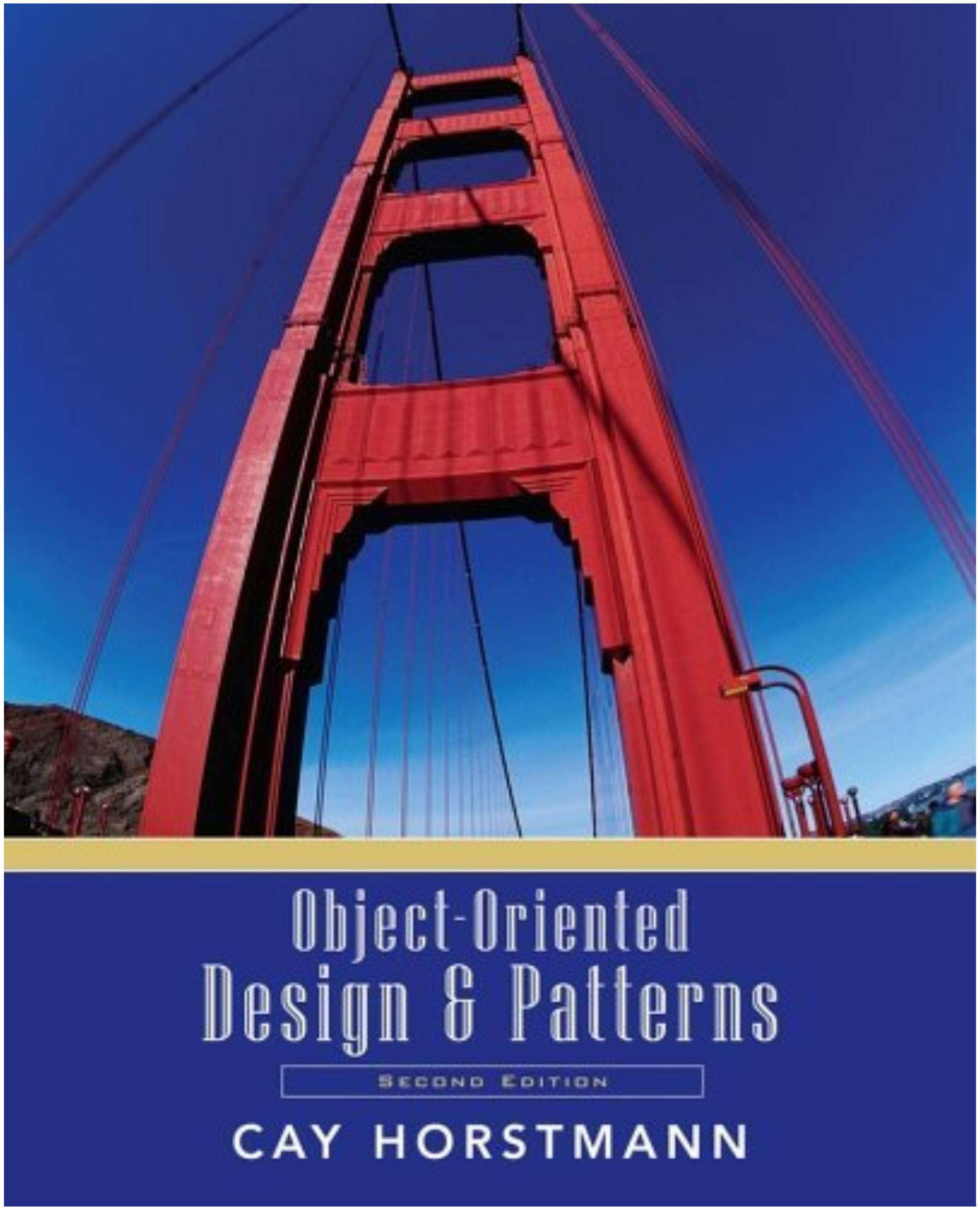
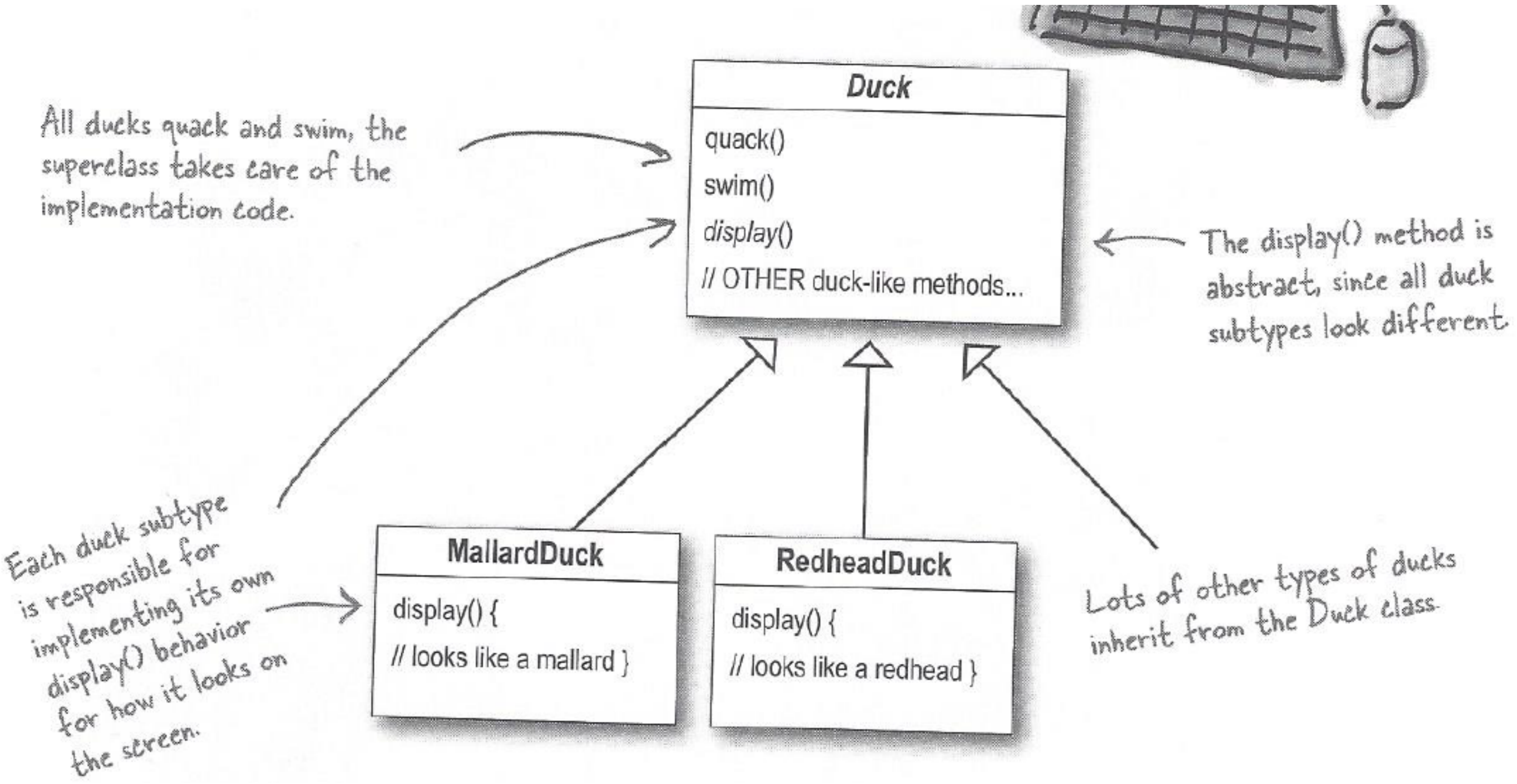
Object-Oriented Design & Patterns

Chapter 10
More Patterns



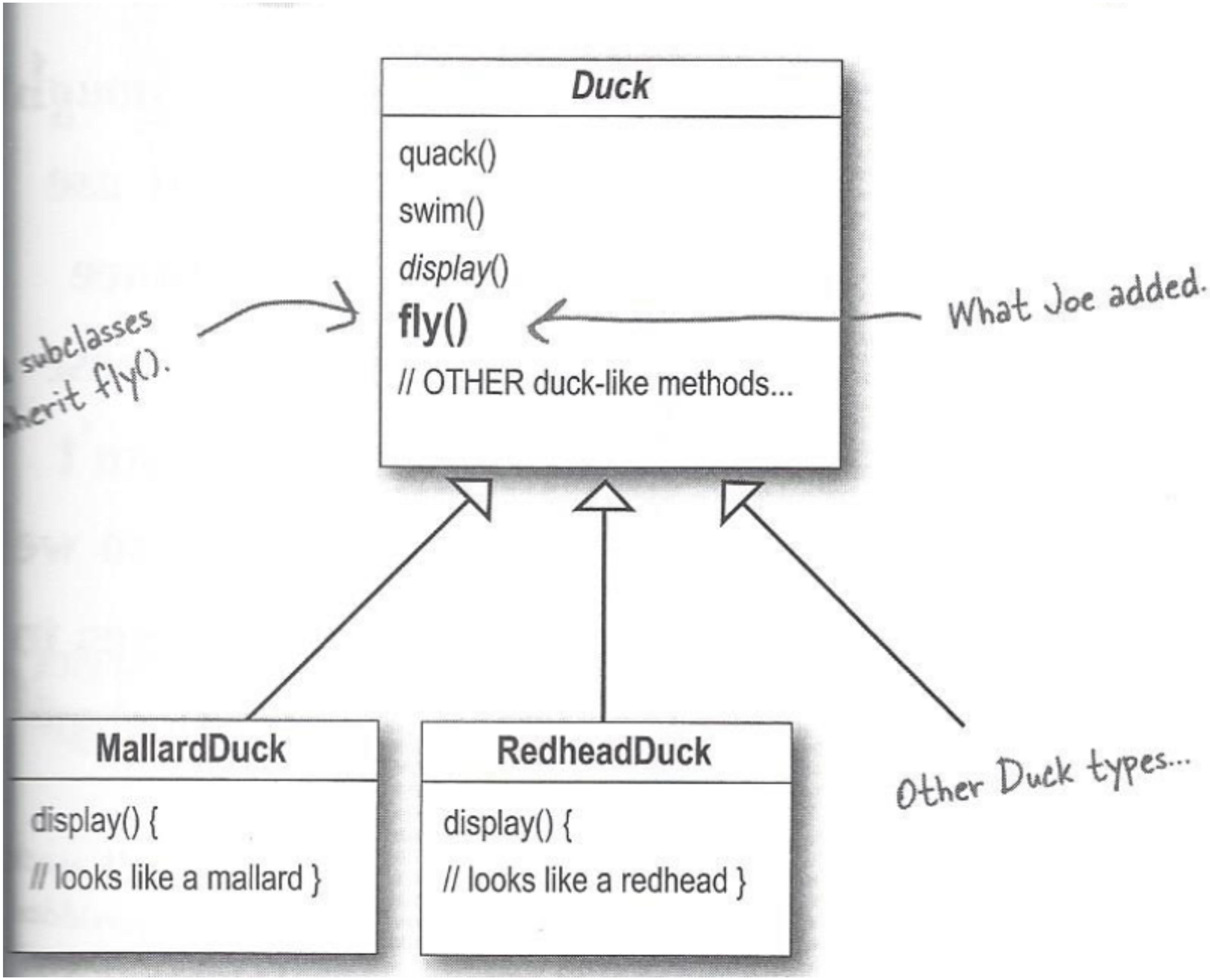

Duck
quack()
swim()
display()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }

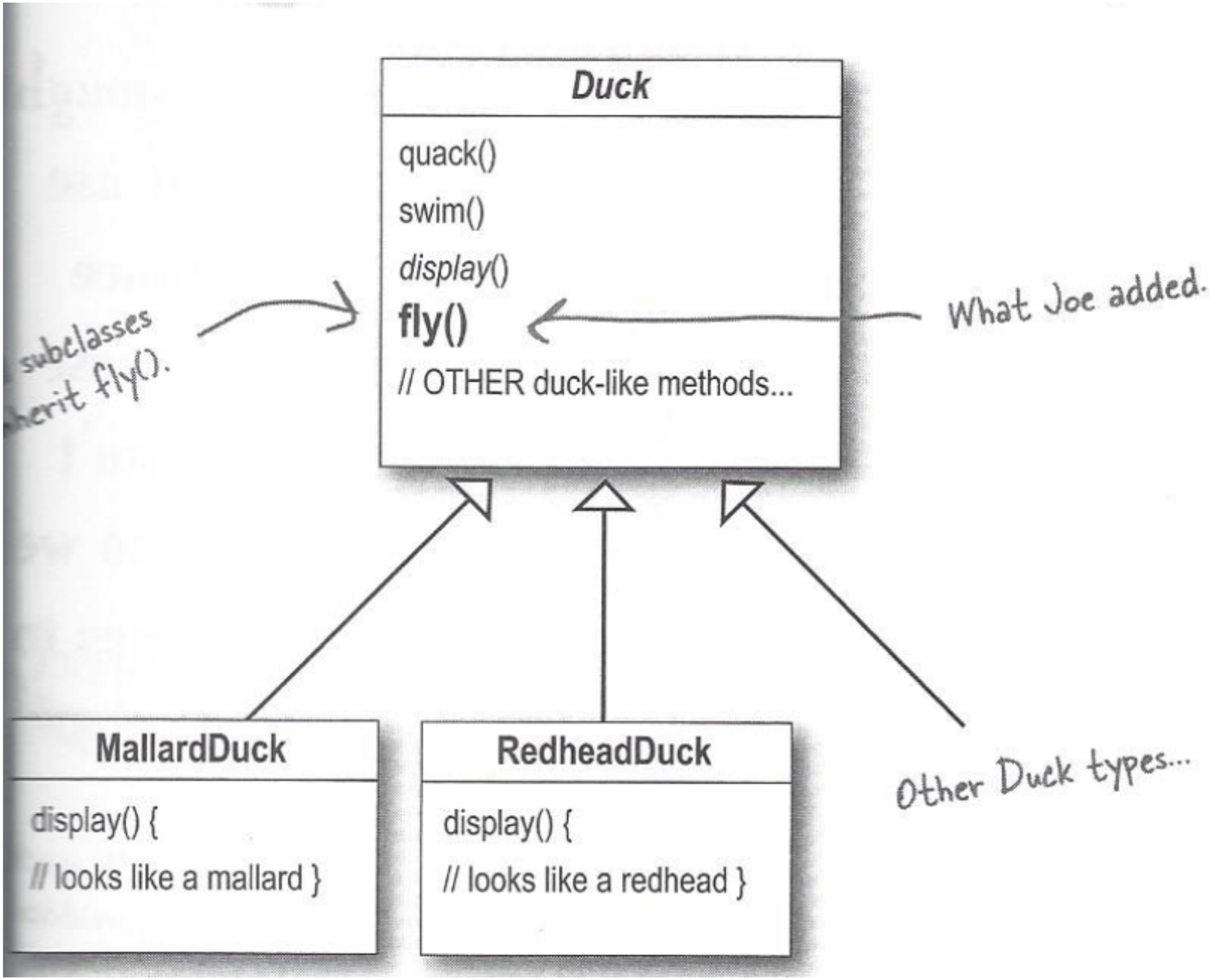
All ducks quack and swim, the superclass takes care of the implementation code.

The display() method is abstract, since all duck subtypes look different.

Each duck subtype is responsible for implementing its own display() behavior for how it looks on the screen.

Lots of other types of ducks inherit from the Duck class.


Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
subclasses
herit fly().
What Joe added.
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
Other Duck types...


Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
What Joe added.
subclasses
herit fly().
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
Other Duck types...

But something went horribly wrong...

Joe, I'm at the shareholder's meeting. They just gave a demo and there were **rubber duckies** flying around the screen. Was this your idea of a joke? You might want to spend some time on Monster.com...

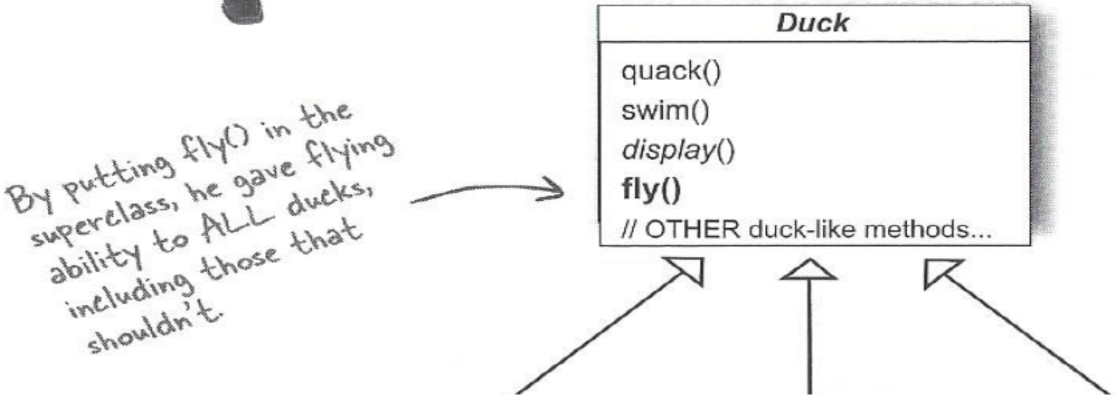
What happened?

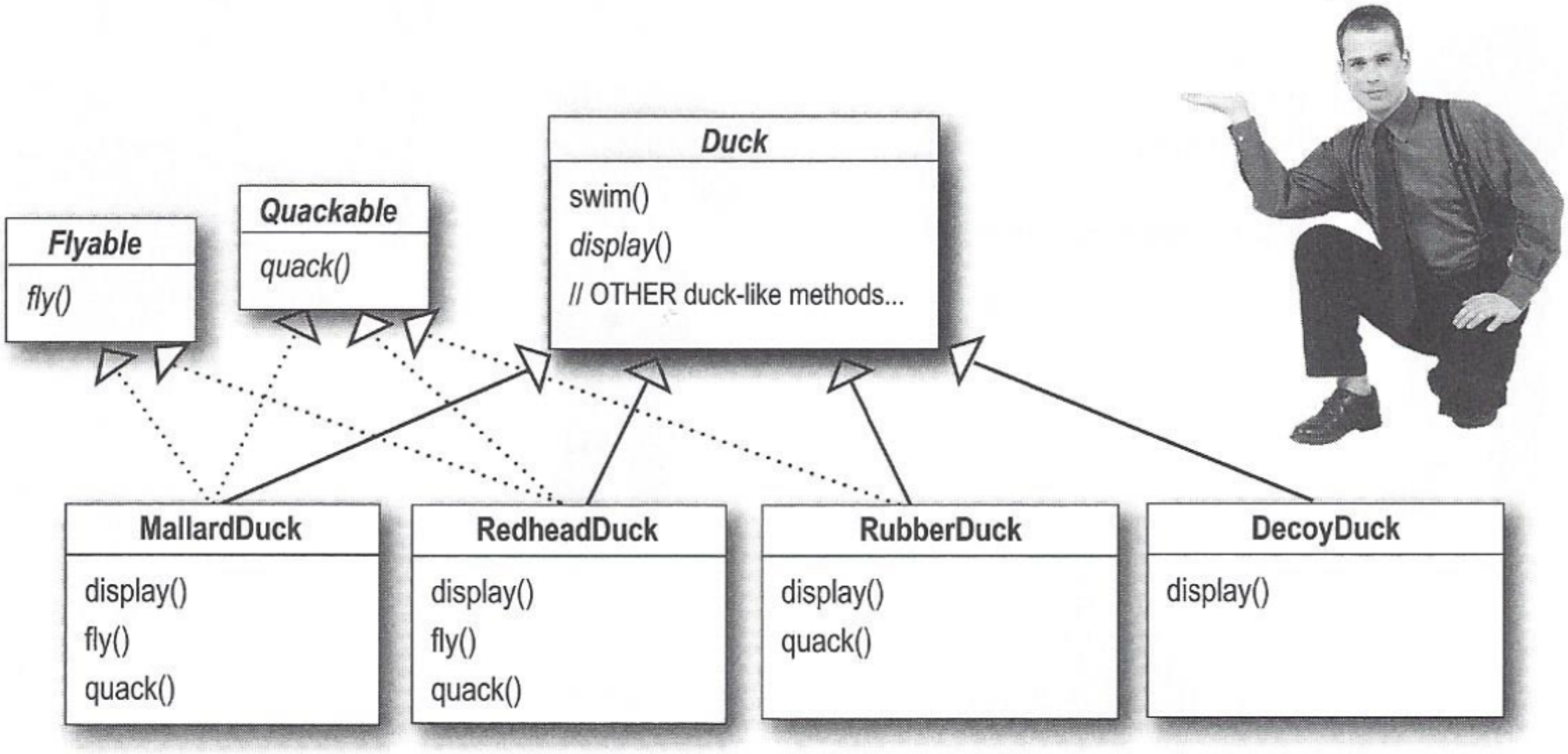
Joe failed to notice that not *all* subclasses of Duck should *fly*. When Joe added new behavior to the Duck superclass, he was also adding behavior that was *not* appropriate for some Duck subclasses. He now has flying inanimate objects in the SimUDuck program.

A localized update to the code caused a non-local side effect (flying rubber ducks)!

OK, so there's a slight flaw in my design. I don't see why they can't just call it a "feature". It's kind of cute...

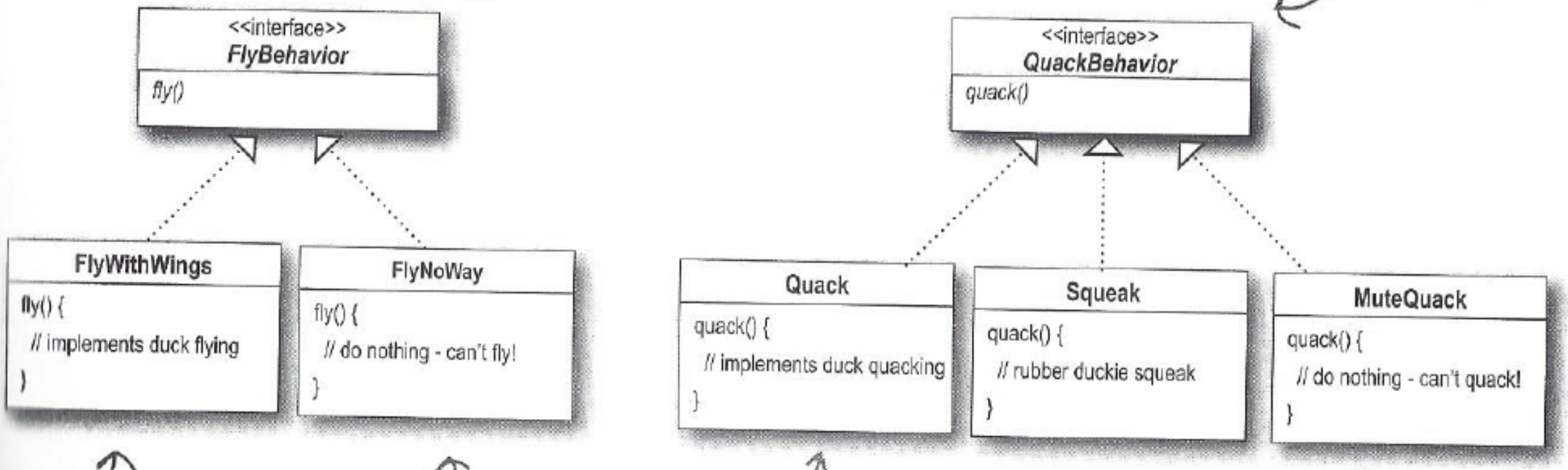
What he tho
was a great u
of inheritanc
for the purp
of __reuse__ hasn
turned out so
when it come
__maintenance.__


Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
By putting fly() in the superclass, he gave flying ability to ALL ducks, including those that shouldn't.


Duck
swim()
display()
// OTHER duck-like methods...
Flyable
fly()
Quackable
quack()
MallardDuck
display()
fly()
quack()
RedheadDuck
display()
fly()
quack()
RubberDuck
display()
quack()
DecoyDuck
display()

FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly method.

Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.



Here's the implementation of flying for all ducks that have wings.

And here's the implementation for all ducks that can't fly.

Quacks that really quack.

Quacks that squeak.

Quacks that make no sound at all.

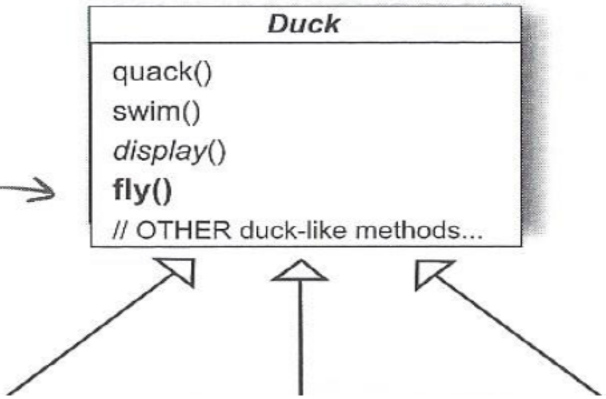
But something went horribly wrong...

Joe, I'm at the shareholder's meeting. They just gave a demo and there were **rubber duckies** flying around the screen. Was this your idea of a joke? You might want to spend some time on Monster.com...

What happened?

Joe failed to notice that not *all* subclasses of Duck should *fly*. When Joe added new behavior to the Duck superclass, he was also adding behavior that was *not* appropriate for some Duck subclasses. He now has flying inanimate objects in the SimUDuck program.

A localized update to the code caused a non-local side effect (flying rubber ducks)!



By putting fly() in the superclass, he gave flying ability to ALL ducks, including those that shouldn't.

OK, so there's a slight flaw in my design. I don't see why they can't just call it a "feature". It's kind of cute...

What he tho
was a great u
of inheritanc
for the purp
of reuse hasn
turned out so
when it come
maintenance.

*Strategies

- Design to interfaces
- Favor composition over inheritance
- Find what varies and encapsulate it

*Program to an Interface

- “Don’t declare variables to be instances of particular concrete classes. Instead, commit only to an interface.” – GOF
- “If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types. The only time you really need to refer to an object’s class is when you’re creating it with a constructor.” - Bloch

*Example

```
// Good – uses interface as type
List<Subscriber> subscribers
                    = new Vector<Subscriber>();
// Bad – uses class as type!
Vector<Subscriber> subscribers
                    = new Vector<Subscriber>();
```

Occasionally you may depend on some functionality not given in the interface.

*Using Classes

- It is entirely appropriate to refer to an object by a class rather than an interface if no appropriate interface exists.
  - String, BigInteger
  - Some frameworks use classes
  - If class provides extra methods not found in interface

Favor Composition Over Inheritance

- We mean implementation inheritance, not interface inheritance.
- **Inheritance violates encapsulation.**
- Example: instrumenting HashSet
  - add
  - addAll

Chapter Topics

- The ADAPTER Pattern
- Actions and the COMMAND Pattern
- The FACTORY METHOD Pattern
- The PROXY Pattern
- The SINGLETON Pattern
- The VISITOR Pattern
- Other Design Patterns

Adapters

- Cable adapter: adapts plug to foreign wall outlet
- OO Programming; Want to adapt class to foreign interface type
- Example: Add CarIcon to container
- Problem: Containers take components, not icons
- Solution: Create an adapter that adapts Icon to Component
- IconAdapter.java
- Ch10/adapter/IconAdapterTester.java

The ADAPTER Pattern

Context

- You want to use an existing class (adaptee) without modifying it.
- The context in which you want to use the class requires target interface that is different from that of the adaptee.
- The target interface and the adaptee interface are conceptually related.

The ADAPTER Pattern

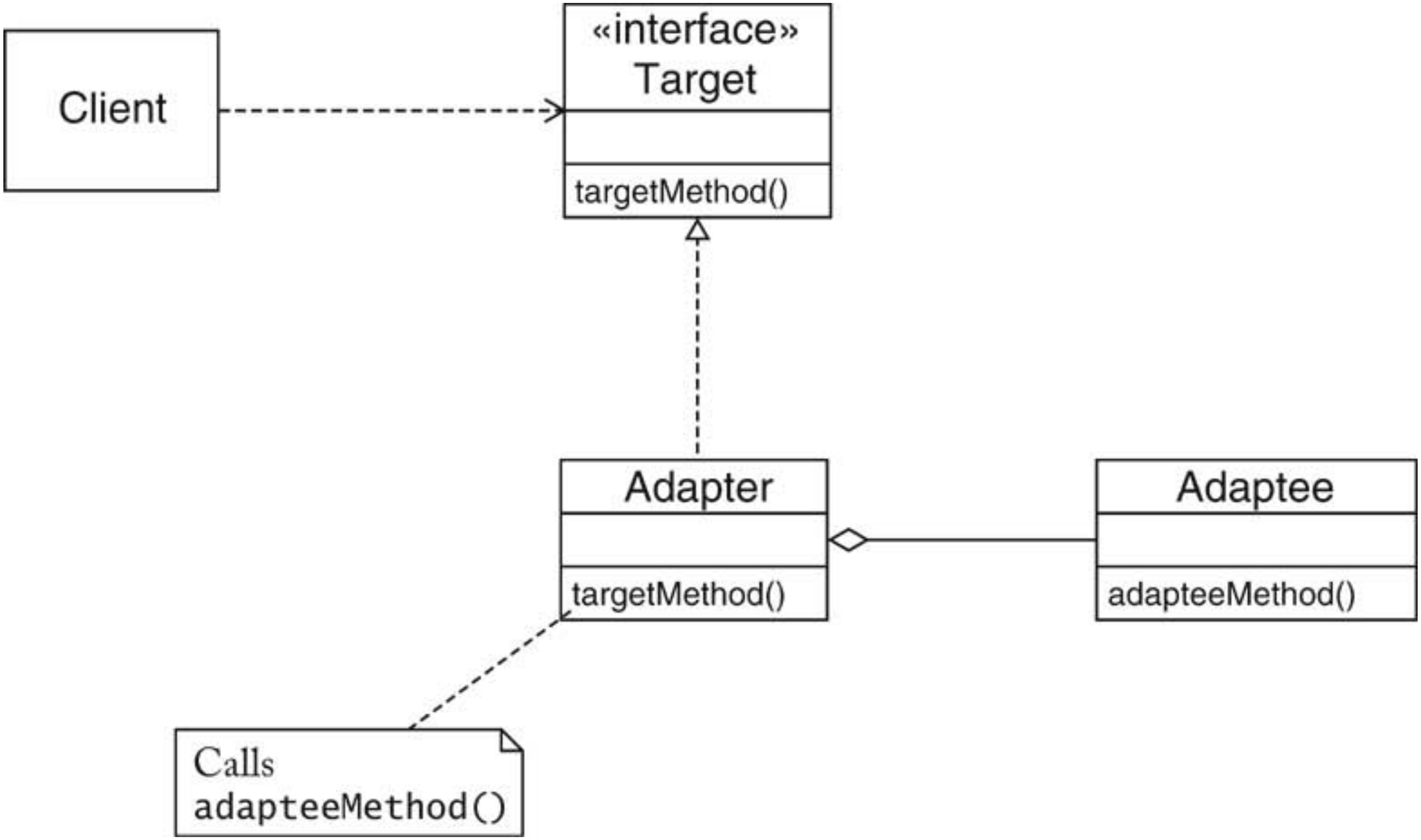
Solution

- Define an adapter class that implements the target interface.

- The adapter class holds a reference to the adaptee. It translates target methods to adaptee methods.

- The client wraps the adaptee into an adapter class object.

The ADAPTER Pattern



The ADAPTER Pattern

Name in Design Pattern	Actual Name (Icon->Component)
Adaptee	Icon
Target	JComponent
Adapter	IconAdapter
Client	The class that wants to add icons into a container
targetMethod()	paintComponent(), getPreferredSize()
adapteeMethod()	paintIcon(), getIconWidth(), getIconHeight()

The ADAPTER Pattern

- In stream library
- Input streams read bytes
- Readers read characters
- Non-ASCII encoding: multiple bytes per char
- System.in is a stream
- What if you want to read characters?
- Adapt stream to reader
- InputStreamReader

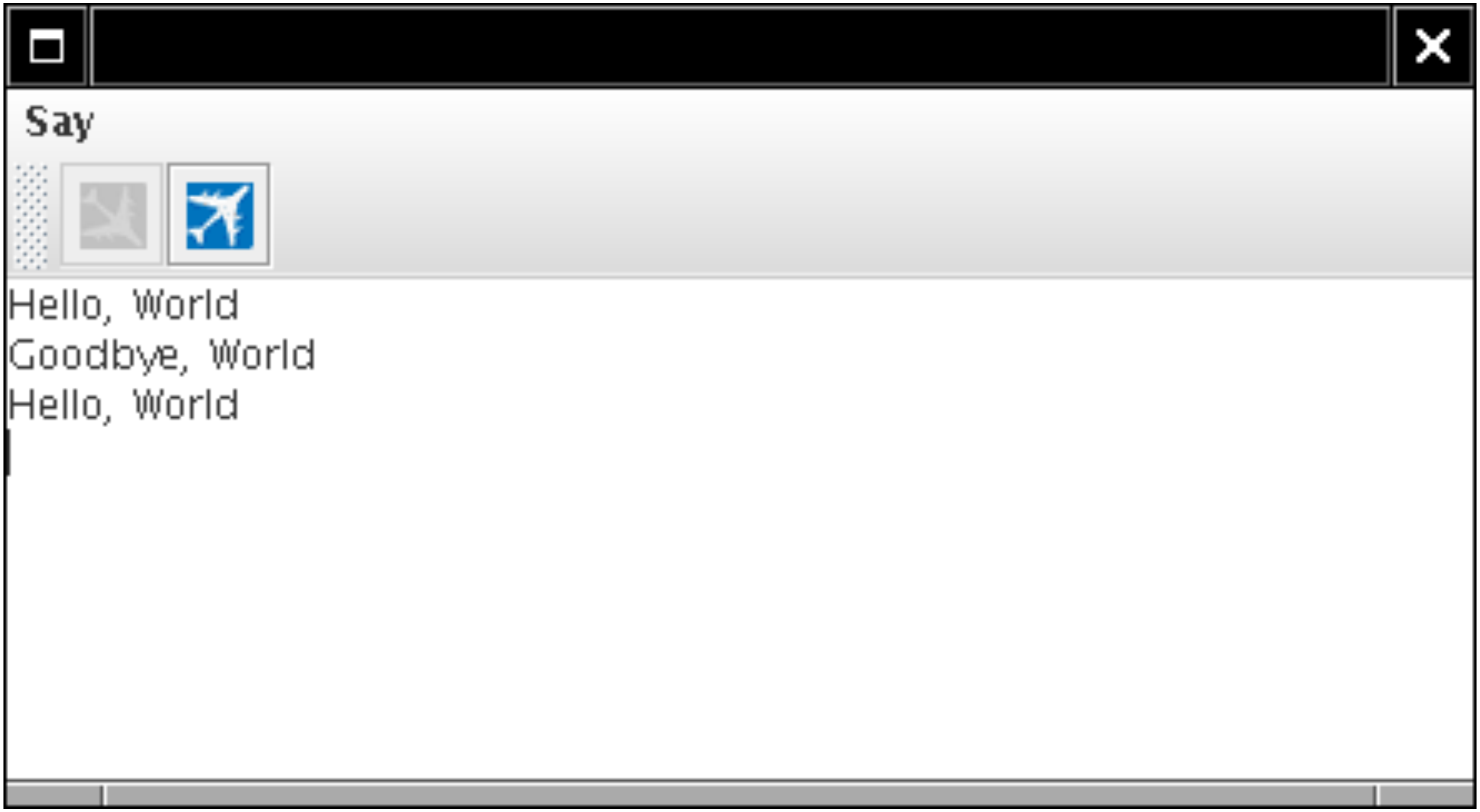
The ADAPTER Pattern

Name in Design Pattern	Actual Name (Stream->Reader)
Adaptee	InputStream
Target	Reader
Adapter	InputStreamReader
Client	The class that wants to read text from an input stream
targetMethod()	read (reading a character)
adapteeMethod()	read (reading a byte)

User Interface Actions

- Multiple routes to the same action
- Example: Cut a block of text
  - Select Edit->Cut from menu
  - Click toolbar button
  - Hit Ctrl+X
- Action can be disabled (if nothing is selected)
- Action has *state*
- Action should be an *object*

User Interface Actions

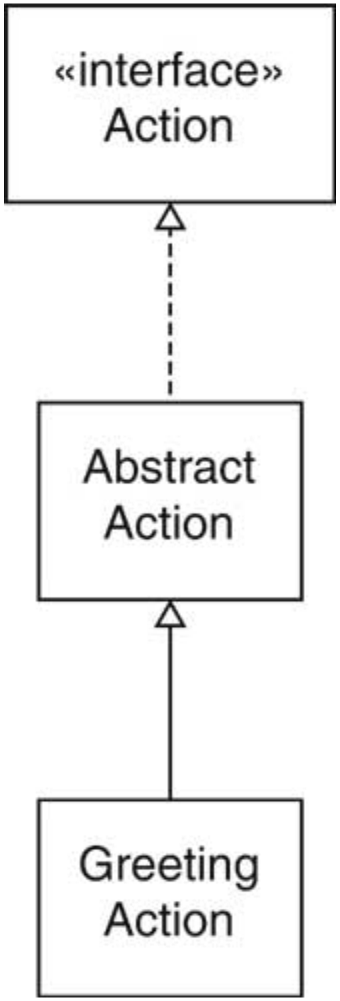


Hello, World
Goodbye, World
Hello, World

The Action Interface Type

- Extends ActionListener
- Can be enabled/disabled
- Additional state, including
  - Action name
  - Icon
- helloAction.putValue(Action.NAME, "Hello");
- menu.add(helloAction);
- Extend AbstractAction convenience class

The Action Interface Type



Action Example

- CommandTester.java
- GreetingAction.java

The COMMAND Pattern

Context

- You want to implement commands that behave like objects
  - because you need to store additional information with commands
  - because you want to collect commands.

The COMMAND Pattern

Solution

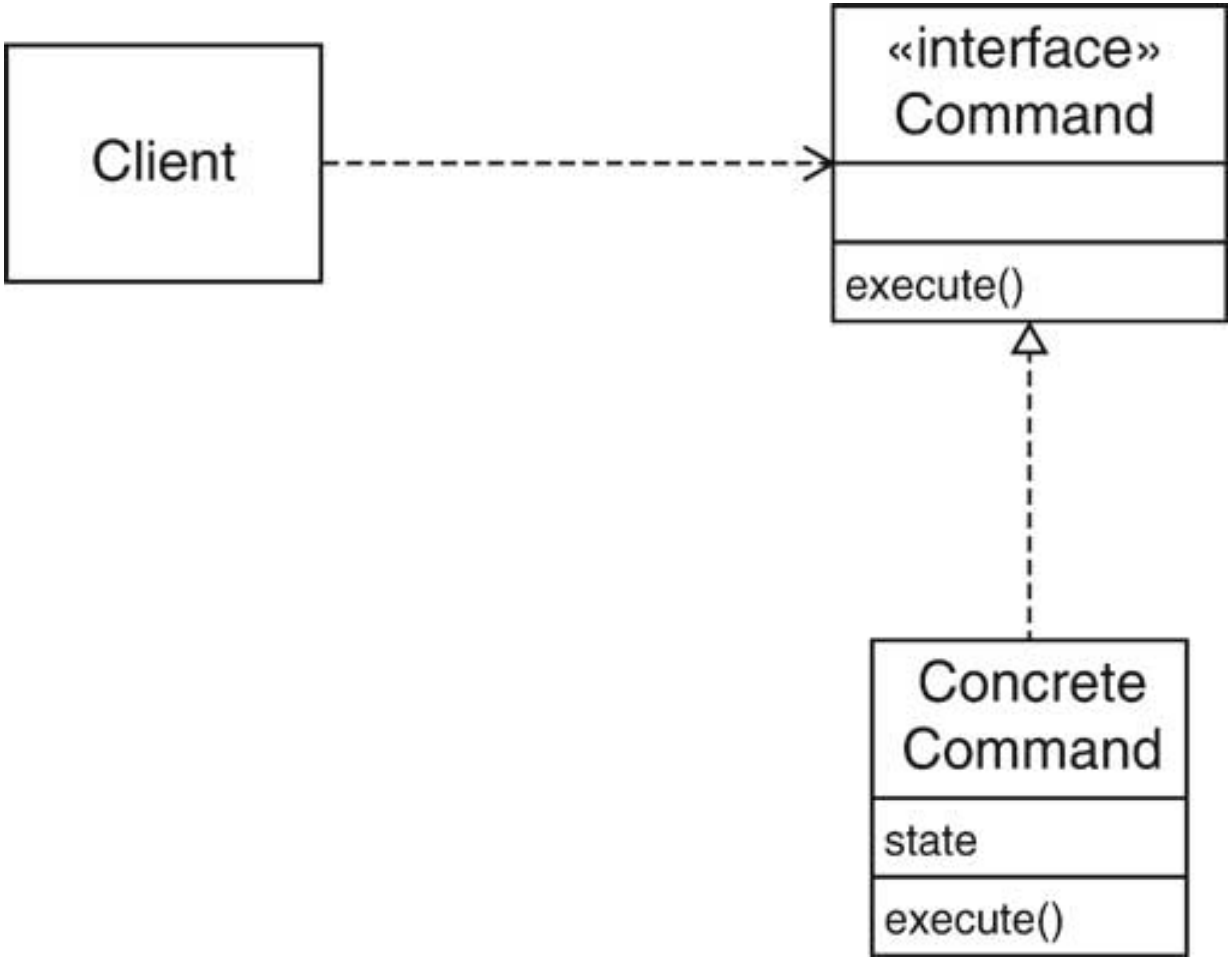
- Define a command interface type with a method to execute the command.

- Supply methods in the command interface type to manipulate the state of command objects.

- Each concrete command class implements the command interface type.

- To invoke the command, call the execute method.

The COMMAND Pattern



The COMMAND Pattern

Name in Design Pattern	**Actual Name (Swing actions)**
`Command`	Action
`ConcreteCommand`	subclass of AbstractAction
`execute()`	actionPerformed()
`state`	name and icon

Factory Methods

- Every collection can produce an iterator
  Iterator iter = list.iterator()
- Why not use constructors?
  Iterator iter = new LinkedListIterator(list);
- Drawback: not generic
  Collection coll = ...;
  Iterator iter = new ???(coll);
- Factory method works for all collections
  Iterator iter = coll.iterator();
- Polymorphism!

The FACTORY METHOD Pattern

Context

- A type (the creator) creates objects of another type (the product).

- Subclasses of the creator type need to create different kinds of product objects.

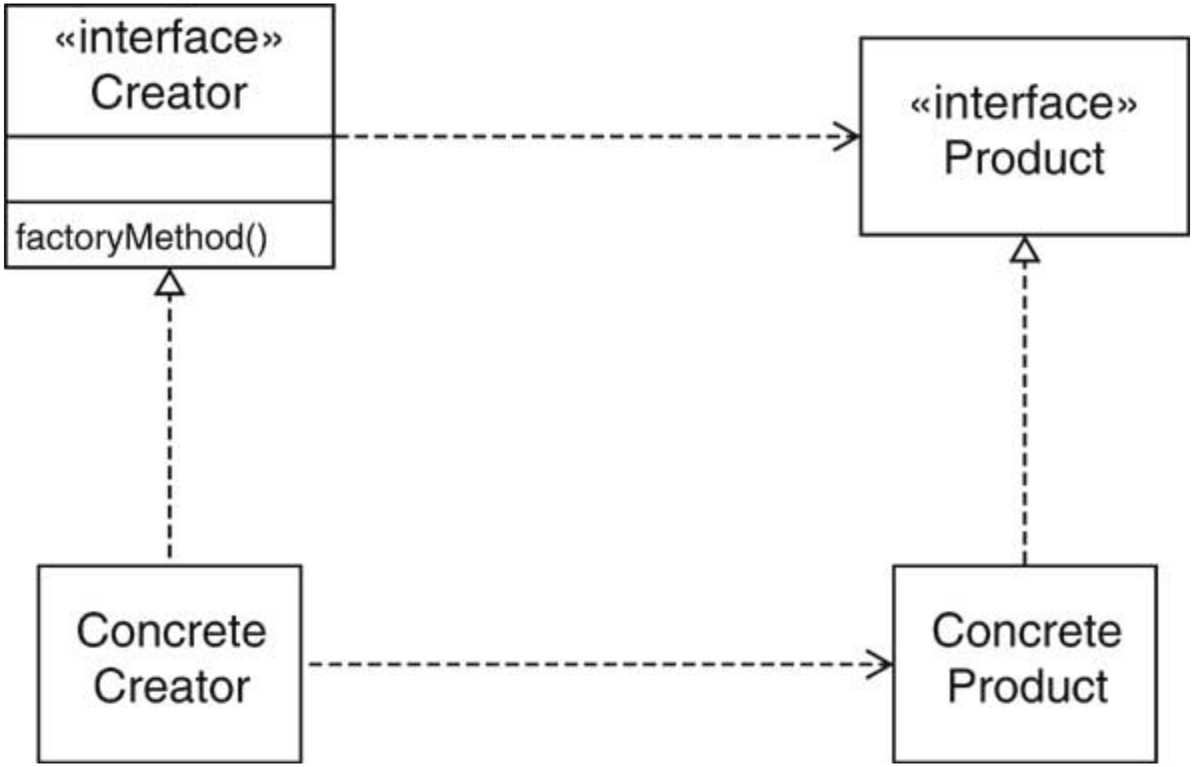
- Clients do not need to know the exact type of product objects.

The FACTORY METHOD Pattern

Solution

- Define a creator type that expresses the commonality of all creators.
- Define a product type that expresses the commonality of all products.
- Define a method, called the factory method, in the creator type.
  The factory method yields a product object.
- Each concrete creator class implements the factory method so that it returns an object of a concrete product class.

The FACTORY METHOD Pattern



The FACTORY METHOD Pattern

Name in Design Pattern	Actual Name (iterator)
`Creator`	`Collection`
`ConcreteCreator`	`A subclass of Collection`
`factoryMethod()`	`iterator()`
`Product`	`Iterator`
`ConcreteProduct`	`A subclass of Iterator (which is often anonymous)`

Not a FACTORY METHOD

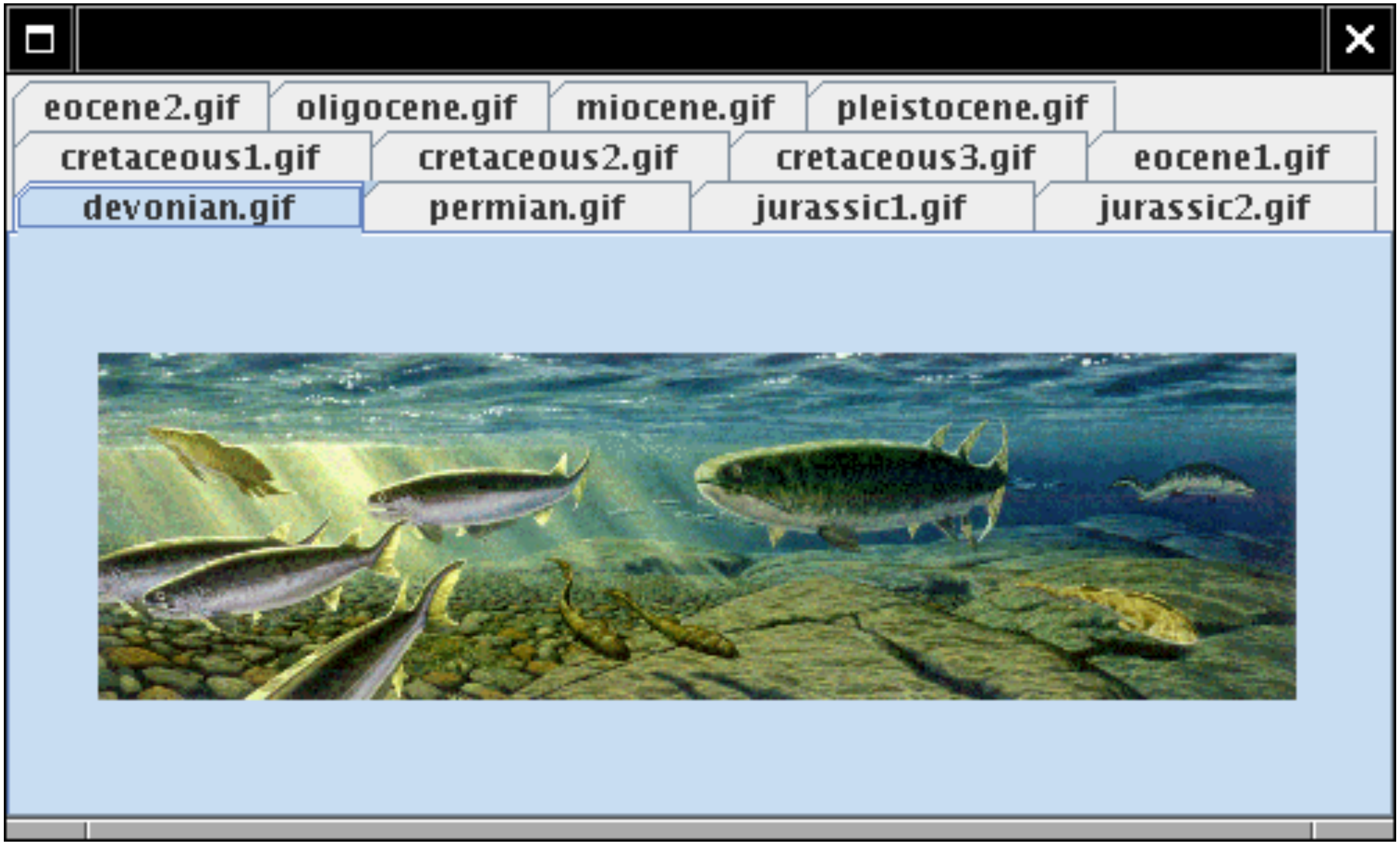
- Not all "factory-like" methods are instances of this pattern
- Create DateFormat instances

  ```
  DateFormat formatter =
  DateFormat.getDateInstance();
  Date now = new Date();
  String formattedDate = formatter.format(now);
  ```
- getDateInstance is a *static* method
- No polymorphic creation

Proxies

- Proxy: a person who is authorized to act on another person s behalf
- Example: Delay instantiation of object
- Expensive to load image
- Not necessary to load image that user doesn't look at
- Proxy defers loading until user clicks on tab

Deferred Image Loading



Deferred Image Loading

- Normally, programmer uses image for label:
  ```
  JLabel label = new JLabel(new ImageIcon(imageName));
  ```
- Use proxy instead:
  ```
  JLabel label = new JLabel(new ImageProxy(imageName));
  ```
- paintIcon loads image if not previously loaded
  ```
  public void paintIcon(Component c, Graphics g, int x, int y)
  {
     if (image == null) image = new ImageIcon(name);
     image.paintIcon(c, g, x, y);
  }
  ```

Proxies

- ImageProxy.java
- ProxyTester.java
- "Every problem in computer science can be solved by an additional level of indirection"
- Another use for proxies: remote method invocation

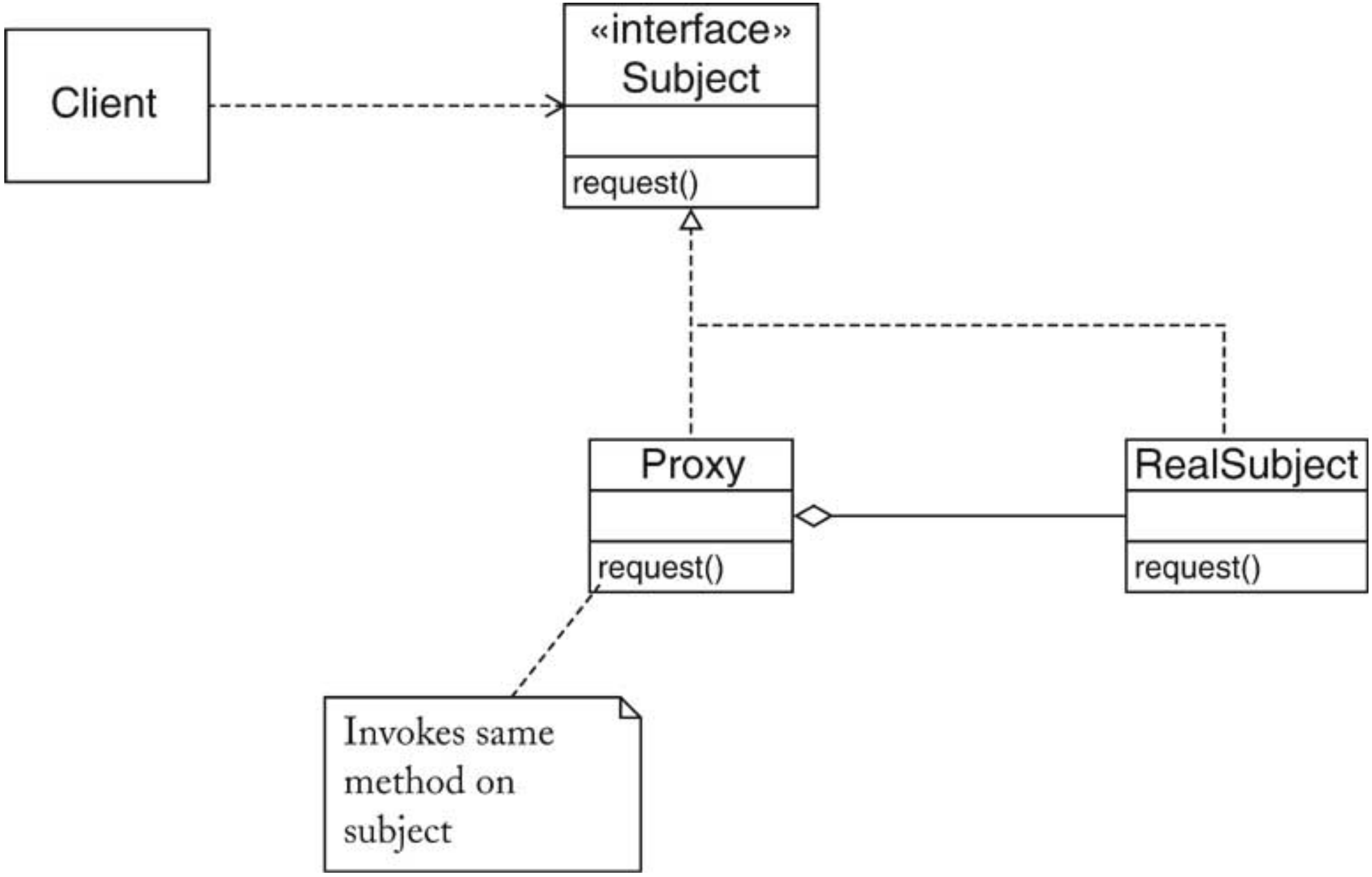
The PROXY Pattern

- **Solution**
- Define a proxy class that implements the subject interface type.
  The proxy holds a reference to the real subject, or otherwise knows how to locate it.

- The client uses a proxy object.

- Each proxy method invokes the same method on the real subject and provides the necessary modifications.

The PROXY Pattern



The PROXY Pattern

Name in Design Pattern	Actual Name (image proxy)
Subject	Icon
RealSubject	ImageIcon
Proxy	ImageProxy
request()	The methods of the Icon interface type
Client	JLabel

Singletons

- "Random" number generator generates predictable stream of numbers
- Example: seed = (seed * 25214903917 + 11) % 248
- Convenient for debugging: can reproduce number sequence
- Only if all clients use *the same random number generator*
- Singleton class = class with one instance

Random Number Generator Singleton

```
public class SingleRandom
  {
    private SingleRandom() { generator = new
                                 Random(); }
    public void setSeed(int seed) {
                      generator.setSeed(seed); }
    public int nextInt() { return generator.nextInt(); }
    public static SingleRandom getInstance() {
                            return instance; }
    private Random generator;
    private static SingleRandom instance = new
                            SingleRandom();
  }
```

The SINGLETON Pattern

Context

- All clients need to access a single shared instance of a class.

- You want to ensure that no additional instances can be created accidentally.

The SINGLETON Pattern

Solution

- Define a class with a private constructor.
- The class constructs a single instance of itself.
- Supply a static method that returns a reference to the single instance.

Not a SINGLETON

- Toolkit used for determining screen size, other window system parameters

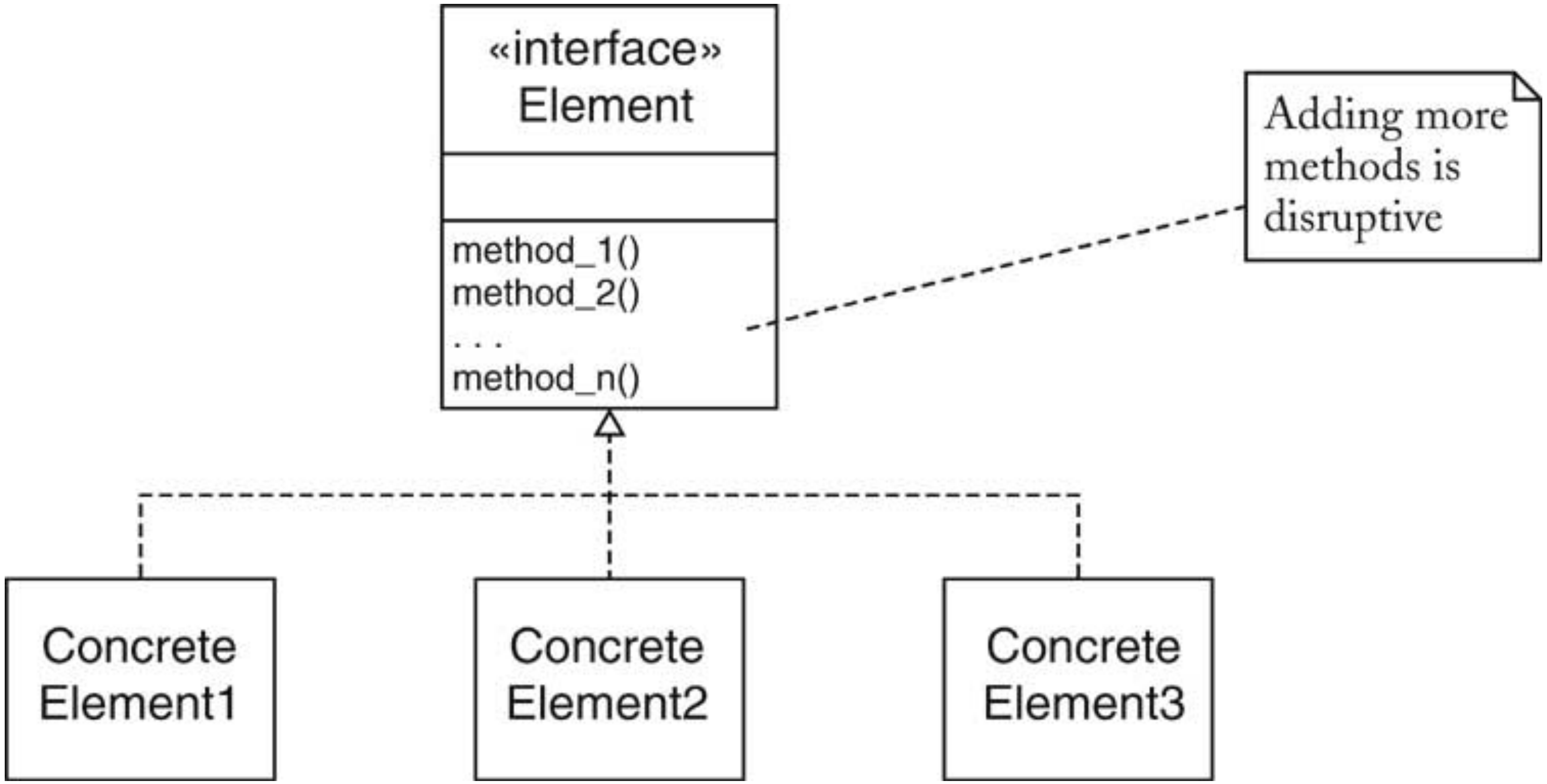
- Toolkit class returns default toolkit
  Toolkit kit = Toolkit.getDefaultToolkit();

- Not a singleton--can get other instances of Toolkit
- Math class not example of singleton pattern
- *No* objects of class Math are created

Inflexible Hierarchies

- How can one add operations to compound hierarchies?
- Example: AWT Component, Container, etc. form hierarchy
- Lots of operations: getPreferredSize,repaint
- Can't add new methods without modifying Component class
- VISITOR pattern solves this problem
- Each class must support one method
  void accept(Visitor v)

Inflexible Hierarchies



Visitors

- Visitor is an interface type
- Supply a separate class for each new operation
- Most basic form of accept method:
  public void accept(Visitor v) { v.visit(this); }
- Programmer must implement visit

Visitors

- Problem: Operation may be different for different element types
- Can't rely on polymorphism
- Polymorphism assumes fixed set of methods, defined in superclass
- Trick: Can use variable set of methods *if set of classes is fixed*
- Supply separate visitor methods:

  ```
  public interface Visitor
  {
     void visitElementType1(ElementType1 element);
     void visitElementType2(ElementType2 element);
     ...
     void visitElementTypen(ElementTypen element);
  }
  ```

Visitors

- Example: Directory tree
- Two kinds of elements: DirectoryNode,FileNode
- Two methods in visitor interface type:
  void visitDirectoryNode(DirectoryNode node)
  void visitFileNode(FileNode node)

Double Dispatch

- Each element type provides methods:

  ```
  public class ElementTypei
  {
     public void accept(Visitor v)
  { v.visitElementTypei(this); }
     ...
  }
  ```

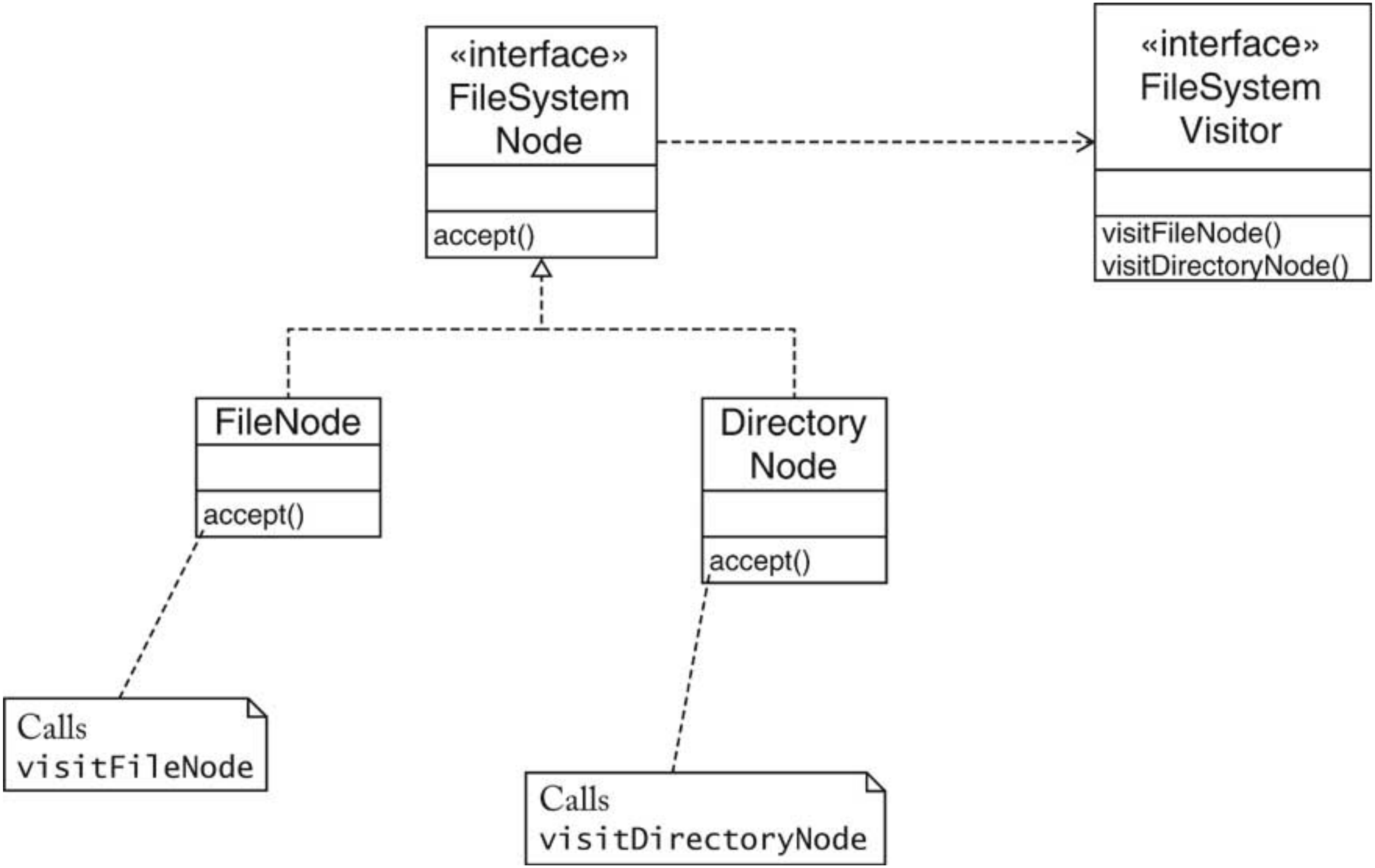
- Completely mechanical
- Example:

  ```
  public class DirectoryNode
  {
     public void accept(Visitor v)
  {   v.visitDirectoryNode(this); }
     ...
  }
  ```

Visitor Example

- Standard File class denotes both files and directories
- Improved design: FileNode,DirectoryNode
- Common interface type: FileSystemNode
- Accepts FileSystemVisitor
- Visitor methods:
  visitFileNode
  visitDirectoryNode

Visitor Example



Visitor Example

- Actual visitor: PrintVisitor
- Prints names of files (in visitFileNode)

- Lists contents of directories (in visitDirectoryNode)

- Maintains indentation level

```
..
  command
    CommandTester.java
    GreetingAction.java
  visitor
    FileNode.java
     DirectoryNode.java
```

Visitor Example

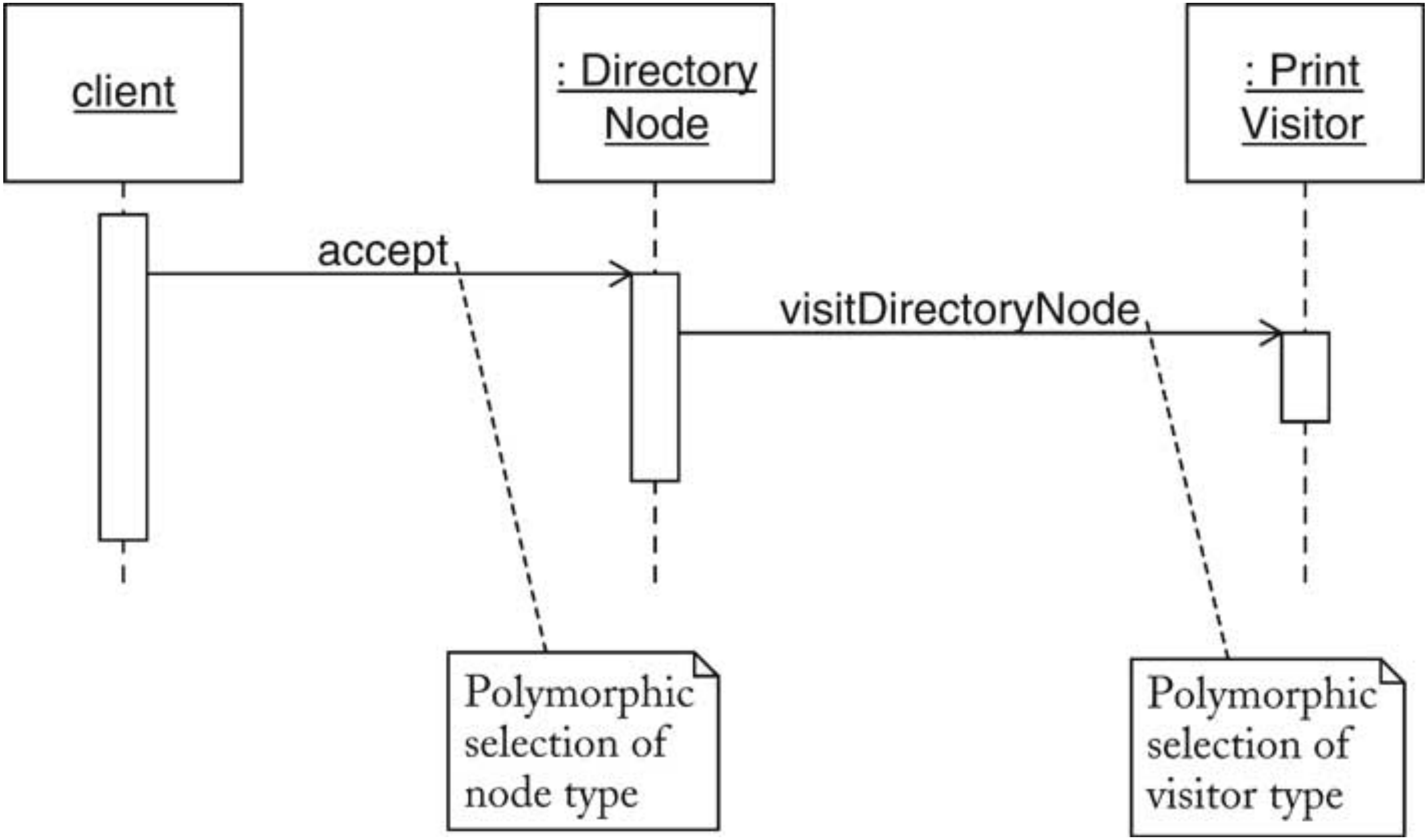
- FileSystemNode.java
- FileNode.java
- DirectoryNode.java
- FileSystemVisitor.java
- PrintVisitor.java
- VisitorTester.java

Double Dispatch Example

- DirectoryNode node = new DirectoryNode(new File("..")); node.accept(new PrintVisitor());

- node is a DirectoryNode
- Polymorphism: node.accept calls DirectoryNode.accept
- That method calls v.visitDirectoryNode
- v is a PrintVisitor
- Polymorphism: calls PrintVisitor.visitDirectoryNode
- Two polymorphic calls determine
  - node type
  - visitor type

Double Dispatch Example


client
: Directory Node
: Print Visitor
accept
visitDirectoryNode
Polymorphic selection of node type
Polymorphic selection of visitor type

The VISITOR Pattern

Context

- An object structure contains element classes of multiple types, and you want to carry out operations that depend on the object types.

- The set of operations should be extensible over time.

- The set of element classes is fixed.

The VISITOR Pattern

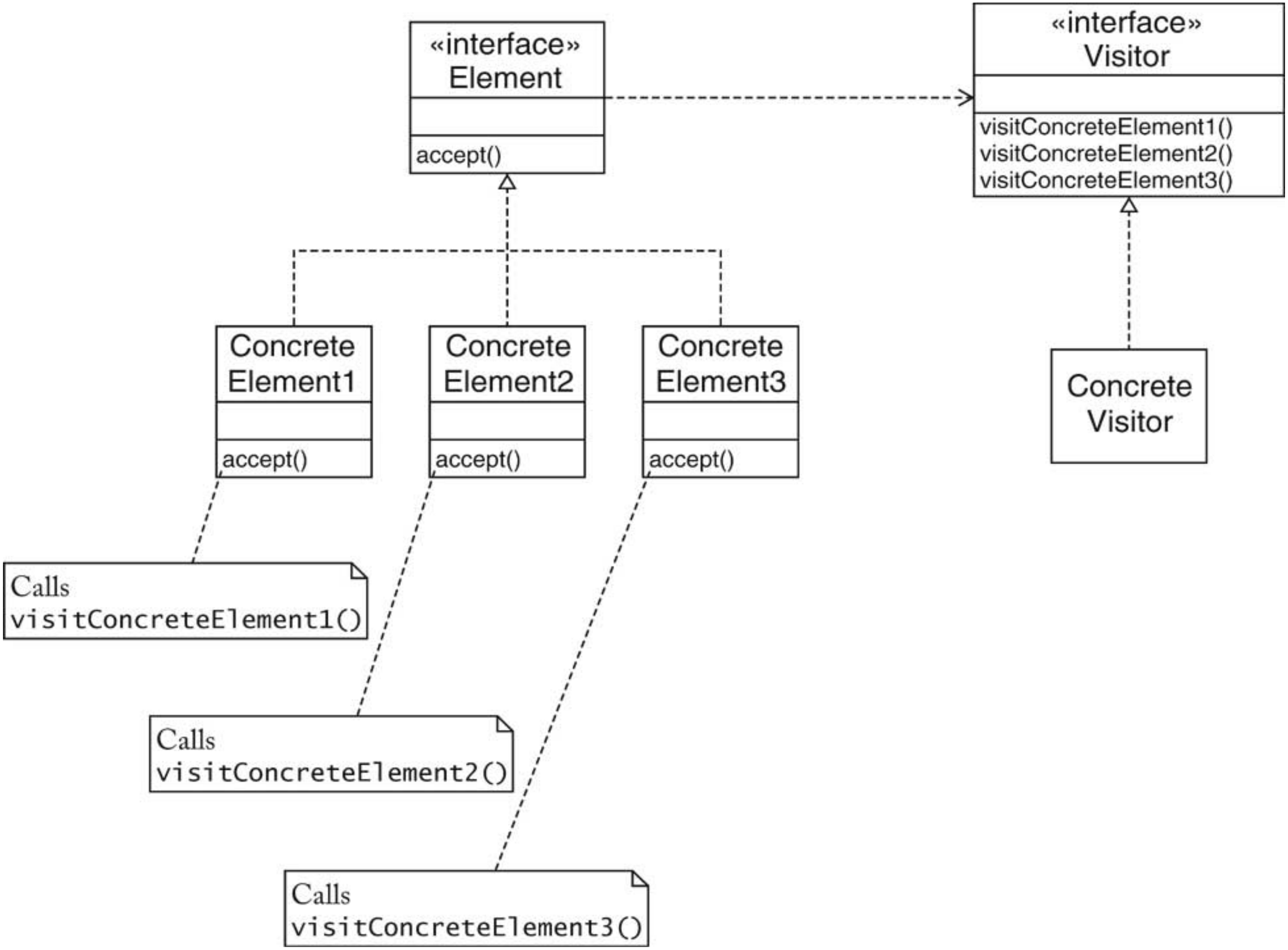
Solution

- Define a visitor interface type that has methods for visiting elements of each of the given types.

- Each element class defines an accept method that invokes the matching element visitation method on the visitor parameter.

- To implement an operation, define a class that implements the visitor interface type and supplies the operation s action for each element type.

The VISITOR Pattern



The VISITOR Pattern

Name in Design Pattern	Actual Name (file system visitor)
`Element`	`FileSystemNode`

`ConcreteElement`	`FileNode,`
	`DirectoryNode`

`Visitor`	`FileSystemVisitor`
`ConcreteVisitor`	`PrintVisitor`

Other Design Patterns

- Abstract Factory
- Bridge
- Builder
- Chain of Responsibility
- Flyweight
- Interpreter
- Mediator
- Memento
- State

Abstract Factory

- An abstract class defines methods that construct related products. Concrete factories create these product sets.
- Example: An abstract class specifies methods for constructing buttons, menus, and so on. Each user interface “look and feel” supplies a concrete subclass.

Abstract Factory

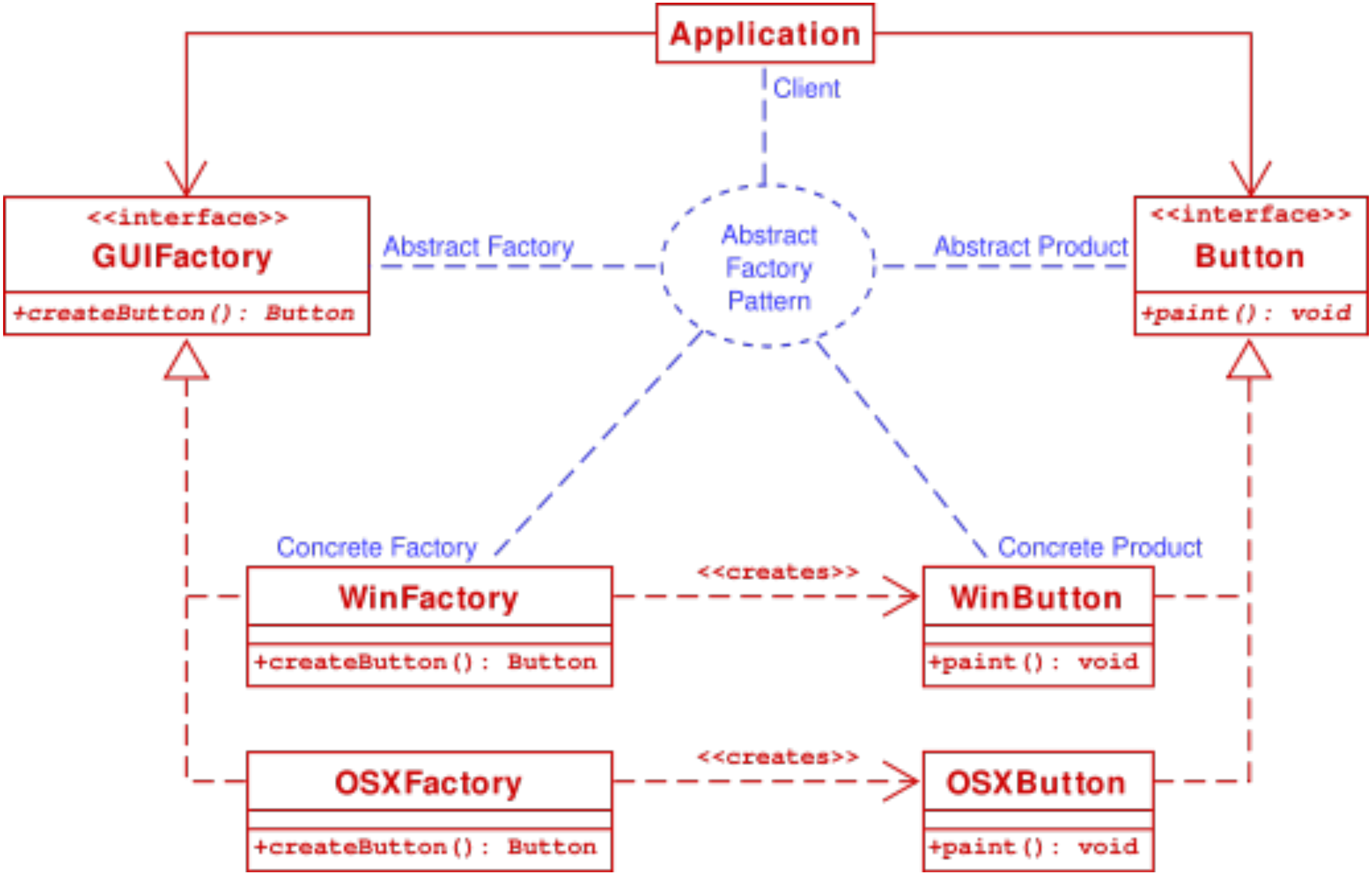
- Name – abstract factory.
- Problem.
  - Families of related objects need to be instantiated.
- Solution.
  - Define an abstract class that specifies which objects are to be made then implement one concrete class for each family. Tables or files can be used to accomplish the same thing.

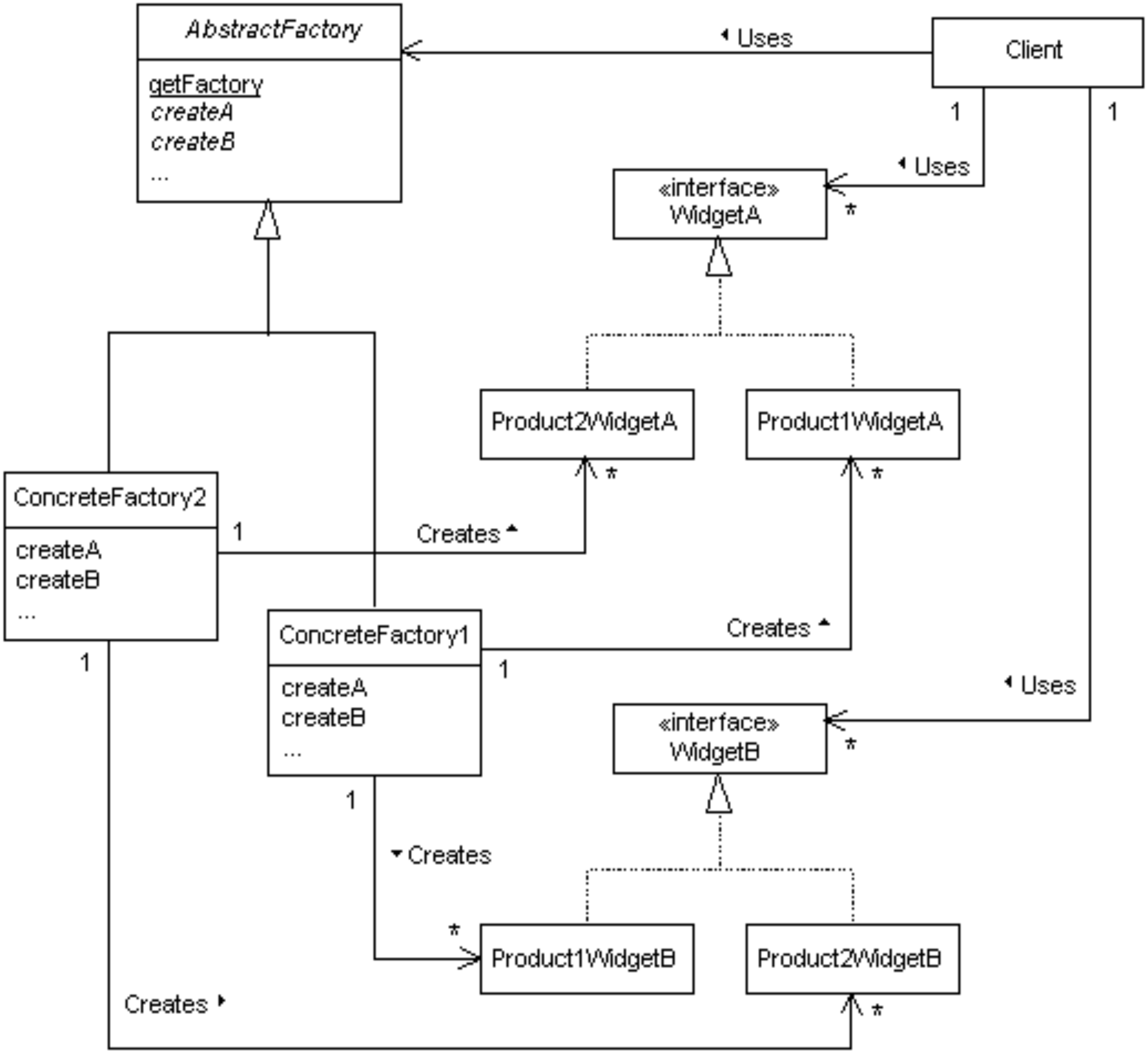
Abstract Factory

- Name: Abstract Factory
- Problem:
  - Families of related objects need to be instantiated.
- Solution:
  - The abstract factory defines the interface for how to create each member of the family. Each family is created by having its own unique concrete factory.

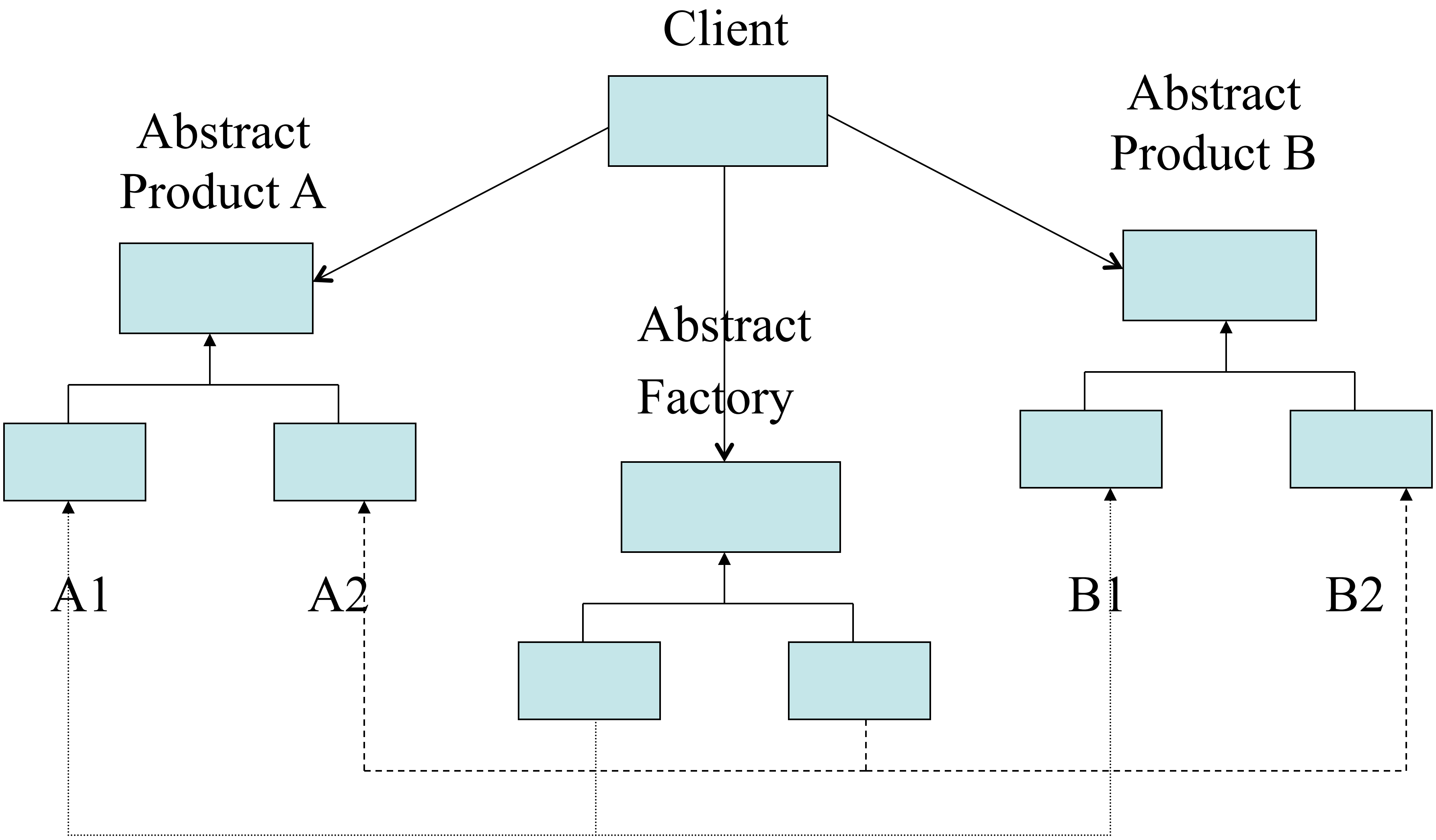
Tradeoffs

- Isolates concrete classes
- Makes exchanging product families easy
- Promotes consistency among products
- Hard to add new kinds of products


Application
Client
<<interface>>
GUIFactory
+createButton() : Button
Abstract Factory
Abstract
Factory
Pattern
Abstract Product
<<interface>>
Button
+paint() : void
Concrete Factory
Concrete Product
WinFactory
+createButton() : Button
<<creates>>
WinButton
+paint() : void
OSXFactory
+createButton() : Button
<<creates>>
OSXButton
+paint() : void


AbstractFactory
getFactory
createA
createB
...
◂ Uses
Client
1
1
◂ Uses
«interface»
WidgetA
*
Product2WidgetA
Product1WidgetA
*
*
ConcreteFactory2
createA
createB
...
1
Creates ▴
ConcreteFactory1
createA
createB
...
Creates ▴
1
1
◂ Uses
«interface»
WidgetB
*
1
▾ Creates
*
Product1WidgetB
Product2WidgetB
Creates ▸
*

Abstract Factory



Client

Abstract Product A

Abstract Product B

Abstract Factory

A1

A2

B1

B2

Summary

- First, identify the rules for instantiation and define an abstract class with an interface that has a method for each object that needs to be instantiated
- Implement concrete classes from this class for each family
- The client object uses this factory object to create the server objects that it needs

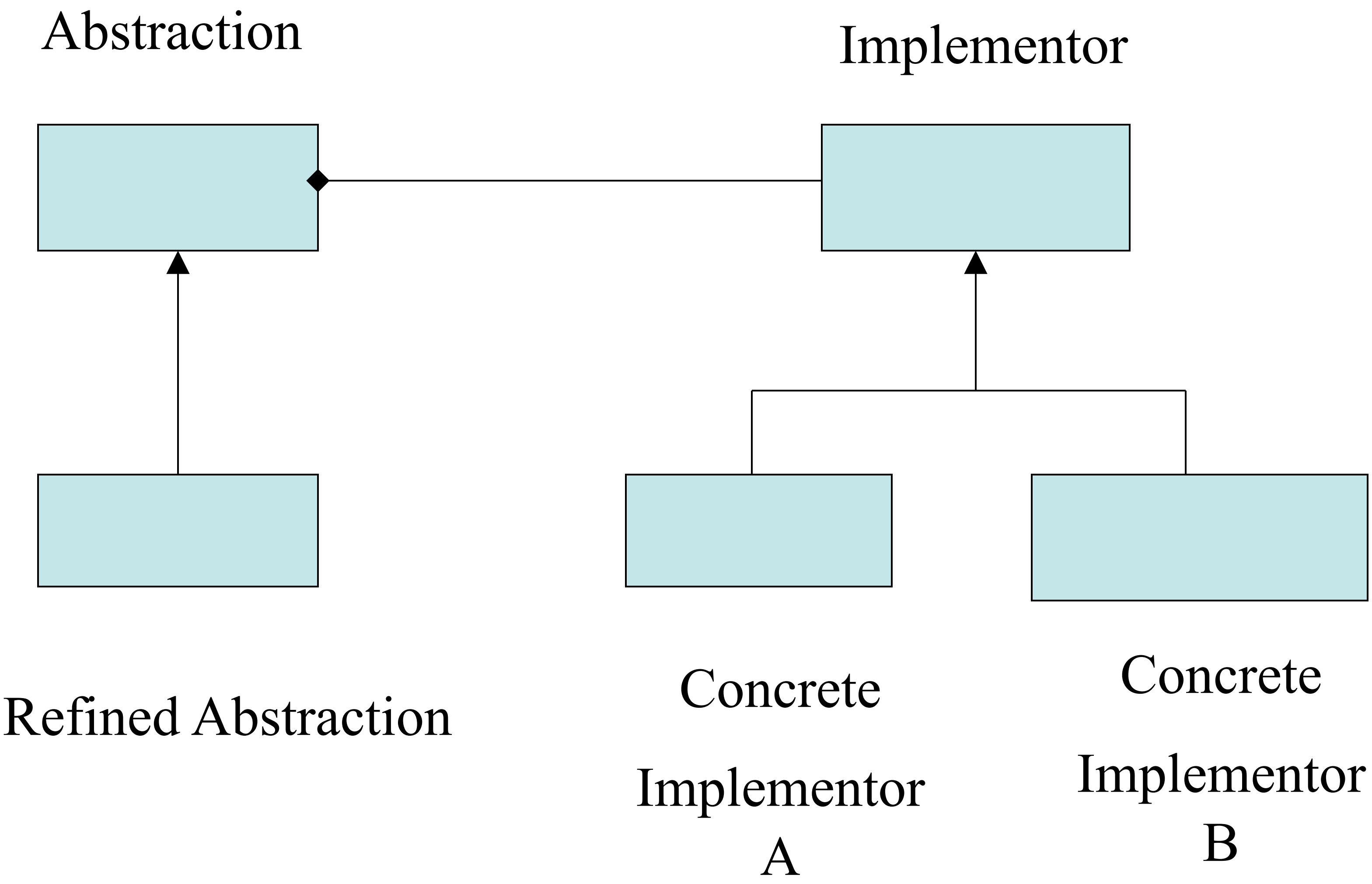
Bridge

- An abstraction and its implementation have separate inheritance hierarchies.
- Example: A hierarchy of window types has separate implementations in various operating systems.

Bridge Pattern

- Name - bridge
- Problem
  - the derivations of an abstract class must use multiple implementations without causing an explosion in the number of classes
- Solution
  - Define an interface for all implementations to use and have the derivations of the abstract class use that.

Bridge Pattern



Abstraction

Implementor

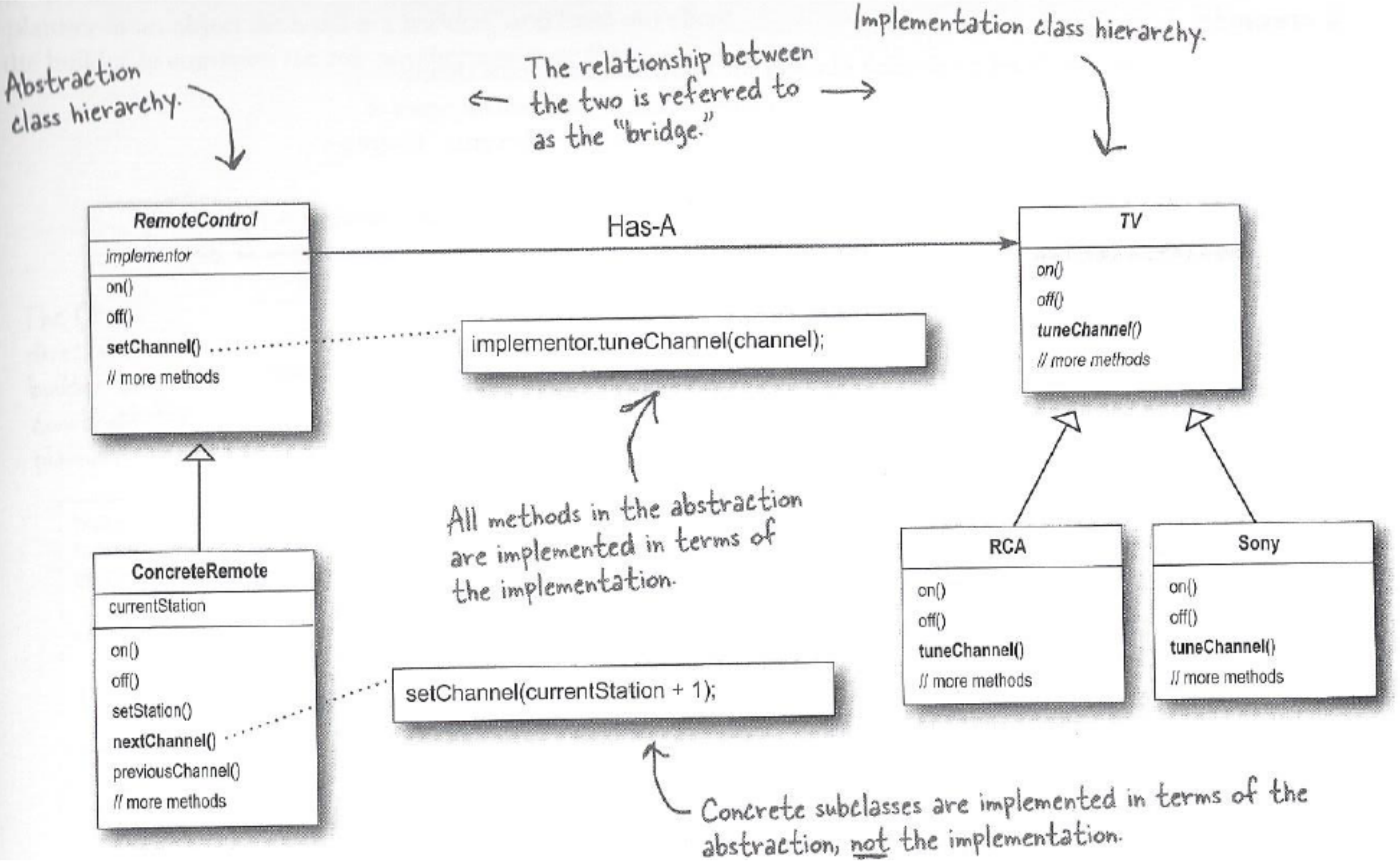
Refined Abstraction

Concrete

Implementor
A

Concrete

Implementor
B



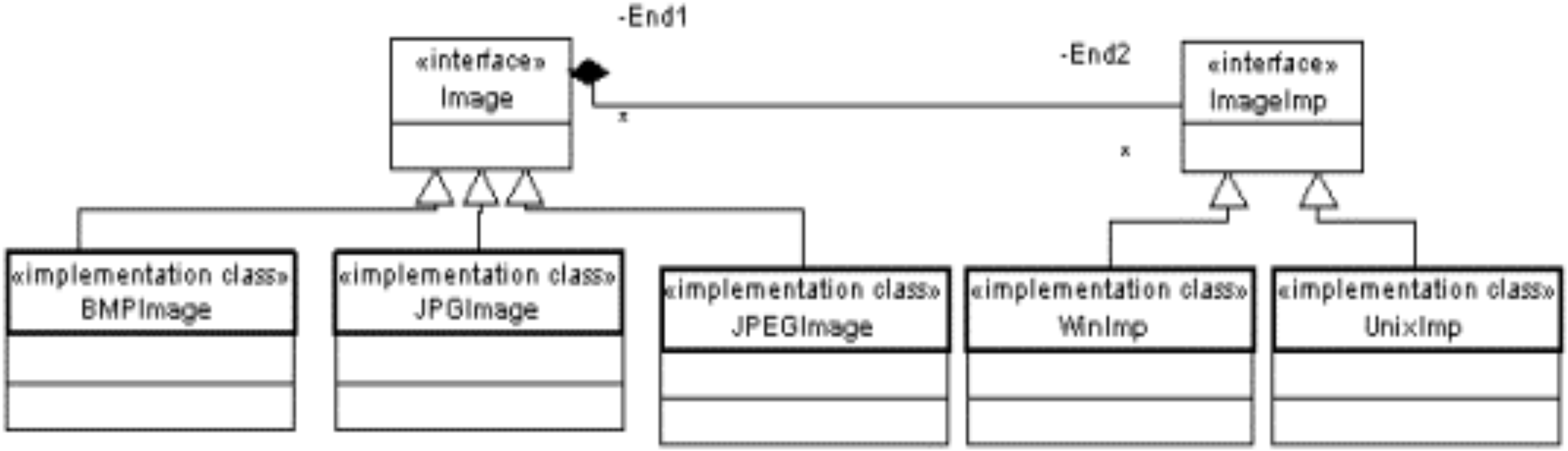
Abstraction class hierarchy.

The relationship between the two is referred to as the "bridge."

Implementation class hierarchy.

All methods in the abstraction are implemented in terms of the implementation.

Concrete subclasses are implemented in terms of the abstraction, not the implementation.


-End1
-End2
«interface»
Image
«interface»
ImageImp
*
*
«implementation class»
BMPImage
«implementation class»
JPGImage
«implementation class»
JPEGImage
«implementation class»
WinImp
«implementation class»
UnixImp

Tradeoffs

- The decoupling of the implementation from the objects that use them increases extensibility.
- Client objects are not aware of implementation issues.
- Variations of shape are encapsulated in shape class
- Variations in drawing are encapsulated in drawing class

Builder

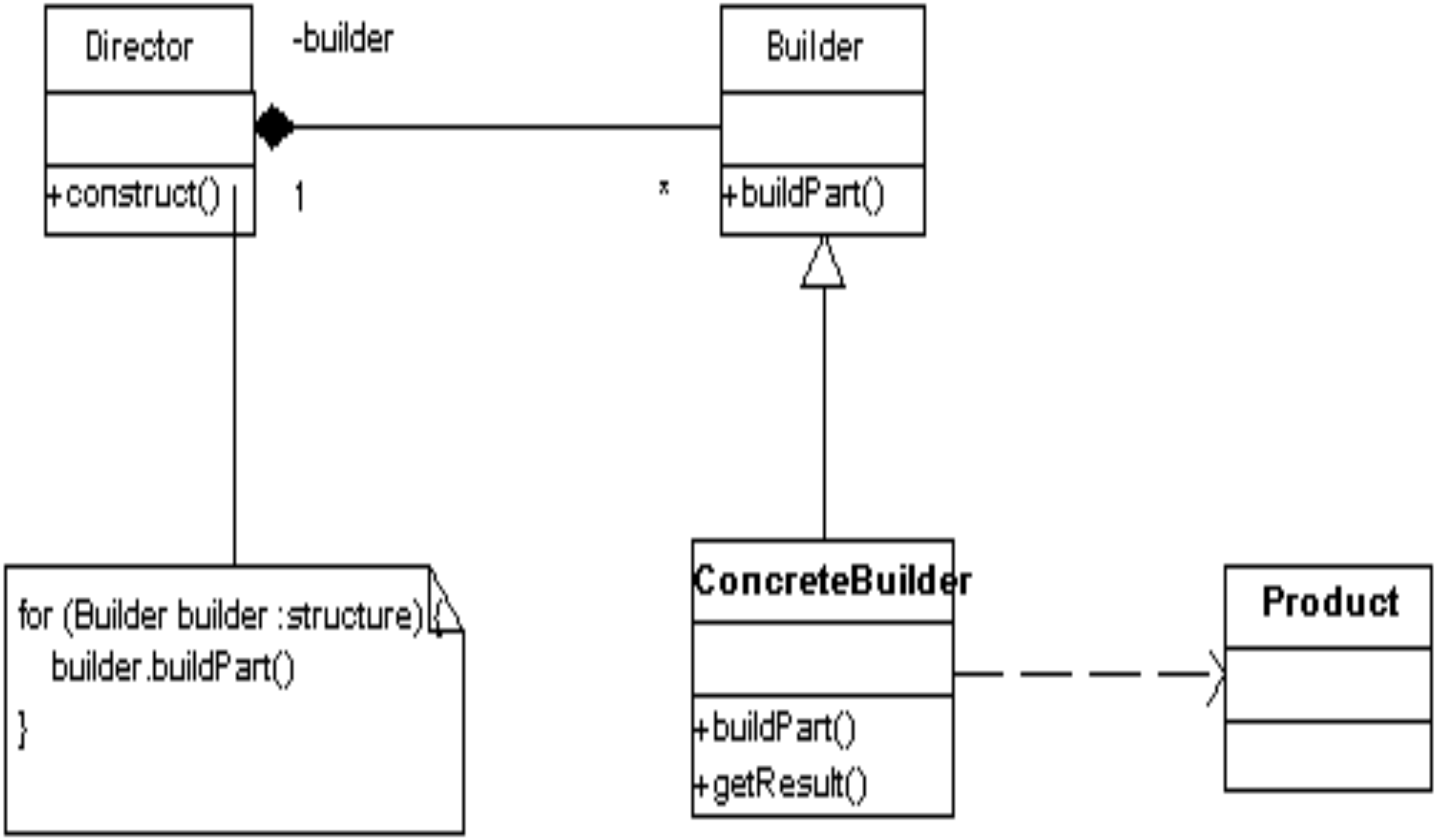
- A builder class has methods to build parts of a complex product, and to retrieve the completed product.
- Example: A document builder has methods to build paragraphs, tables, and so on.

Builder

Problem

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they' re assembled.
- The construction; process must allow different representations for the object that' s constructed.

Builder



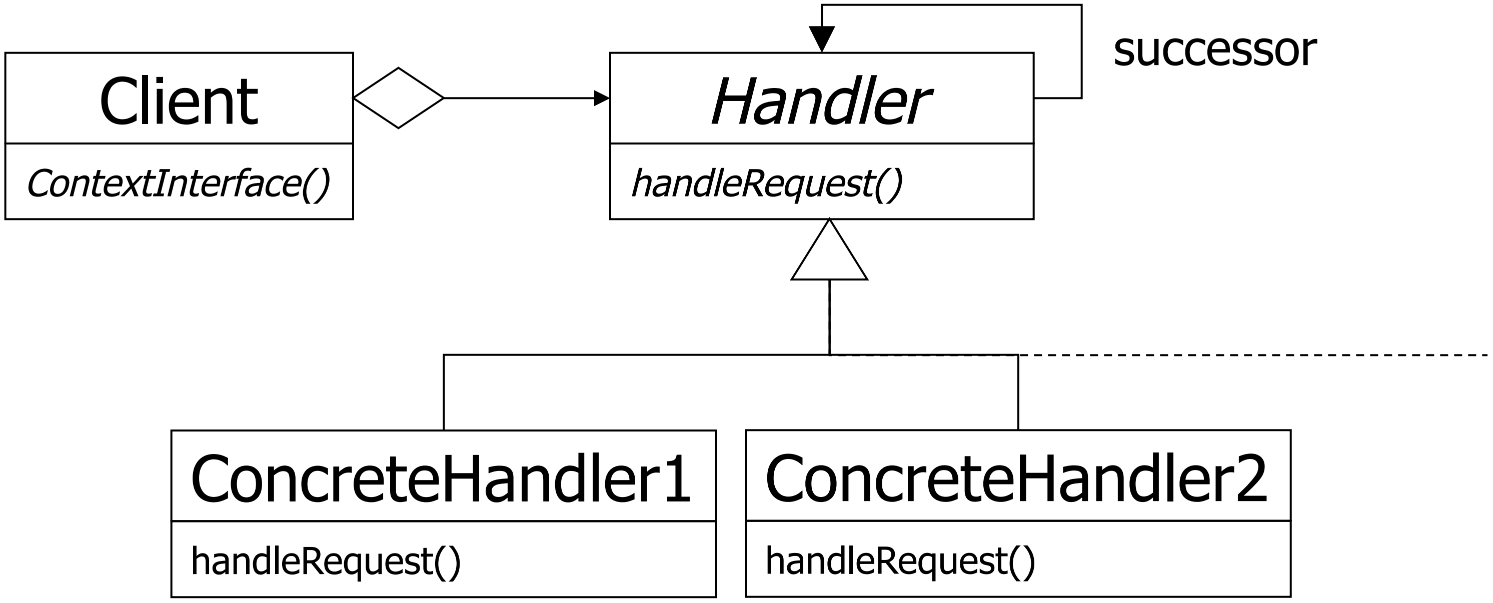
```
for (Builder builder :structure) {
    builder.buildPart()
}
```

Chain of Responsibility

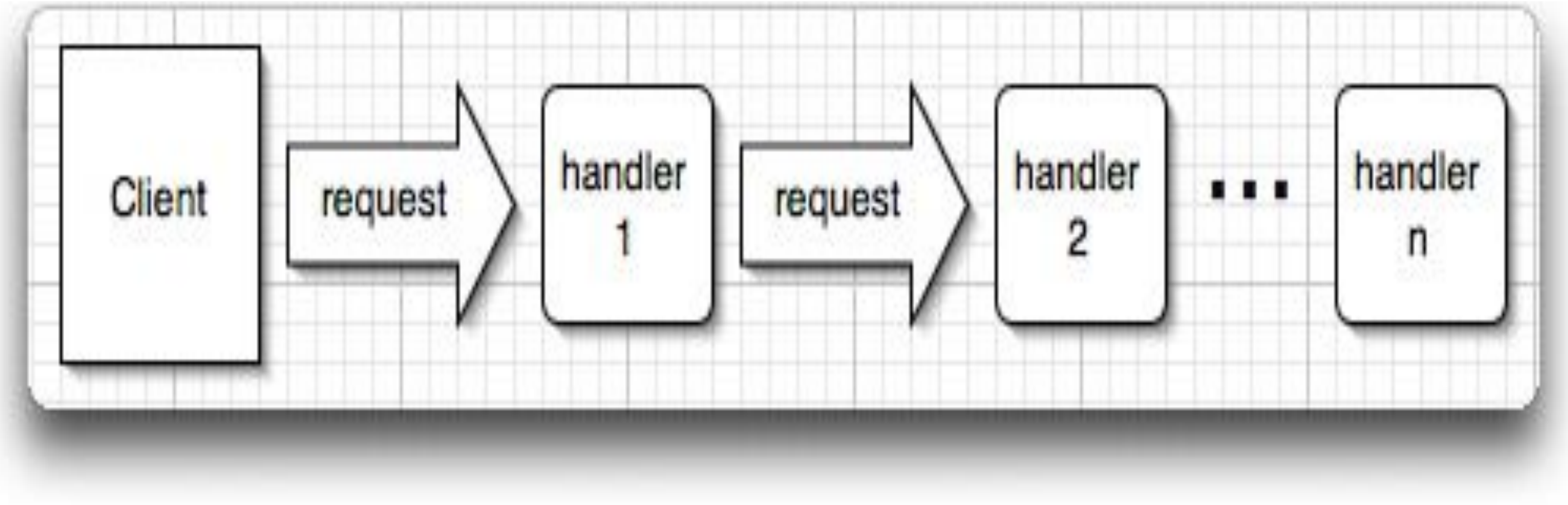
- A request is passed to the first handler in a chain. Each handler acts on the request (or chooses not to act) and passes the request on to the next handler.
- Example: An event handling mechanism passes a mouse or keyboard event to a component, which then passes it to the parent component.

Chain of Responsibility

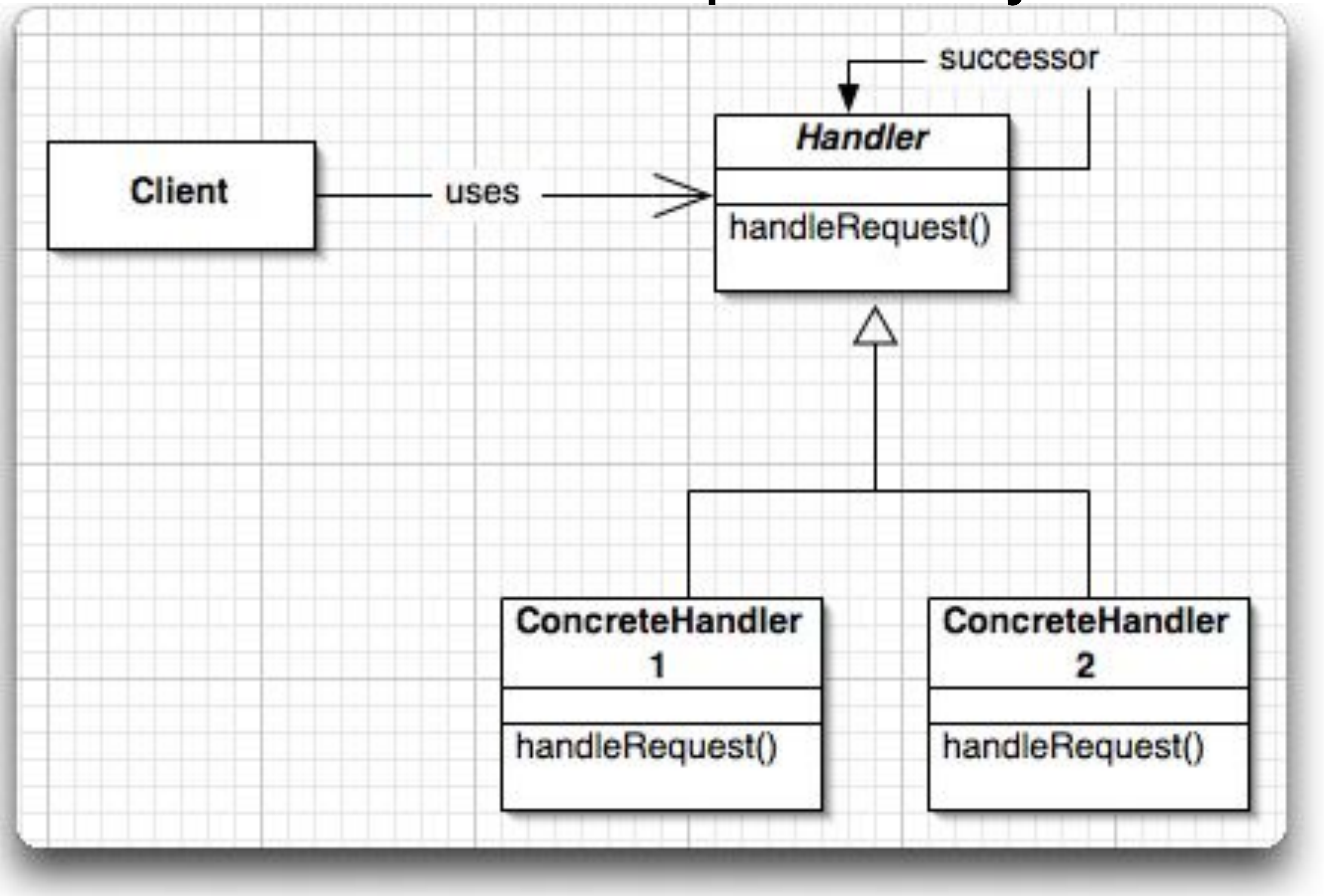
- Decouple sender of a request from receiver
- Give more than one object a chance to handle
- Flexibility in assigning responsibility
- Often applied with Composite



Chain of Responsiblity



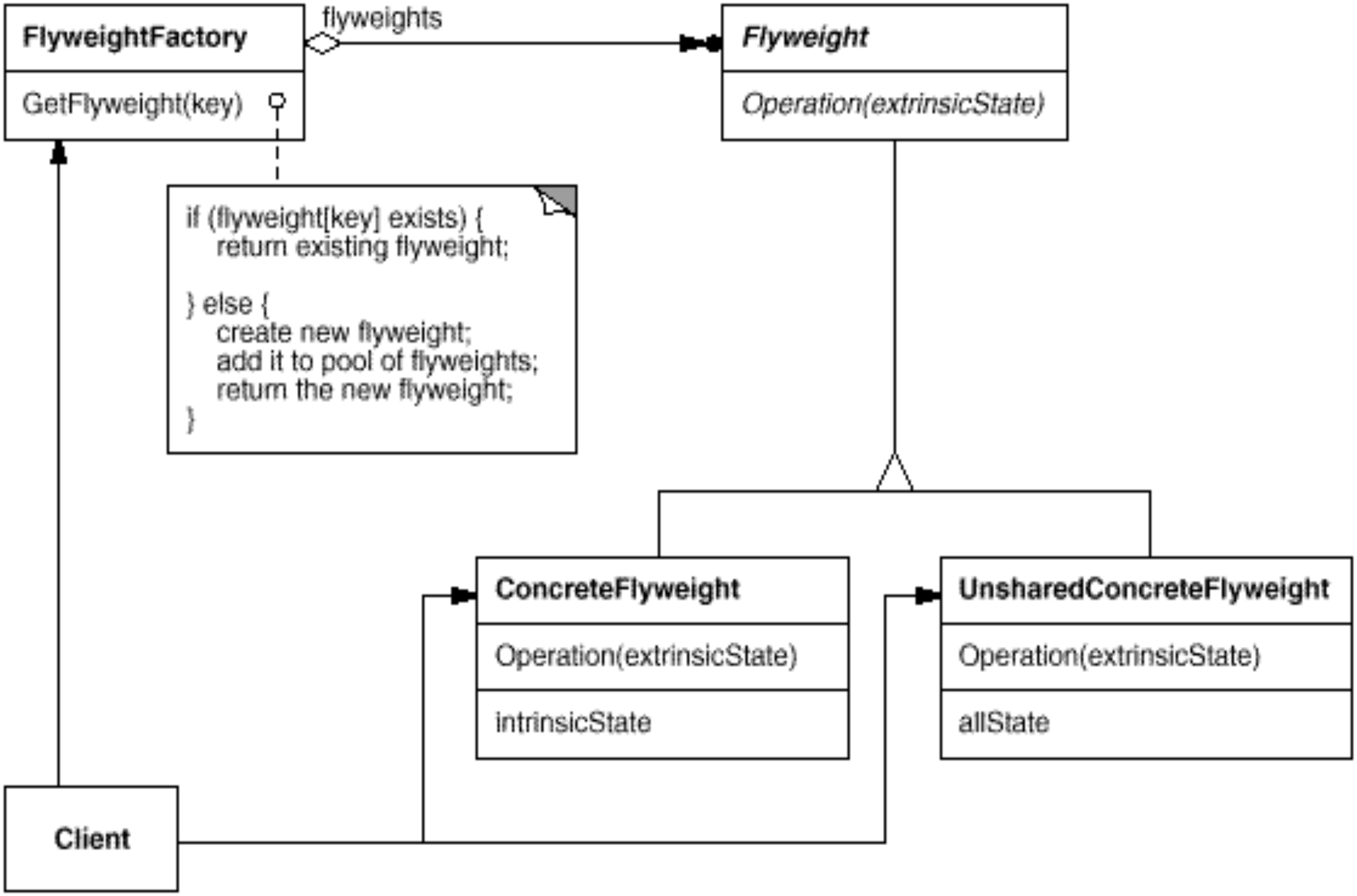
Chain of Reponsibility



Flyweight

- Use shared objects instead of large numbers of separate objects with identical state.
- Example: a word processor uses shared objects for styled characters rather than a separate object for each character.

Flyweight



FlyweightFactory

GetFlyweight(key)

flyweights

Flyweight

Operation(extrinsicState)

```
if (flyweight[key] exists) {
    return existing flyweight;

} else {
    create new flyweight;
    add it to pool of flyweights;
    return the new flyweight;
}
```

ConcreteFlyweight

Operation(extrinsicState)

intrinsicState

UnsharedConcreteFlyweight

Operation(extrinsicState)

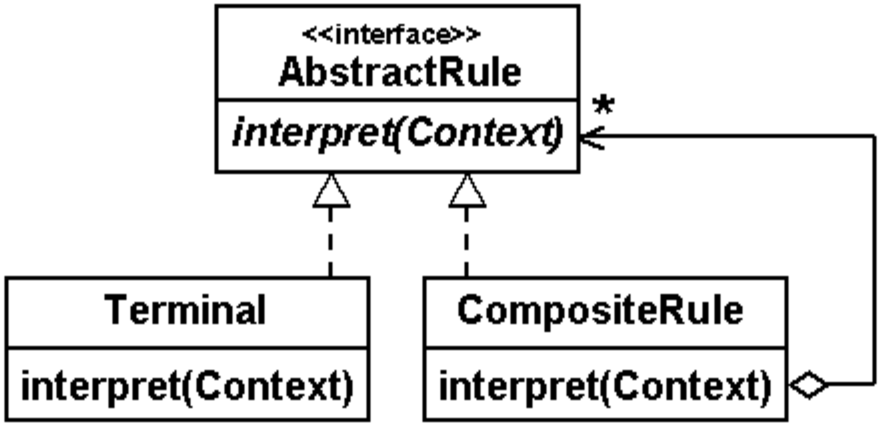
allState

Client

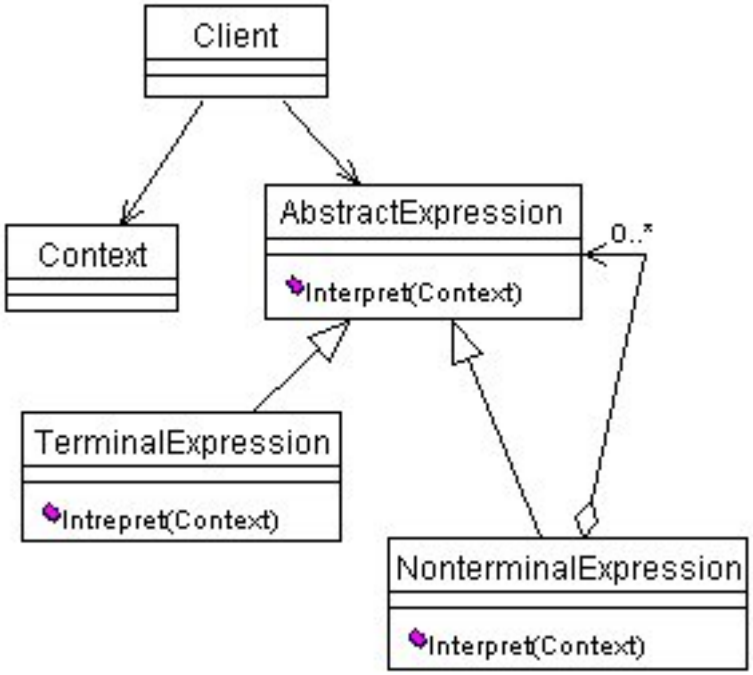
Interpreter

- A class hierarchy represents grammar rules. The interpreter recursively evaluates a parse tree of rule objects.
- Example: a program interactively evaluates mathematical expressions by building and evaluating a parse tree.

Interpreter



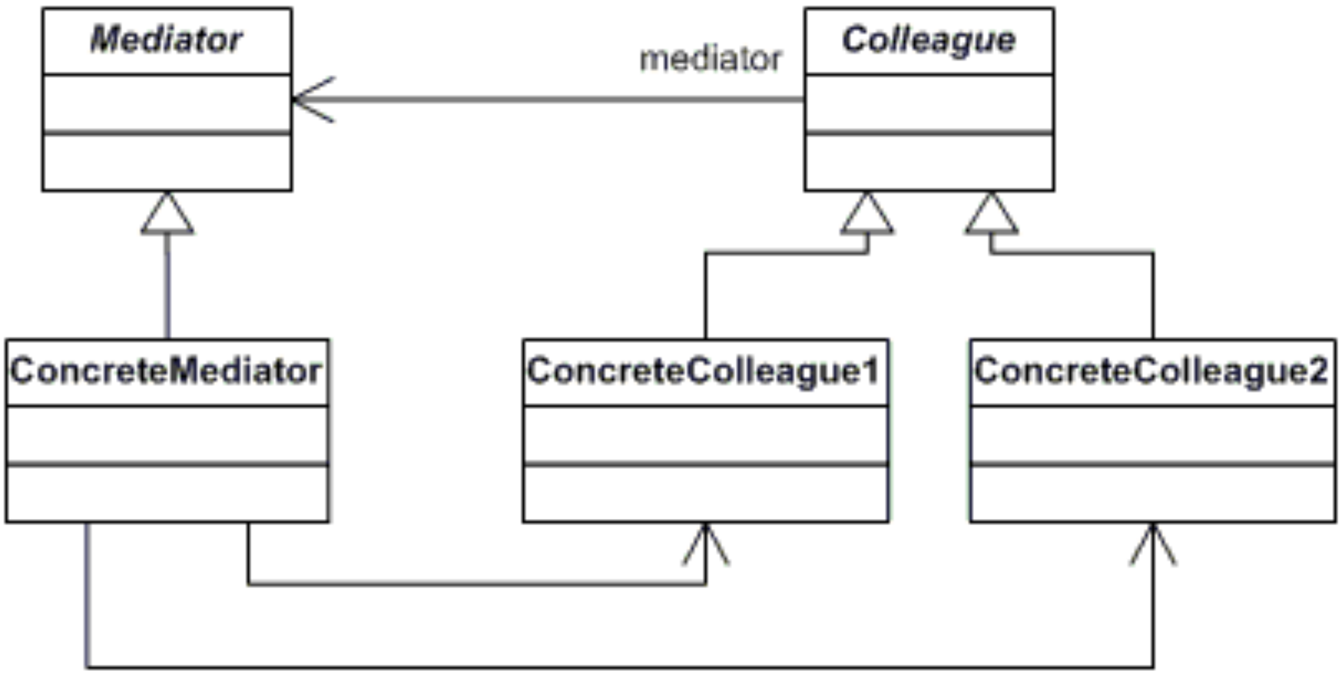
Interpreter



Mediator

- An object encapsulates the interaction of other objects.
- Example: All components in a dialog box notify a mediator of state changes. The mediator updates affected components.

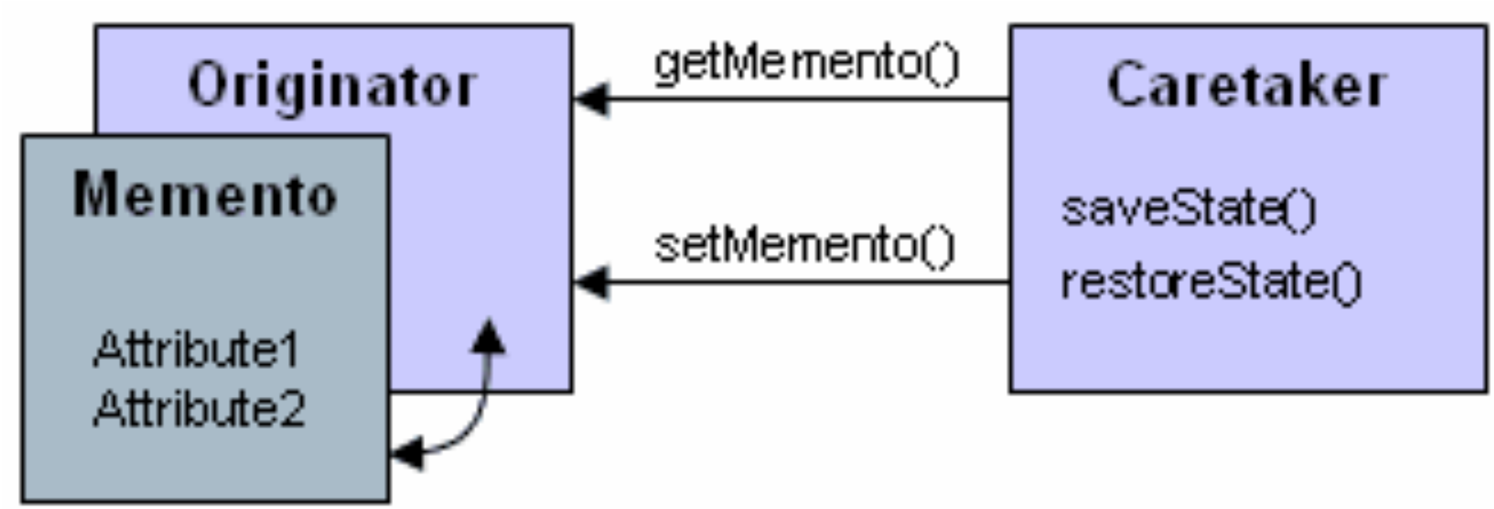
Mediator


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

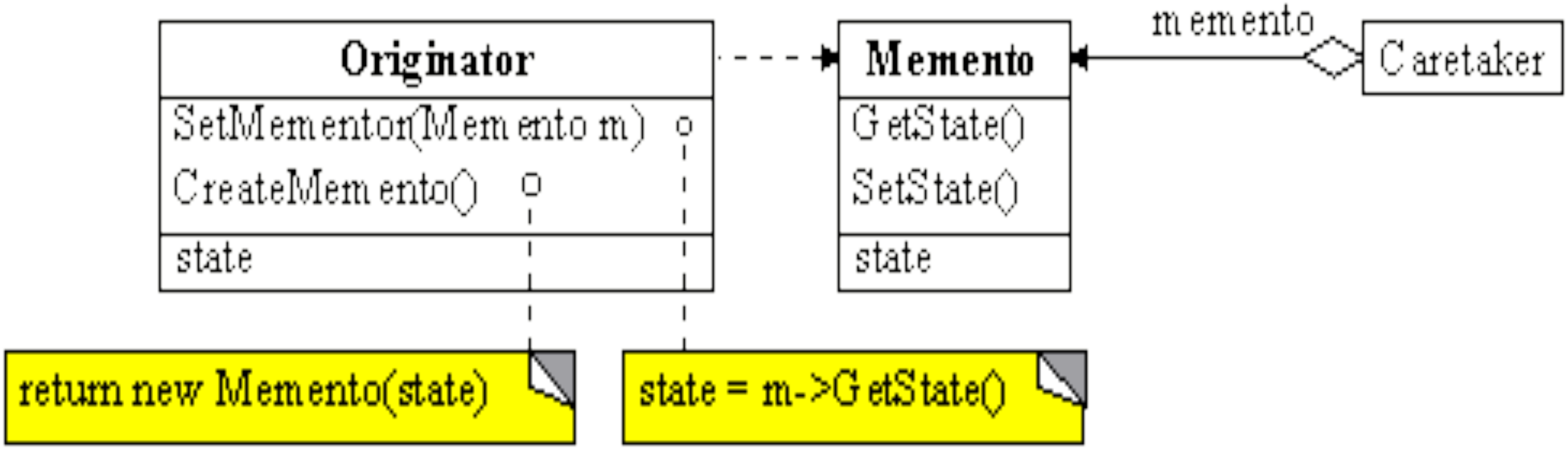
Memento

- An object yields an opaque snapshot of a part of its state, and can later return to its state from that snapshot.
- Example: An “undo” mechanism requests a memento from an object before mutating it. If the operation is undone, the memento is used to roll the object back to its old state.

Memento


Originator
Memento
Attribute1
Attribute2
getMemento()
setMemento()
Caretaker
saveState()
restoreState()

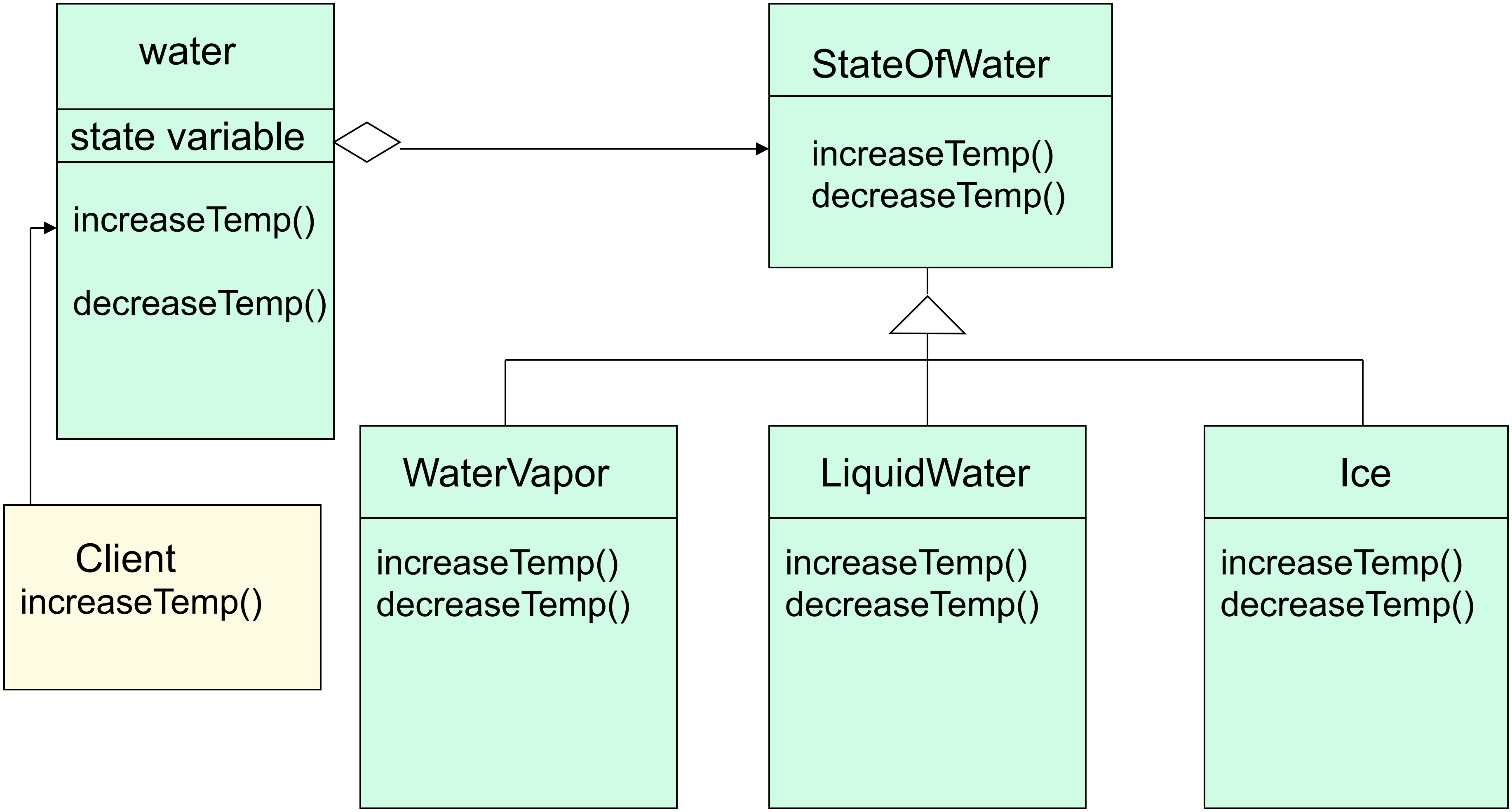
Memento


Originator
SetMementor(Memento m)
CreateMemento()
state
Memento
GetState()
SetState()
state
memento
Caretaker
return new Memento(state)
state = m->GetState()

State

- A separate object is used for each state. State-dependent code is distributed over the various state classes.
- An image editor has different drawing states. Each state is handled by a separate “tool” object.

State

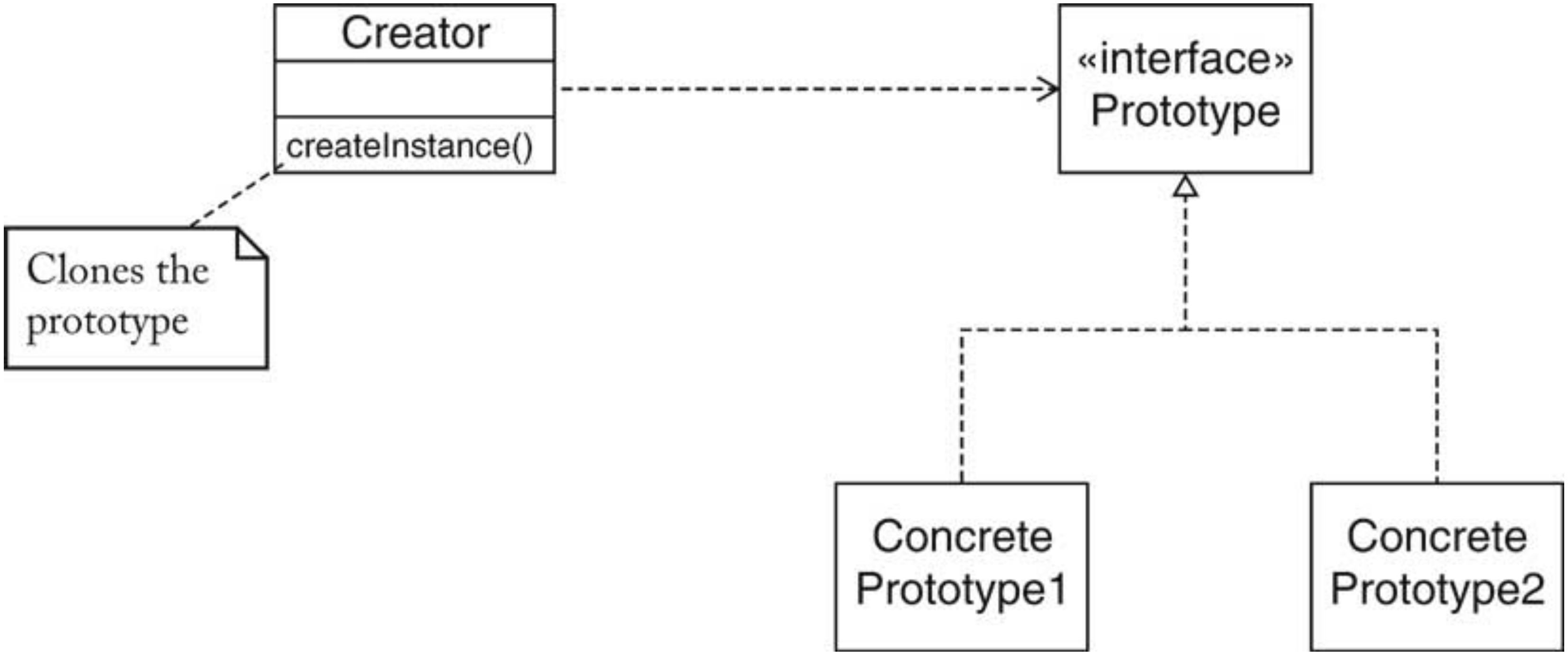

water
state variable
increaseTemp()
decreaseTemp()
StateOfWater
increaseTemp()
decreaseTemp()
Client
increaseTemp()
WaterVapor
increaseTemp()
decreaseTemp()
LiquidWater
increaseTemp()
decreaseTemp()
Ice
increaseTemp()
decreaseTemp()

PROTOTYPE Pattern

- **Context**
  - A system instantiates objects of classes that are not known when the system is built.
  - You do not want to require a separate class for each kind of object.
  - You want to avoid a separate hierarchy of classes whose responsibility it is to create the objects.

PROTOTYPE Pattern

- **Solution**
  - Define a prototype interface type that is common to all created objects.
  - Supply a prototype object for each kind of object that the system creates.
  - Clone the prototype object whenever a new object of the given kind is required.


Creator
createInstance()
Clones the prototype
«interface»
Prototype
Concrete Prototype1
Concrete Prototype2

PROTOTYPE Pattern

Name in Design Pattern	**Actual name (graph editor)**
Prototype	Node
ConcretePrototype1	CircleNode
Creator	The GraphPanel that handles the mouse operation for adding new nodes

Applicability

- Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented;
- when the classes to instantiate are specified at run-time, for example, by dynamic loading; or

Applicability

- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
- when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Design Pattern Space

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

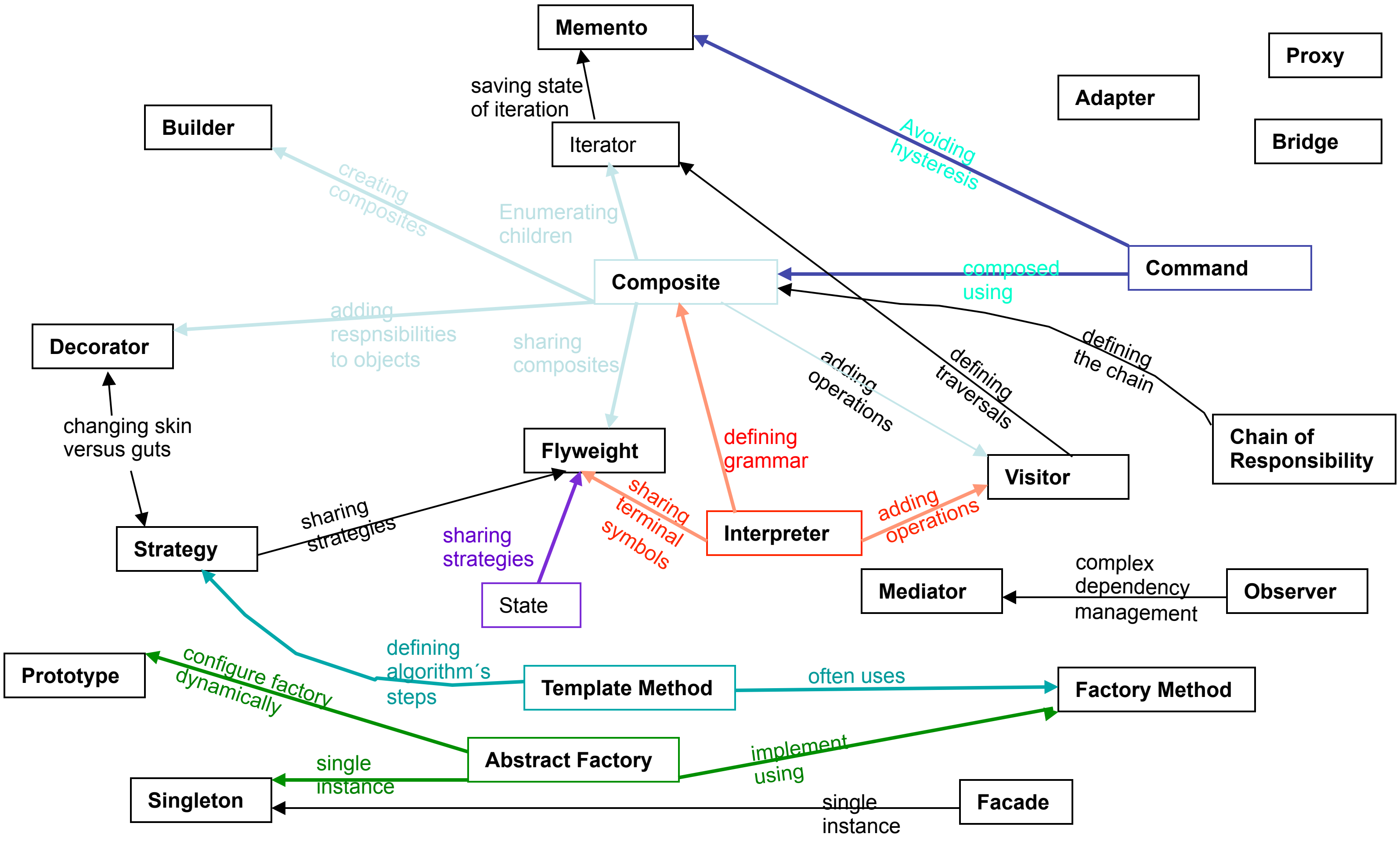
Defer object creation to another class

Defer object creation to another object

Describe ways to assemble objects

Describe algorithms and flow control

Relations among Design Patterns



Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
Avoiding hysteresis
Bridge
creating composites
Enumerating children
Command
Composite
composed using
adding respnsibilities to objects
Decorator
sharing composites
adding operations
defining traversals
defining the chain
changing skin versus guts
Flyweight
defining grammar
Chain of Responsibility
Visitor
sharing strategies
sharing terminal symbols
adding operations
Strategy
sharing strategies
Interpreter
State
Mediator
complex dependency management
Observer
Prototype
configure factory dynamically
defining algorithm´s steps
Template Method
often uses
Factory Method
Abstract Factory
single instance
implement using
Singleton
single instance
Facade