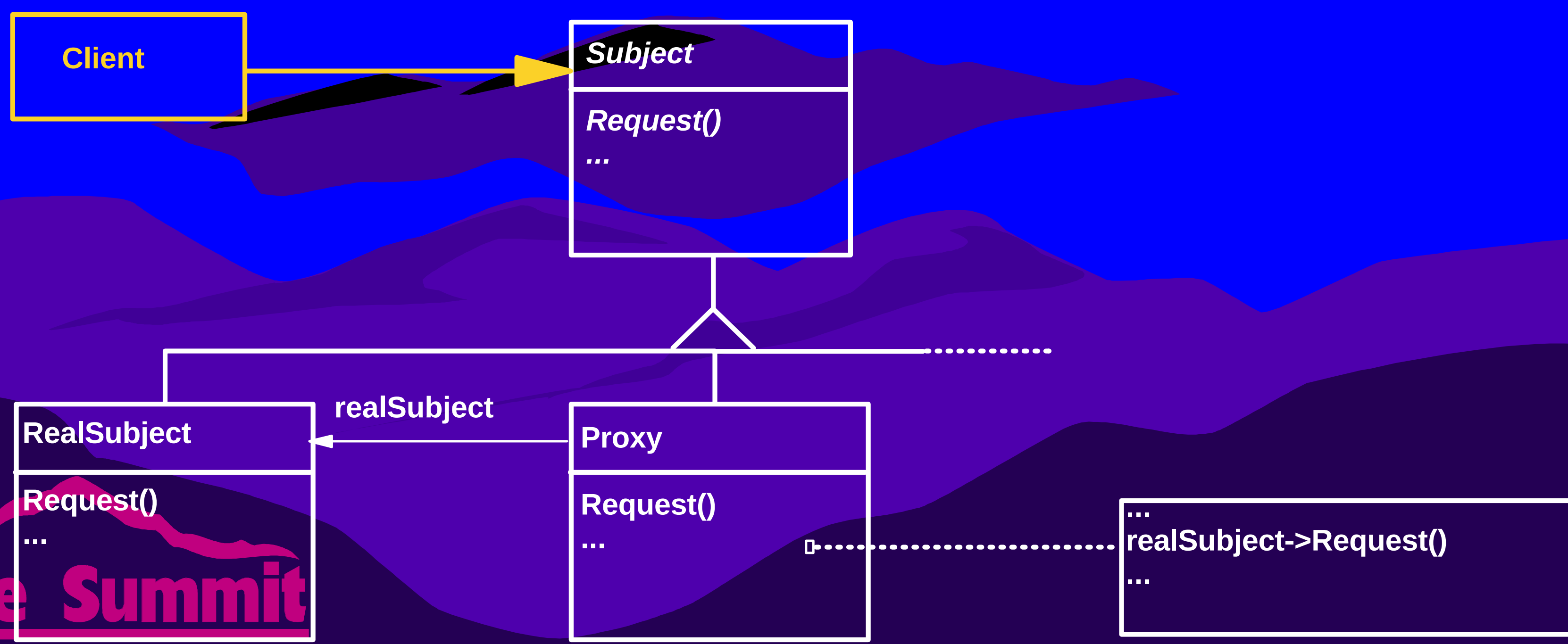


File System API Example - Pattern #2

- Design problem
  - Symbolic links, "shortcuts" or aliases
- Solution - use Proxy pattern

Proxy

- Intent
  - Provide a surrogate or placeholder for another object to control object to control access to it.
- Structure



Proxy Participants

- Proxy
  - Maintains a reference that lets the proxy access the real subject.
  - Provides an interface identical to Subject's
  - Controls access to the real subject
  - *Remote proxies* encoded messages sent to a different address space
  - *Virtual proxies* cache information for postponed access to real subject.
  - *Protection proxies* checks callers access permissions.
- Subject
  - Defines the common interface for RealSubject and Proxy
- RealSubject
  - Defines the real object that the proxy represents

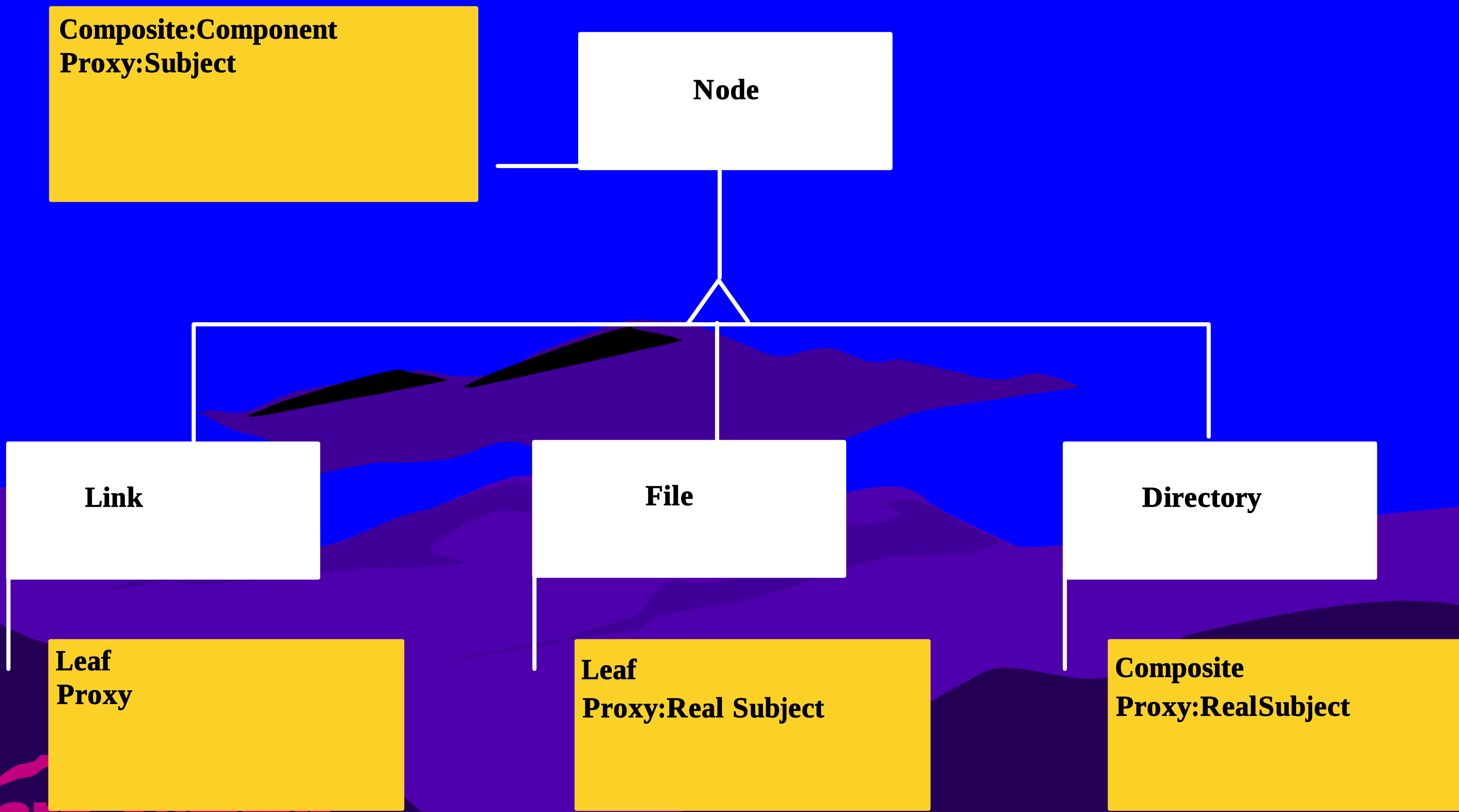
Proxy Sample Code

- **Proxy.java**

```
interface Subject {
    void request();
    void request2();
}

class Proxy implements Subject {
    RealSubject realSubject;
    Proxy() {
        realSubject = new RealSubject();
    }
    public void request() { realSubject.request(); }
    public void request2() { realSubject.request2(); }
};

class RealSubject implements Subject {
    public void request()  {;}
    public void request2() {;}
};

class Client {
    public static void main(String[] args) {
        Proxy p = new Proxy();
    }
}
```

Case Study - FileSystem

- (See FileSys2.java and FileSys2chg.java)



File System API Example - Pattern #3

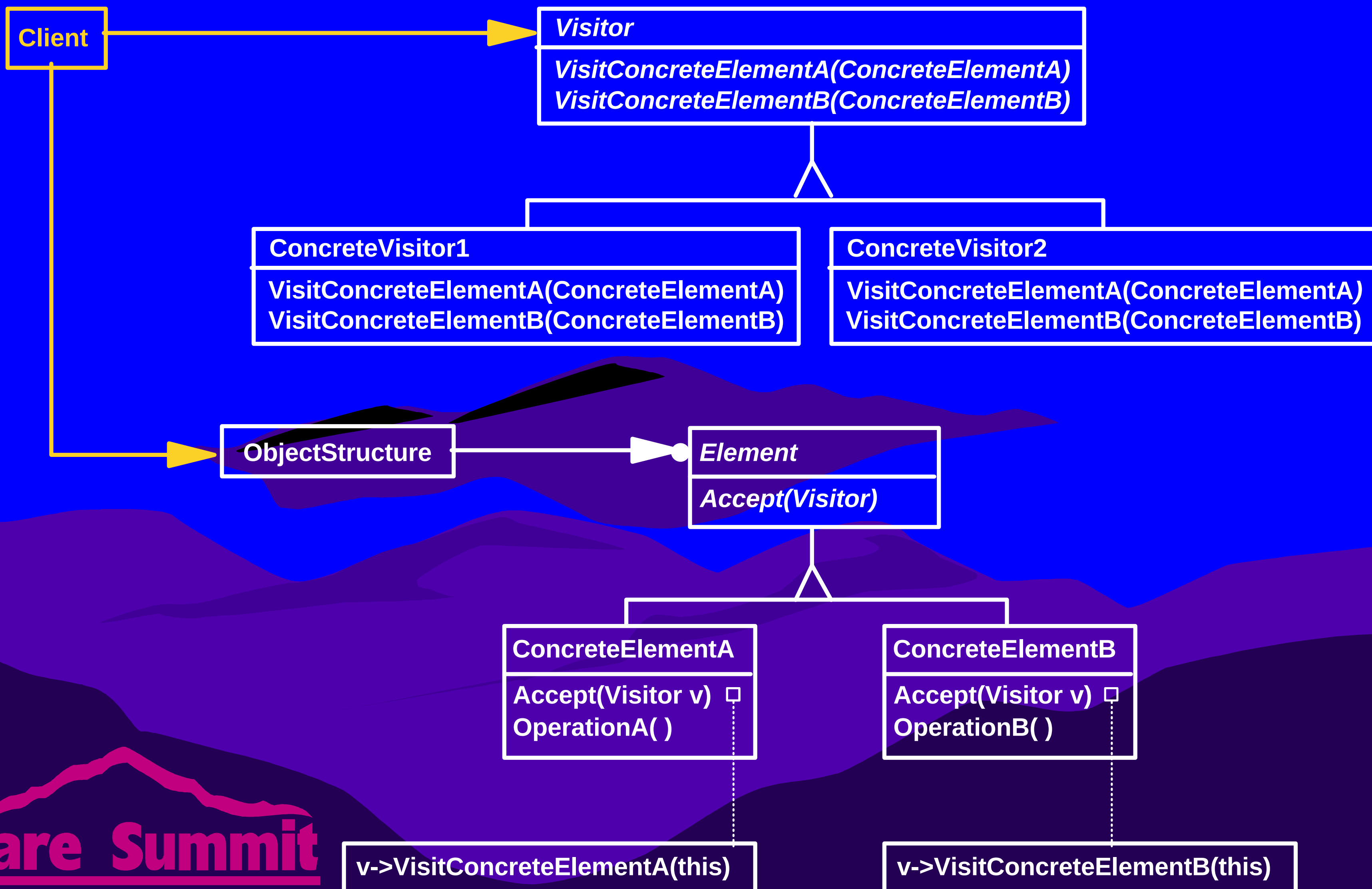
- Design Problem
  - Adding more and more features causes code-bloat in base class node
- Solution - use Visitor pattern

Visitor

- Intent
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor (Continued)

- Structure



Visitor Participants

- Visitor
  - Declares a Visit operation for each class of ConcreteElement
- ConcreteVisitor
  - Implements each operation declared by Visitor
- Element
  - Defines an Accept operation that takes a visitor as an argument.
- ConcreteElement
  - Implements an Accept operation that takes a visitor as an argument.
- ObjectStructure
  - Can enumerate its elements.
  - May provide a high-level interface allowing the visitor to visit elements.
  - May either be a composite or a collection such as a list or a set.

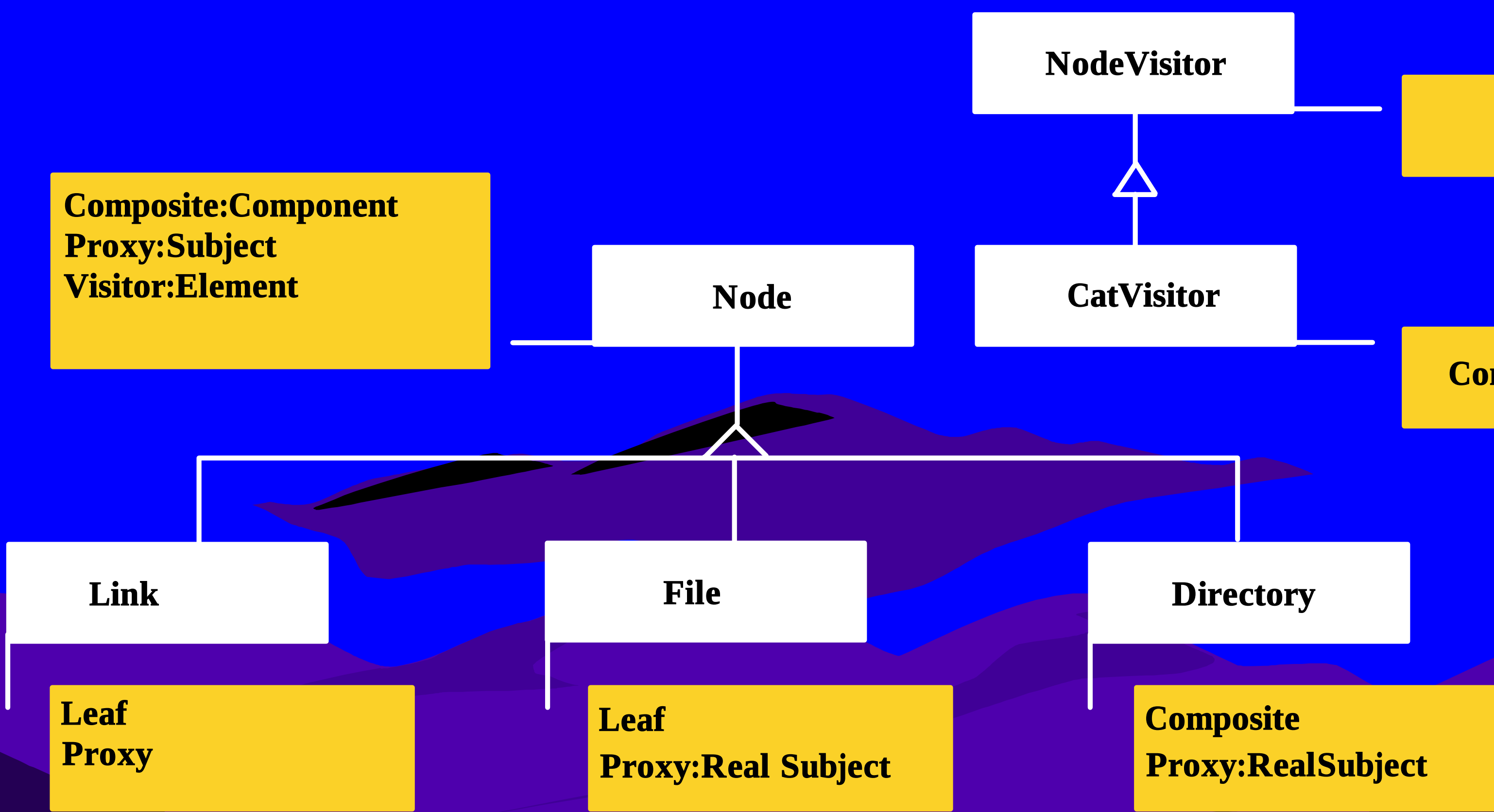
Visitor Sample Code

- **Visitor.java**

```
abstract class Visitor {
      abstract void VisitConcreteElementA(ConcreteElementA a);
      abstract void VisitConcreteElementB(ConcreteElementB b);
}
class ConcreteVisitor1  extends Visitor {
      void VisitConcreteElementA(ConcreteElementA a) { ; }
      void VisitConcreteElementB(ConcreteElementB b) { ; }
}
class ConcreteVisitor2 extends Visitor {
      void VisitConcreteElementA(ConcreteElementA a) { ; }
      void VisitConcreteElementB(ConcreteElementB b) { ; }
}
class ObjectStructure {
      Element[] e;
      Visitor v = new ConcreteVisitor1();
      int len=e.length;
      ObjectStructure() {
            for(int i=0;i..len; i++)
                  e[i].Accept(v);
      }
}
class Element {
      void Accept(Visitor v) { ; }
}
class ConcreteElementA extends Element {
      void Accept(Visitor v) { v.VisitConcreteElementA(this);}
      void OperationA() {;}
}
class ConcreteElementB extends Element {
      void Accept(Visitor v) { v.VisitConcreteElementB(this);}
      void OperationB() {;}
}
```

Case Study - FileSystem

- (See FileSys3.java and FileSys3chg.java)

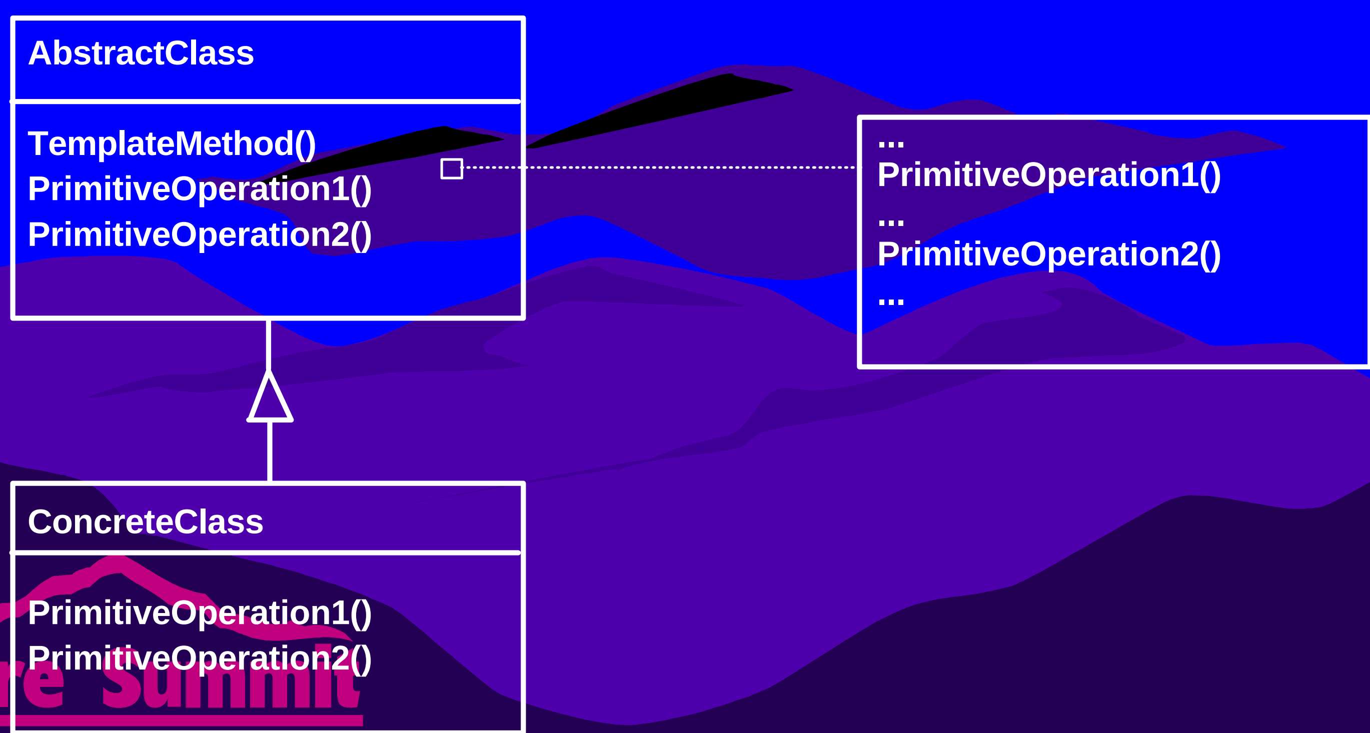


File System API Example - Pattern #4

- Design problem
  - Security policies
- Solution - use Template Method pattern

Template Method

- **Intent**
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Structure**



Template Method Participants

- AbstractClass
  - Defines abstract primitive operations
  - Implements a template method defining the skeleton of an algorithm.

  The template method calls primitive operations as well as operations defined in AbstractClass or other objects.

- ConcreteClass
  - Implements primitive operations

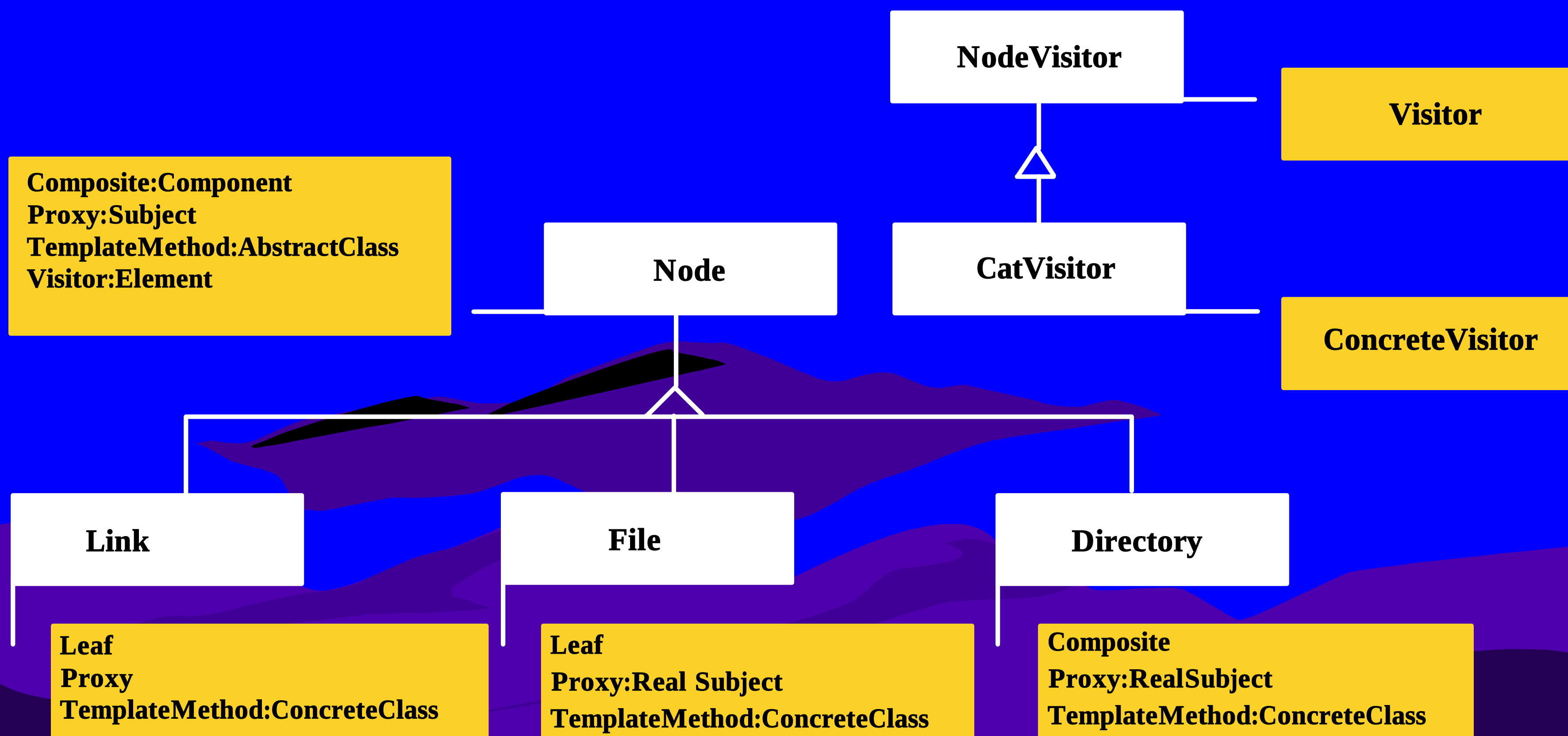
Template Method Sample Code

- **TemplateMethod.java**

```
abstract class AbstractClass {
    void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
    }
    abstract void primitiveOperation1();
    abstract void primitiveOperation2();
}
class ConcreteClass extends AbstractClass {
    void primitiveOperation1() { ; }  // implement operation 1
    void primitiveOperation2() { ; }  // implement operation 2
}
class Client {
    public static void main(String[] args) {
        AbstractClass x = new ConcreteClass();
        x.templateMethod();
    }
}
```

Case Study - FileSystem

- (See FileSys4.java and FileSys4chg.java)

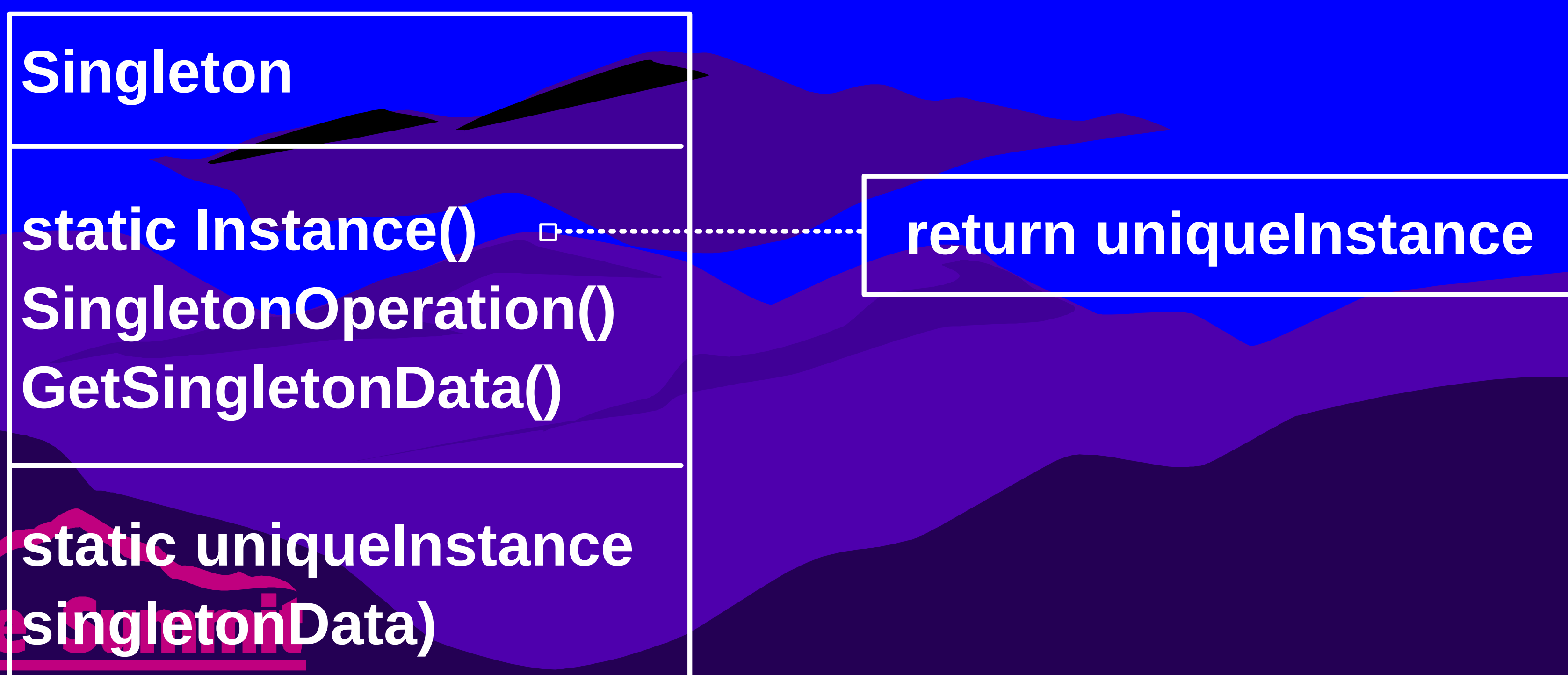


File System API Example - Pattern #5

- Design problem
  - Multi-level protection
- Solution - use Singleton pattern

Singleton

- Intent
  – Ensure a class only has one instance, and provide a global point of access to it.
- Structure



Singleton Participants

- Singleton
  - Defines an Instance operation that lets clients access its unique instance.
  - May be responsible for creating its own unique instance.

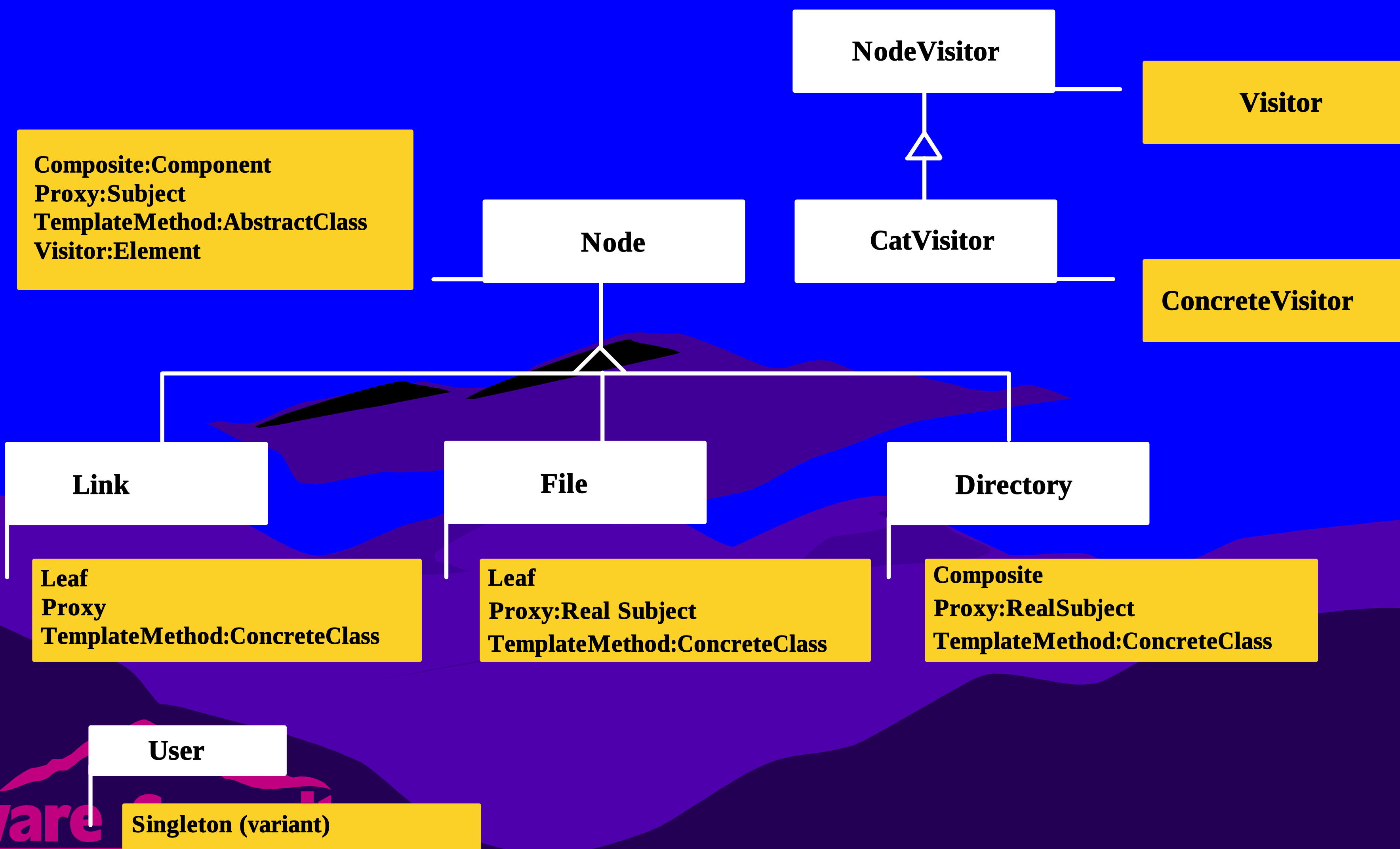
Singleton Sample Code

- **Singleton.java**

```
class SingletonData {;}

class Singleton {
    Singleton() {
    }

    static Singleton Instance()  {
        if(uniqueInstance == null)
            uniqueInstance = new Singleton();
        return uniqueInstance ;
    }

    SingletonData GetSingletonData() {
        if(singletonData == null)
            singletonData = new SingletonData();
            return singletonData;
    }

    static Singleton uniqueInstance=null;
    static SingletonData singletonData=null;
}
```

Case Study - FileSystem

- (See FileSys5.java and FileSys5chg.java)

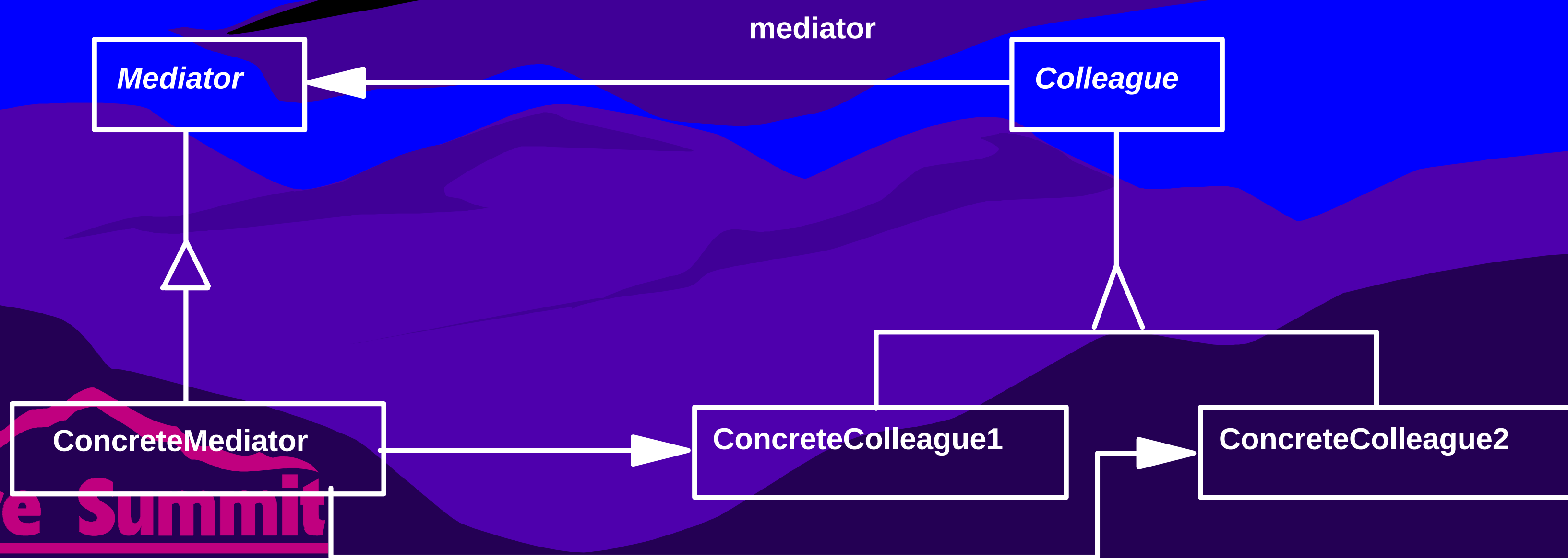


File System API Example - Pattern #6

- Design problem
  - Associating users and groups
- Solution - use Mediator pattern

Mediator

- Intent
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Structure



Mediator Participants

- Mediator
  - Defines an interface for communicating with Colleague objects.
- ConcreteMediator
  - Implements cooperative behavior by coordinating Colleague objects.
  - Knows and maintains its colleagues.
- Colleague classes
  - Each Colleague class knows its Mediator object.
  - Each colleague communicates with its mediator instead of a colleague.

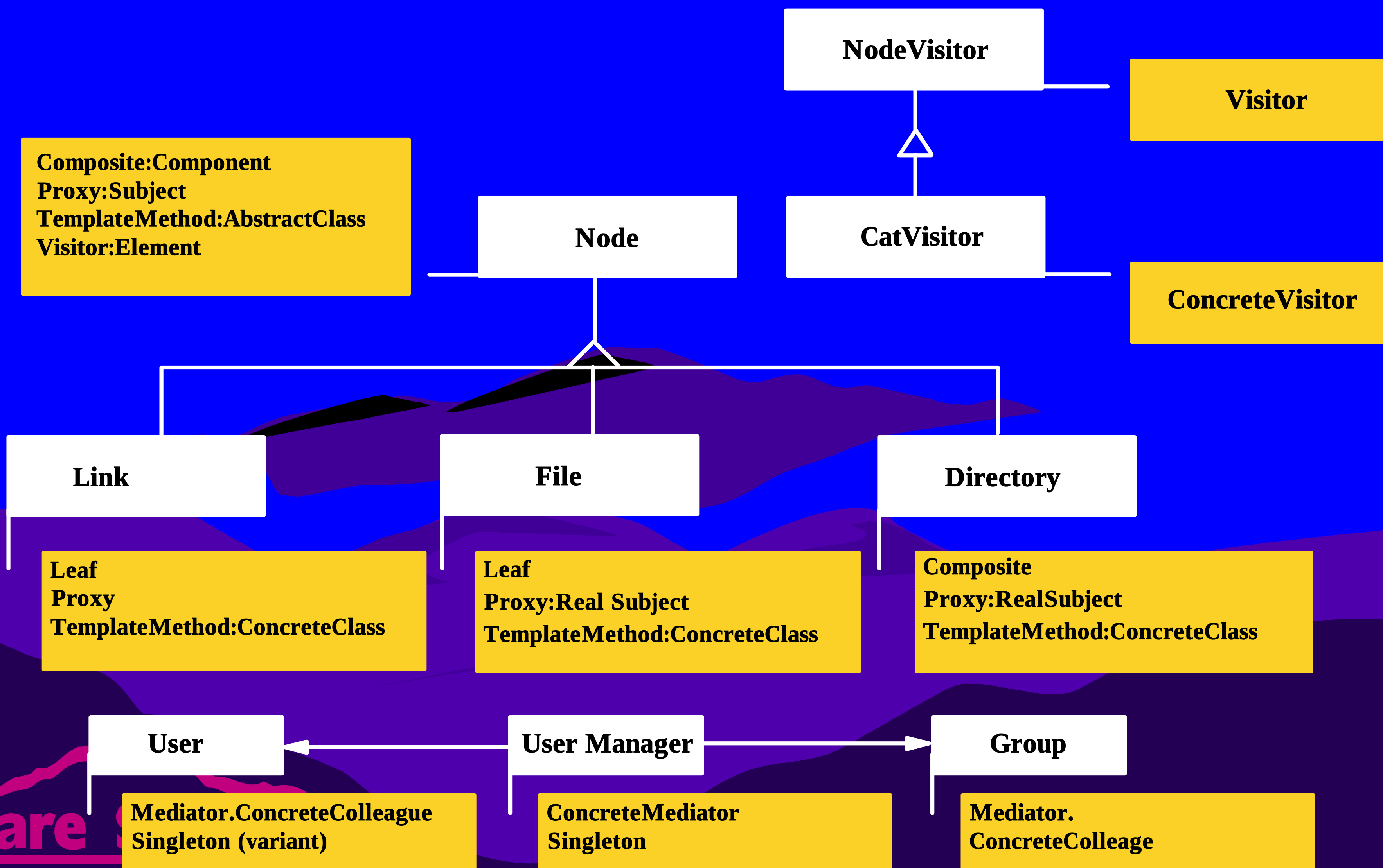
Mediator Sample Code

- **Mediator.java**

```
abstract class Mediator {
    abstract void methodA(Colleague c);
}
class ConcreteMediator1  extends Mediator {
    void methodA(Colleague c){;}
}
class ConcreteMediator2  extends Mediator {
    void methodA(Colleague c){;}
}
abstract class Colleague {
    Mediator mediator;
    Colleague(Mediator m) { mediator=m; }
    void changed() { mediator.methodA(this); }  // passes self option
}
class ColleagueA extends Colleague {
    ColleagueA(Mediator m) { super(m); }
}
class ColleagueB extends Colleague {
    ColleagueB(Mediator m) { super(m); }
}
```

Case Study - FileSystem

- (See FileSys6.java and FileSys6chg.java)



Patterns Are Everywhere

- Core Java
  - Bridge
    - java.io.Button and java.io.ButtonPeer ….*etc.*
  - Decorator
    - java.io.FilterStream
  - Composite
    - java.awt.Component , java.awt.Container
    - java.awt.Component subclasses; java.awt.Button, java.awt.Canvas
  - Strategy
    - java.awt.Container, java.awt.LayoutManager
  - Abstract Factory & Singleton
    - java.awt.Toolkit
  - Iterator
    - java.util.Iterator and java.util.Dictionary

Patterns Are Everywhere
(Continued)

- Swing - has same patterns as awt +
  - Composite
    - swing.text.Element, swing.text.View, swing.text.Document classes
  - Factory
    - swing.text.ViewFactory
  - AbstractFactory
    - Swing Look and Feel classes
  - + many more

Patterns Are Everywhere
(Continued)

- San Francisco project - GofF based
  - AbstractFactory and Command
  - Property Container (based on Composite)
  - Policy - Strategy derivative
  - Chain of Responsibitity Driven Policy (CofR derivative)
  - Generic Interface - (Facade derivative)
  - Controller - (based on Mediator)
  - Life Cycle - (based on State)

Patterns Are Everywhere
(Continued)

- San Francisco project - Unique Patterns
  - Keys and Keyables
  - Cached Balances
  - Keyed Attribute Retrieval
  - Extensible Item
  - Hierarchy Level Information
  - Ables and Ings

Patterns Are Everywhere
(Continued)

- *Concurrent Programming* - Doug Lea
  - Part of the Java Series books - ISBN 0-20-169581-2
  - Excellent book contains examples of pattern uses in a concurrent programming context.
  - Not the easiest of books to read.

References

- Threshold Computers Systems - Contact Web Site

  www.thresholdobjects.com
- Threshold Pattern Tools

  www.qwan.com
- Books
  - *The Timeless Way of Building,* Christopher Alexander, OUP, ISBN 0-19-502402-8
  - *A Pattern Language,* Christopher Alexander, OUP, ISBN 0-19-501919-9
  - *The Patterns Handbook*, Linda Rising, SIGS, ISBN 0-52-164818-1
  - *Design Patterns*, Gamma, Helm, Johnson, Vlissides, AW, ISBN 0-20-163361-2
  - *Pattern Hatching*, Vlissides, AW, ISBN 0-20-143293-5