F. Tip and
M. Weintraub

DESIGN PATTERNS

Thanks go to Andreas Zeller for allowing incorporation of his materials

STRUCTURAL PATTERNS

- concerned with how classes and objects are composed to form larger structures
  - **structural class patterns**: use inheritance to compose interfaces or implementations
  - **structural object patterns**: describe ways to compose objects to realize new functionality

- Adapter
- Composite
- Proxy
- Flyweight
- Façade
- Bridge
- Decorator

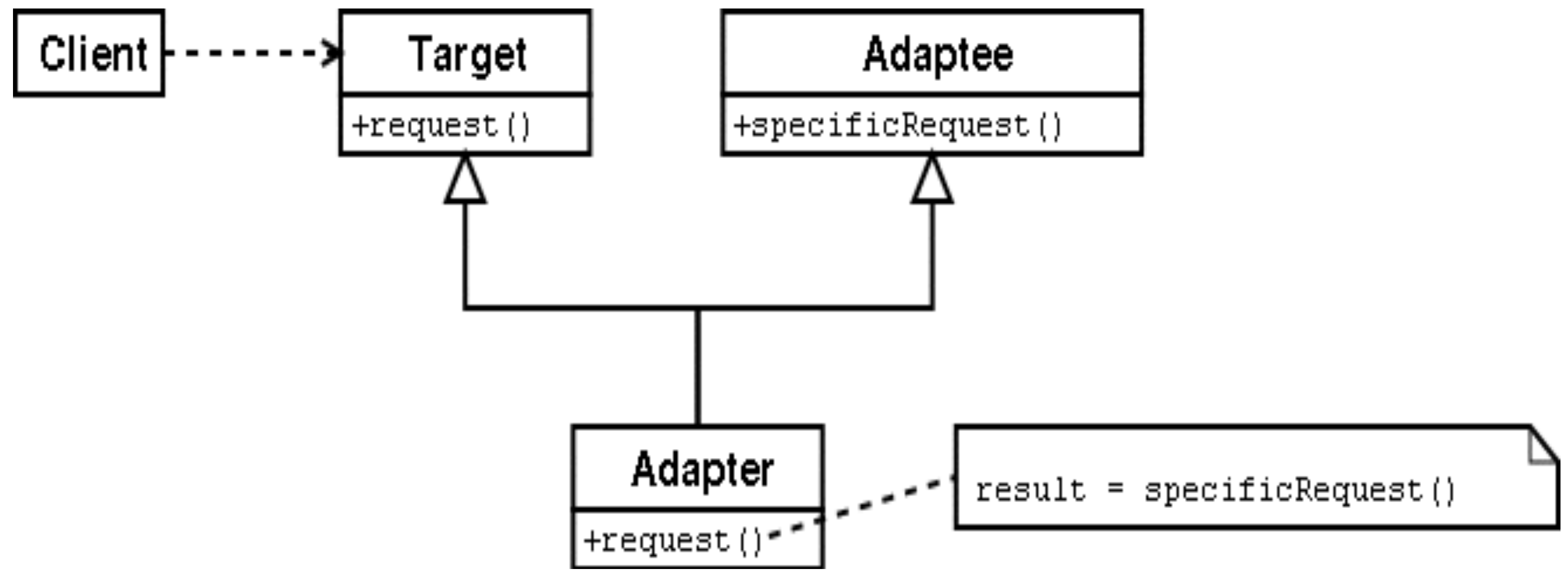
ADAPTER

- converts the interface of a class into another interface that clients expect
  - **Class Adapter**: uses (multiple) inheritance
  - **Object Adapter**: relies on object composition

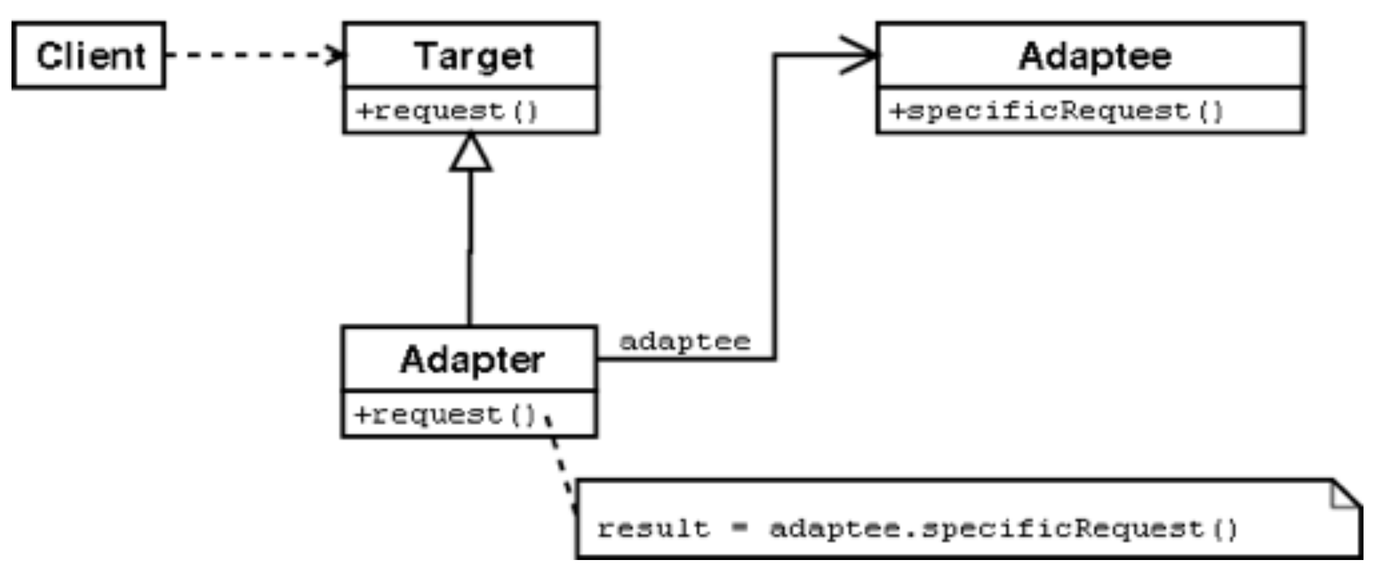
- use Adapter when:
  - you want to use an existing class, and its interface does not match the one you need
  - (object adapter) you need to use several existing subclasses, but it’s impractical to adapt their interface by subclassing every one.

ADAPTER: PARTICIPANTS

- **Target**
  - defines the interface that you need to implement
- **Client**
  - collaborates with objects conforming to the Target interface
- **Adaptee**
  - defines an existing interface that needs adapting
- **Adapter**
  - adapts the interface of Adaptee to the Target interface



class adapter



object adapter

ADAPTER: EXAMPLE

- suppose we have a **`Client`** application that uses a **`Stack`**, with operations **`push()`**, **`pop()`**, **`size()`**

- instead of implementing **`Stack`** from scratch, we want to use an existing **`Vector`** that provides almost the right functionality
  - Vector has methods **`elementAt()`**, **`insertElementAt()`**, **`size()`**

- solution: create a **`StackAdapter`** class
  - (class adapter) extends **`Vector`**, implements **`Stack`**
  - (object adapter) has pointer to a **`Vector`**, implements **`Stack`**

CLASS ADAPTER

```java
public class Client {
  public static void main(String[] args) {
    Stack<String> s =
      new StackAdapter<String>();
    s.push("foo");
    s.push("bar");
    System.out.println(s.pop());
    System.out.println(s.pop());
  }
}

interface Stack<T> {
  public void push(T t);
  public T pop();
  public int size();
}
```

```java
class StackAdapter<T> extends Vector<T>
                      implements Stack<T> {
  StackAdapter(){
    super();
  }
  public void push(T t){
    insertElementAt(t, size());
  }
  public T pop(){
    T t = elementAt(size()-1);
    removeElementAt(size()-1);
    return t;
  }

  // inherit size() method from Vector
}
```

OBJECT ADAPTER

```java
public class Client {
  public static void main(String[] args) {
    Stack<String> s =
      new StackAdapter<String>();
    s.push("foo");
    s.push("bar");
    System.out.println(s.pop());
    System.out.println(s.pop());
  }
}

interface Stack<T> {
  public void push(T t);
  public T pop();
  public int size();
}
```

```java
class StackAdapter<T> implements Stack<T> {
  StackAdapter() {
    _adaptee = new Vector<T>();
  }
  public void push(T t) {
    _adaptee.insertElementAt(t, _adaptee.size());
  }
  public T pop() {
    T t = _adaptee.elementAt(_adaptee.size()-1);
    _adaptee.removeElementAt(_adaptee.size()-1);
    return t;
  }
  public int size() {
    return _adaptee.size();
  }
  private Vector<T> _adaptee;
}
```

ADAPTER: TRADEOFFS

- **class adapters**:
  - adapts Adaptee to Target by committing to a specific Adapter class; will not work when we want to adapt a class and its subclasses
  - lets Adapter override/reuse some of Adaptee's behavior
  - introduces only one object, no additional pointer indirection is needed to get to Adaptee

- **object adapters**:
  - lets a single Adapter work with many Adaptees (and change them at run time)
  - makes it harder to override Adaptee behavior (requires subclassing of Adaptee, and making Adapter refer to the subclass)

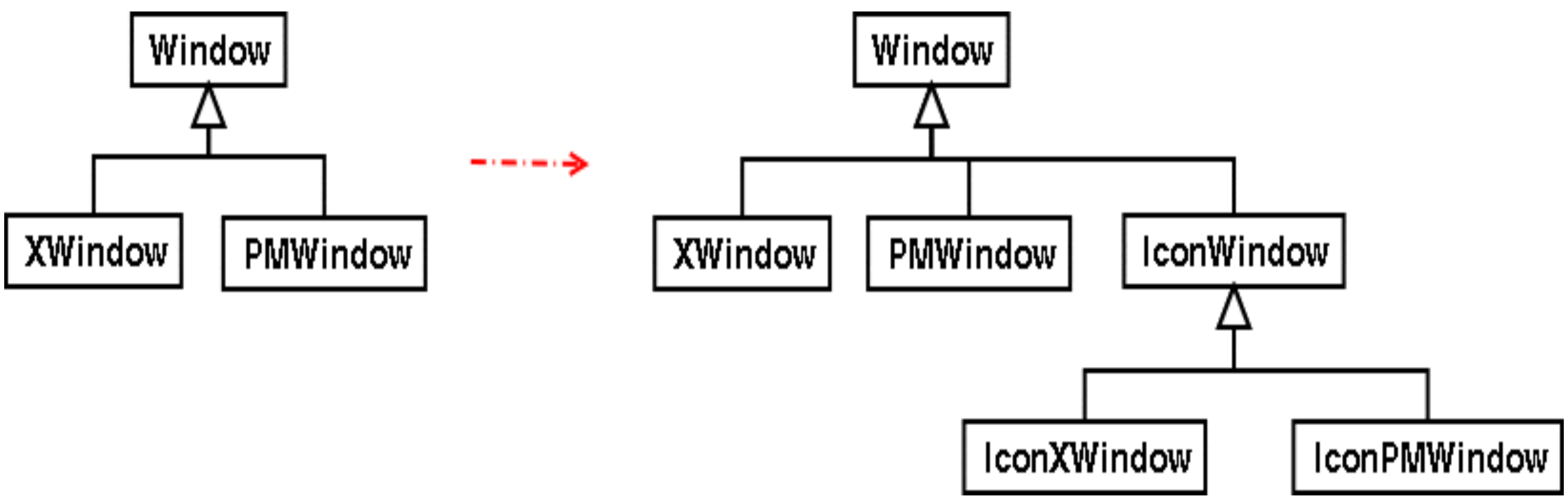
BRIDGE

- decouple an abstraction from its implementation so that the two can vary independently

- use **Bridge** when:
  - you want to avoid a permanent binding between an abstraction and its implementation
  - both the abstractions and implementations need to be subclassed and you want to avoid a proliferation of classes caused by extension in multiple, orthogonal extensions
  - you want to share an implementation among multiple objects, and hide this fact from the client

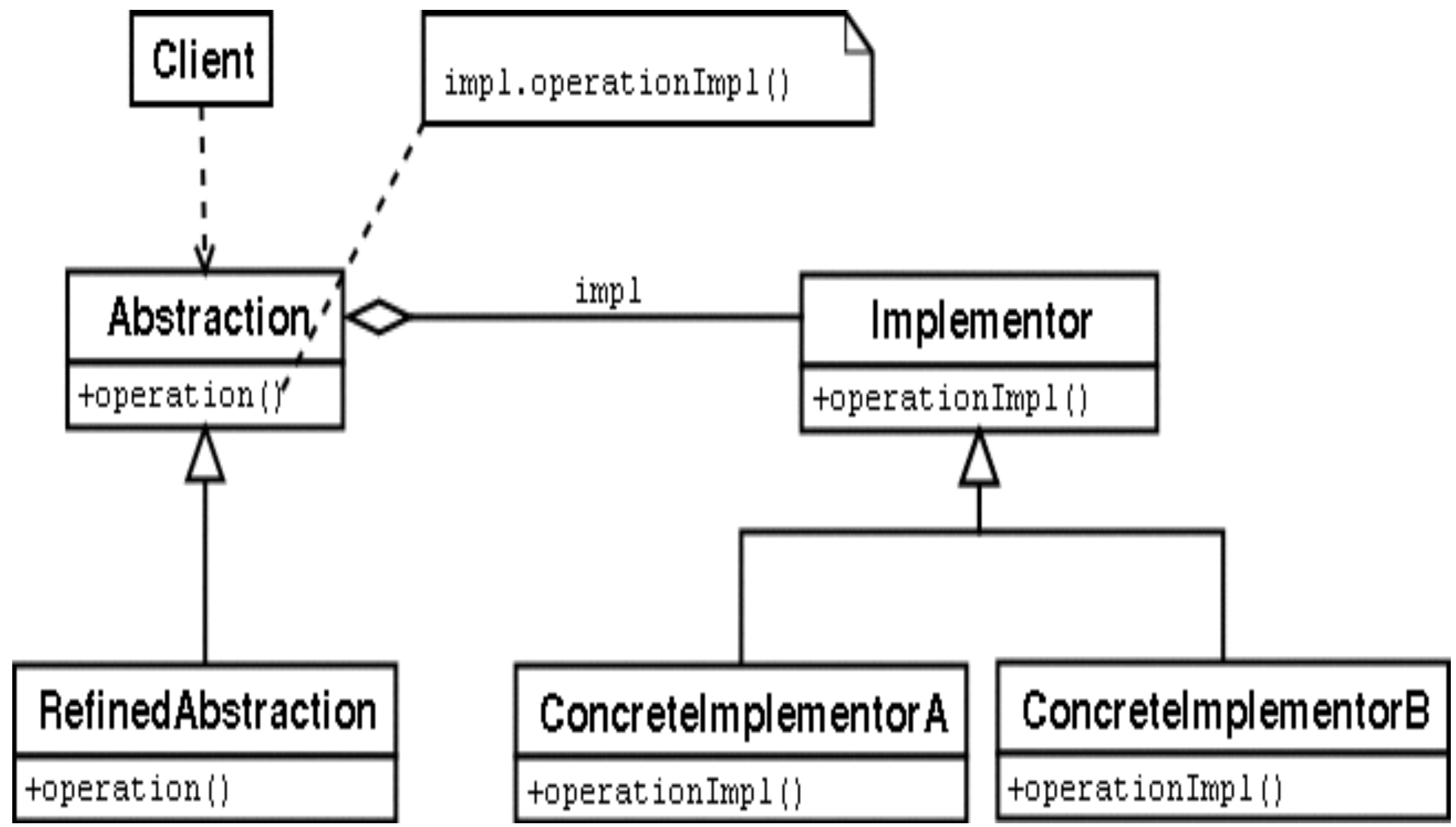
BRIDGE: WHEN TO APPLY?

- when extending a class hierarchy in multiple “dimensions” leads to:
  - an combinatorial explosion in number of classes
  - difficulties in sharing of implementations
  - exposure of platform dependencies to clients



BRIDGE: PARTICIPANTS

- **Abstraction**
  - defines the abstraction's interface
  - maintains a reference to an object of type Implementor
- **RefinedAbstraction**
  - extends the interface defined by Abstraction
- **Implementor**
  - defines the interface for the implementation classes; doesn't have to match interface of Abstraction
- **ConcreteImplementor**
  - implements the Implementor interface and defines its concrete implementation



BRIDGE: EXAMPLE

```java
public enum StackType {
   Array,
   LinkedList
}

public class Client {
  public static void main(String[] args) {
    Stack<String> s =
      new Stack<String>(StackType.Array);
    s.push("foo");
    s.push("bar");
    System.out.println(s.pop());
    System.out.println(s.pop());
  }
}
```

```java
class Stack<T> {
  Stack(StackType implType){
    switch (implType){
    case LinkedList:
      _impl = new LinkedListBasedStack<T>();
    case Array:
    default:
      _impl = new ArrayBasedStack<T>();
    }
  }
  public void push(T t){ _impl.push(t); }
  public T pop(){ return _impl.pop(); }

  private StackImpl<T> _impl;
}

interface StackImpl<T> {
  public void push(T t);
  public T pop();
}
```

TWO IMPLEMENTATIONS

```
class ArrayBasedStack<T>
        implements StackImpl<T> {
  public void push(T t){
    if ( !(_size == MAX_SIZE-1)){
      _elements[++_size] = t; }
  }
  public T pop(){
    if ((_size == -1)){ return null; }
    return _elements[_size--];
  }
  private final int MAX_SIZE = 100;
  private T[] _elements =
      (T[])new Object[MAX_SIZE];
  private int _size = -1;
}
```

```
class LinkedListBasedStack<T>
        implements StackImpl<T> {
  private class Node {
    // details omitted
  }
  public void push(T t){
    if (_tail == null){
      _tail = new Node(t);
    } else {
      _tail.next = new Node(t);
      _tail.next.prev = _tail;
      _tail = _tail.next;
    }
  }
  public T pop(){
    if (_tail == null) return null;
    T ret = _tail.value;
    _tail = _tail.prev;
    return ret;
  }
  private Node _tail;
}
```

BRIDGE VS. ADAPTER

- Adapter and Bridge lead to code that looks quite similar.

- However, they serve different purposes:
  - **Adapter** is *retrofitted* to make existing unrelated classes work together.
  - **Bridge** is *designed up-front* to let the abstraction and the implementation vary independently.

COMPOSITE

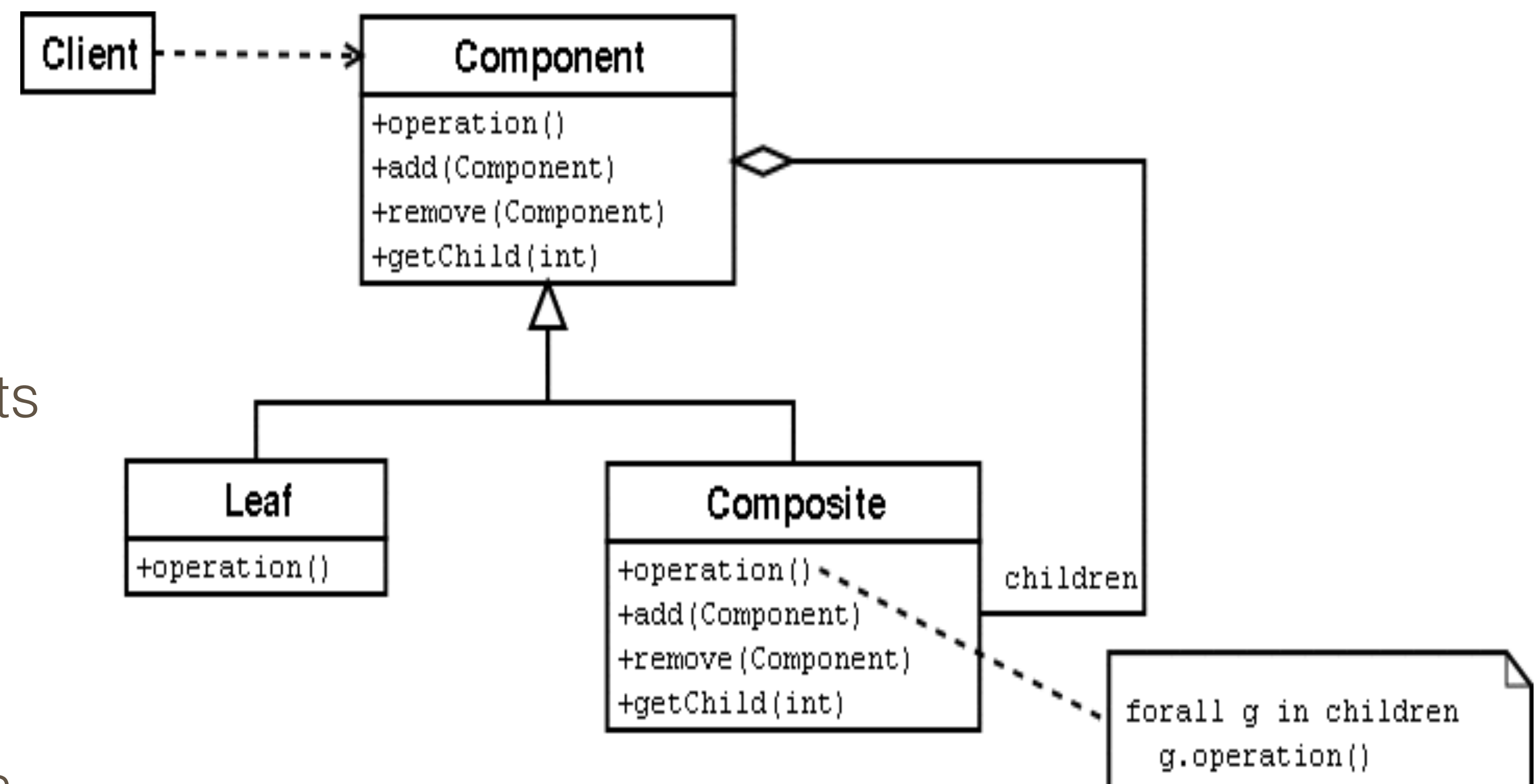
- Compose objects into tree structures to represent part-whole hierarchies.

- **Composite** lets you treat individual objects and compositions of objects uniformly.

- Apply Composite when:
  - you want to model part-whole hierarchies of objects
  - you want clients to be able to ignore the difference between compositions of objects and individual objects

COMPOSITE: PARTICIPANTS

- **Component**
  - declares common interface
  - implements default behavior
  - declares interface for accessing/ managing child components and (optional) for accessing parent
- **Leaf**
  - represents leaf objects
  - defines behavior for primitive objects
- **Composite**
  - defines behavior for components with children
  - stores child components
  - implements child-related operations in Component
- **Client**
  - manipulates objects via the Component interface



COMPOSITE EXAMPLE: UNIX FILE SYSTEMS

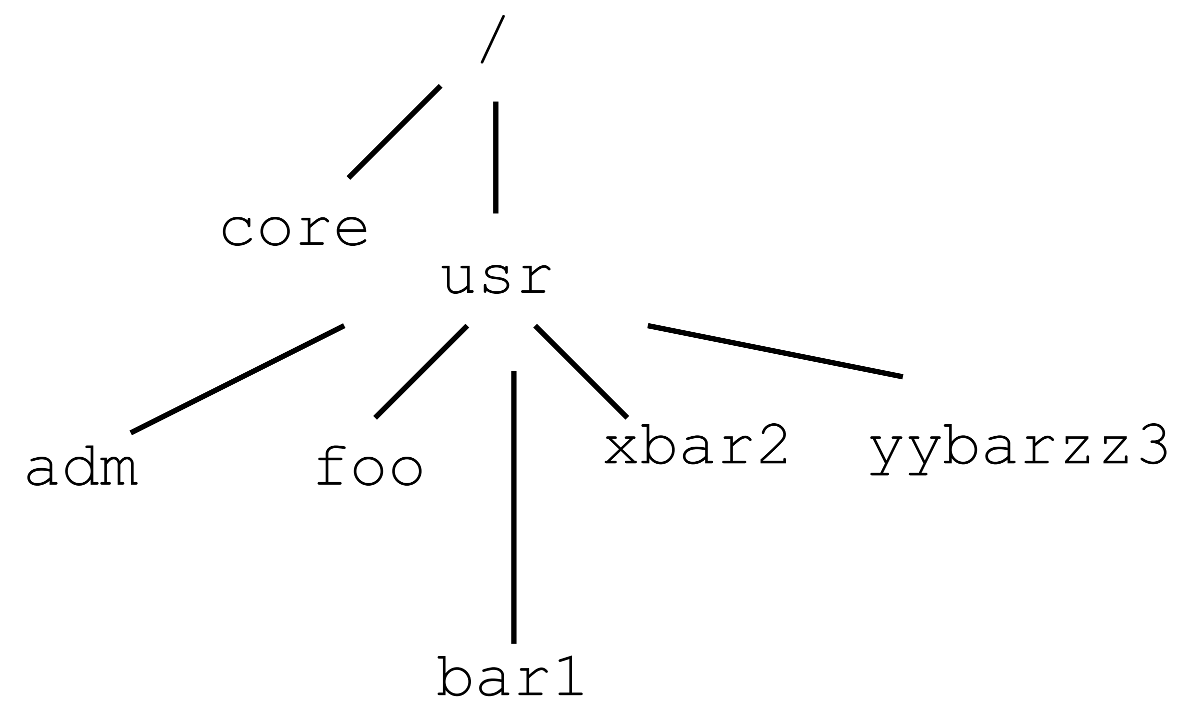
- a **Node** (Component) is a:
  - **File** (Leaf) or a
  - **Directory** (Composite)

- the **`find`** command can be used to find files with a particular name
  - uses auxiliary operation **`getAbsoluteName()`**

- usage: **`find <directory> -name <pattern>`**
  - **`find . -name "*.java"`** finds all Java source files in the current directory and its subdirectories and prints their absolute name
  - we consider a simplified version: a method **`Node.find(s)`** that finds all the files whose name contains **`s`** as a substring.

CLIENT PROGRAM

```java
public class Main {
  public static void main(String[] args){
    Directory root = new Directory("");
    Directory usr = new Directory("usr", root);
    new File("core", root);
    new File("adm", usr);
    new Directory("foo", usr);
    new File("bar1", usr);
    new File("xbar2", usr);
    new Directory("yybarzz3", usr);
    System.out.println(root.find("bar"));
  }
}
```



prints

```
[/usr/bar1, /usr/xbar2, /usr/yybarzz3/]
```

NODE, FILE, DIRECTORY

```
abstract class Node {
  Node(String name, Directory parent) { … }
  public String getAbsoluteName() { … }
  public String toString() {
    return getAbsoluteName();
  }
  public abstract List<String> find(String s);
  protected String _name;
  protected Directory _parent;
}

class File extends Node {
  File(String n, Directory p){
    super(n,p);
  }
  public List<String> find(String s){
    List<String> result =
      new ArrayList<String>();
    if (_name.indexOf(s) != -1){
      result.add(this.getAbsoluteName());
    }
    return result;
  }
}
```

```
class Directory extends Node {

  Directory(String n){ this(n, null); }
  Directory(String n, Directory p){ … }
  public String getAbsoluteName(){ … }
  public void add(Node n){
    _children.add(n);
  }

  public List<String> find(String s){
    List<String> result =
      new ArrayList<String>();
    if (_name.indexOf(s) != -1){
      result.add(getAbsoluteName());
    }
    for (Node child : _children){
      result.addAll(child.find(s));
    }
    return result;
  }
  private List<Node> _children;
}
```

COMPOSITE: CONSIDERATIONS

- composite makes clients more *uniform*
  - some operations only make sense for leaf or composite classes, but not for both
  - composite makes it easy to add new kinds of components

- implementation issues:
  - need explicit parent reference in Component
  - sharing components for efficiency (→**Flyweight**)
  - storage management issues
  - child ordering relevant or not (→**Iterator**)
  - caching traversal/search information for efficiency

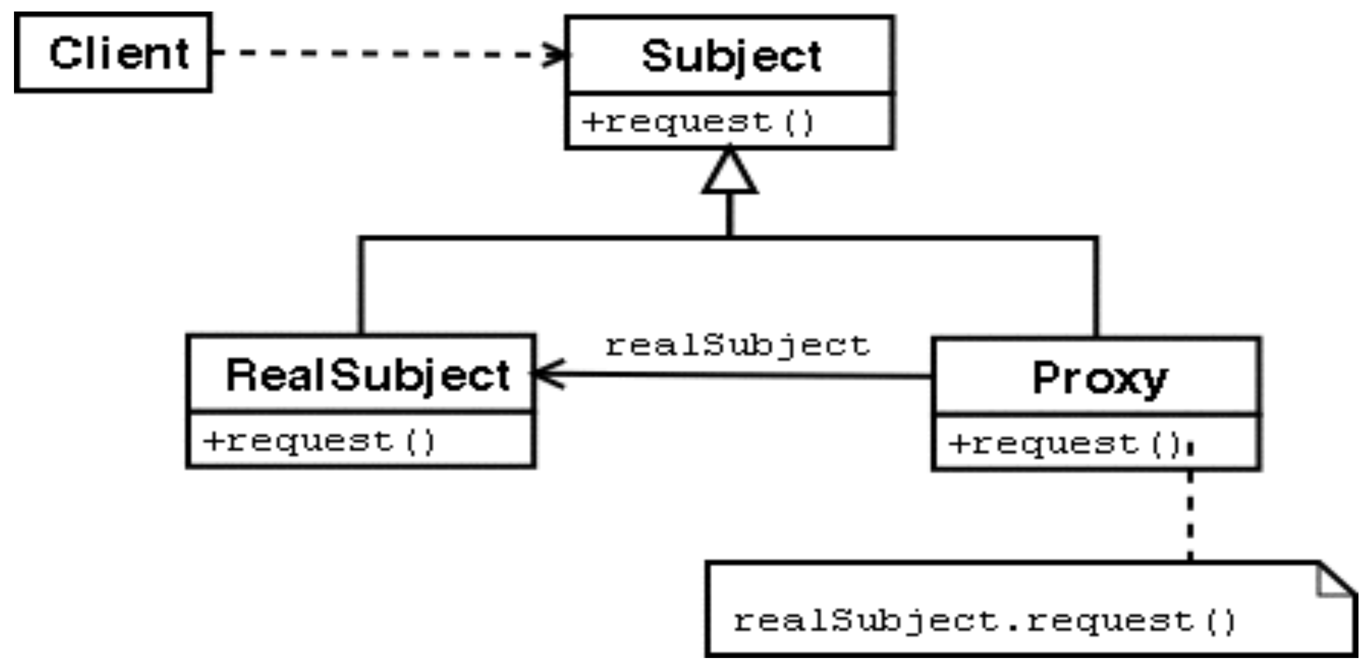
PROXY

- Proxy provides a surrogate or placeholder for another object to control access to it

- Apply **Proxy** when:
  - you need a local representative for an object that lives in a different address space (remote proxy)
  - you want to avoid the creation of expensive objects until they are really needed (virtual proxy)
  - you want to control access to an object (protection proxy)
  - you need a smart pointer that performs additional actions when an object is accessed (e.g., reference-counting, loading persistent objects into memory)

PROXY: PARTICIPANTS

- **Proxy**
  - maintains reference that lets proxy access real subject
  - provides an **interface identical** to the subject's
  - controls access to the real subject, and may be responsible for creating & deleting it
  - other responsibilities:
    - remote proxies: encoding and transferring request
    - virtual proxies: caching information
    - protection proxies: check access permissions
- **Subject**
  - defines the **common interface** for RealSubject and Proxy so that Proxy can be used anywhere RealSubject is used
- **RealSubject**
  - defines the real object represented by the Proxy



PROXY EXAMPLE: SYMBOLIC LINKS

- in Unix, you can create symbolic links to files and directories with the "ln" command
  - syntax: **`ln -s <directory> <linkName>`**

- after this command, you can access the directory also via the link

- you can tell the **`find`** command to follow symbolic links by specifying the **`-follow`** option

- we now extend the File System example with symbolic links, implemented using Proxy

LINK

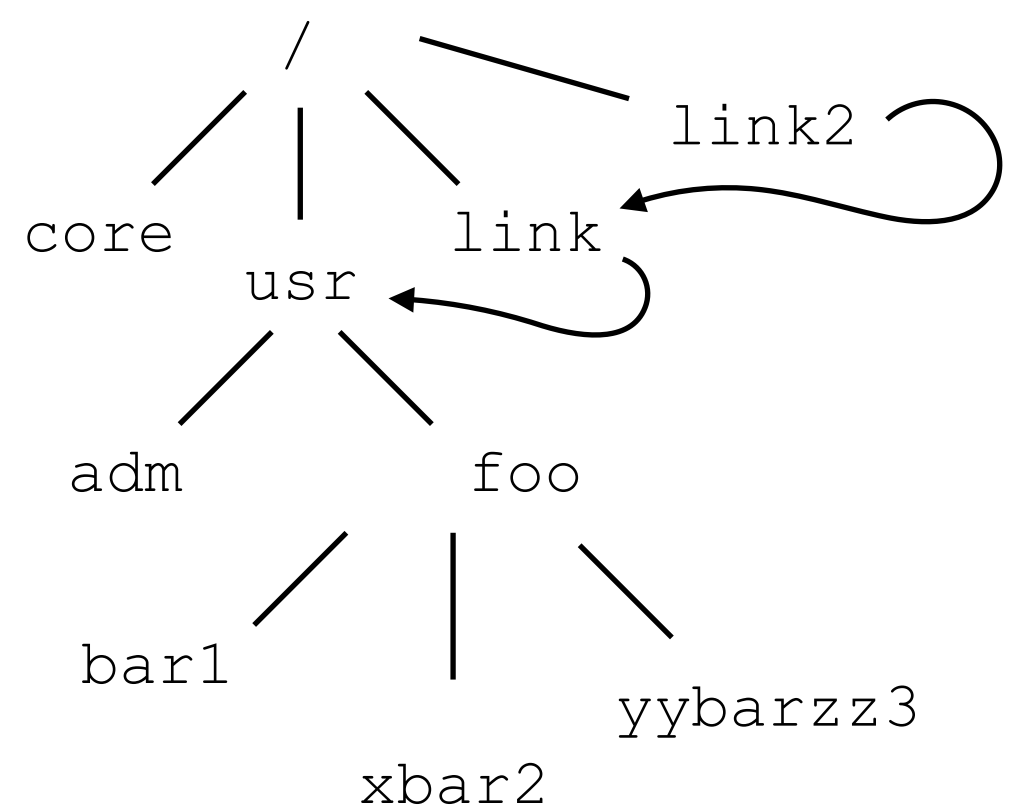
```java
class Link extends Node {
  Link(String n, Node w, Directory p){ … }
  public String getAbsoluteName(){ … }

  public Vector<String> find(String s){
    Vector<String> result = new Vector<String>();
    if (_name.indexOf(s) != -1){
      result.add(getAbsoluteName());
    }
    Vector<String> resultsViaLink = _realNode.find(s);
    int n = _realNode.getAbsoluteName().length();
    for (String r : resultsViaLink){
      String name = super.getAbsoluteName() + "/" + r.substring(n);
      result.add(name);
    }
    return result;
  }

  private Node _realNode;
}
```

UPDATED CLIENT PROGRAM

```java
public class Main {
  public static void main(String[] args){
    Directory root = new Directory("");
    new File("core", root);
    Directory usr = new Directory("usr", root);
    new File("adm", usr);
    Directory foo = new Directory("foo", usr);
    new File("bar1", foo);
    new File("xbar2", foo);
    new File("yybarzz3", foo);
    Link link = new Link("link", usr, root);
    new Link("link2", link, root);
    System.out.println(root.find("bar"));
  }
}
```



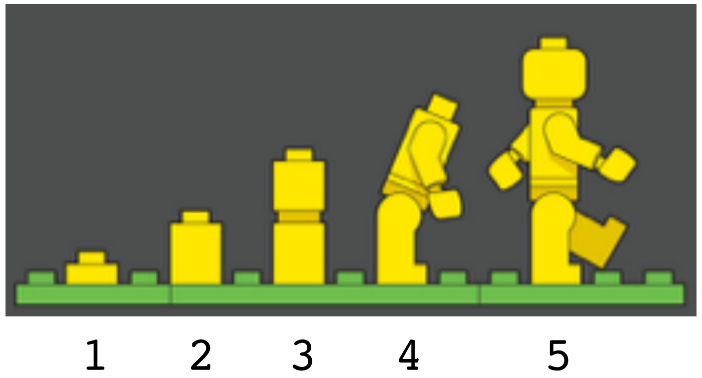

```
[/usr/foo/bar1, /usr/foo/xbar2, /usr/foo/yybarzz3, /link/foo/bar1, /link/foo/xbar2,
 /link/foo/yybarzz3, /link2/foo/bar1, /link2/foo/xbar2, /link2/foo/yybarzz3]
```

BEHAVIORAL PATTERNS

- concerned with algorithms and the assignment of responsibilities between objects
  - **behavioral class patterns** use inheritance to distribute behavior between classes
  - **behavioral object patterns** use composition to distribute behavior between objects

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

ITERATOR

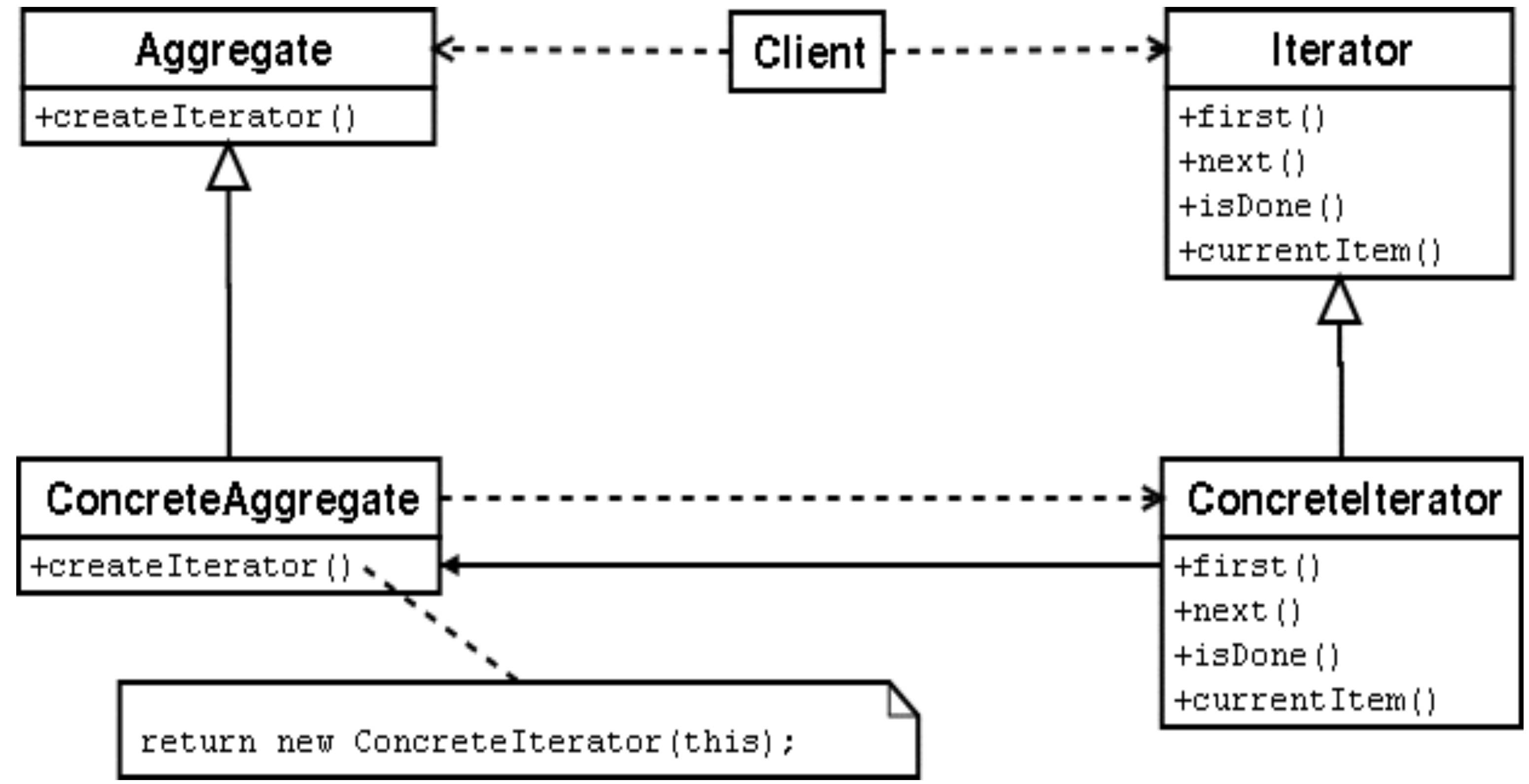


- provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

- apply **Iterator** for the following purposes:
  - to access an aggregate object's contents without exposing its internal representation
  - to support multiple traversals of aggregate objects
  - to provide a uniform interface for traversing different aggregate structures (support polymorphic iteration)

ITERATOR: PARTICIPANTS

- **Iterator**
  - defines an interface for accessing and traversing elements
- **ConcreteIterator**
  - implements the Iterator interface
  - keeps track of the current position in the traversal of the aggregate
- **Aggregate**
  - defines an interface for creating an Iterator object
- **ConcreteAggregate**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator



ITERATOR: EXAMPLE

- use Iterator to allow clients to iterate through the Files in a directory
  - without exposing Directory's internal structure to the client

```java
interface Iterator<T> {
  void first();
  void next();
  boolean isDone();
  T current();
}
```

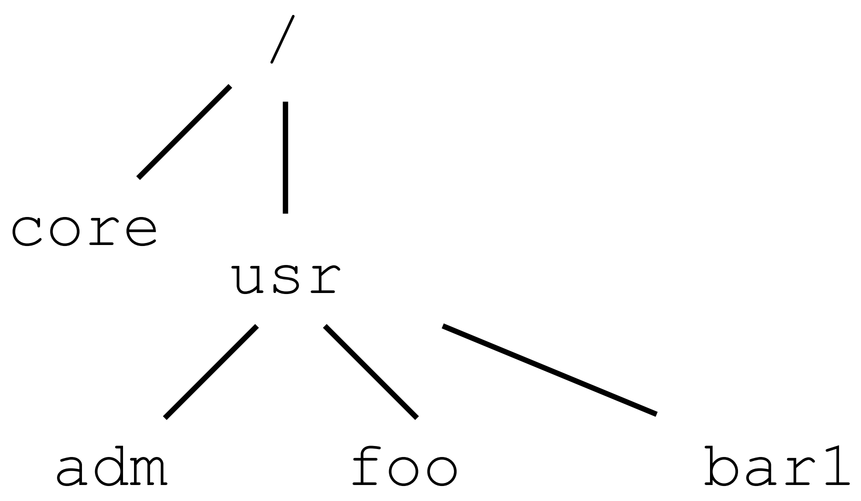
```java
class Directory extends Node {
  …
  private class DirectoryIterator implements Iterator<Node> {
    private List<Node> _files;
    private int _fileCnt;

    DirectoryIterator(Directory d) {
      _files = d._children; _fileCnt = 0;
    }
    public void first() { _fileCnt = 0; }
    public void next() { _fileCnt++; }
    public boolean isDone() {
      return _fileCnt == _files.size();
    }
    public Node current() {
      return _files.get(_fileCnt);
    }
  }
  …
```

CLIENT

```java
public class Main {
  public static void main(String[] args){
    Directory root = new Directory("");
    Directory usr = new Directory("usr", root);
    new File("core", root);
    new File("adm", usr);
    new Directory("foo", usr);
    new File("bar1", usr);

    // use iterator to print contents of /usr
    Iterator<Node> it = usr.iterator();
    for (it.first(); !it.isDone(); it.next()){
      Node n = it.current();
      System.out.println(n.getAbsoluteName());
    }
  }
}
```



prints:

```
/usr/adm
/usr/foo/
/usr/bar1
```

ITERATOR: CONSIDERATIONS

- two kinds of iterators:
  - **internal iterators**: iteration controlled by iterator itself. Client hands iterator operation to perform; iterator applies op. to each element
  - **external iterators**: client controls iteration (by requesting next element)
- some danger associated with external iterators
  - e.g., an element of the underlying collection may be removed during iteration. Iterators that can deal with this are called **robust**.
- iterators may support additional operations such as **`skipTo(int)`** or **`remove()`**
  - the Java libraries define an interface **`java.util.Iterator`** with **`hasNext()`**, **`next()`**, **`remove()`** methods
  - if **`remove()`** is not supported by a ConcreteIterator, an **`UnsupportedOperationException`** is thrown

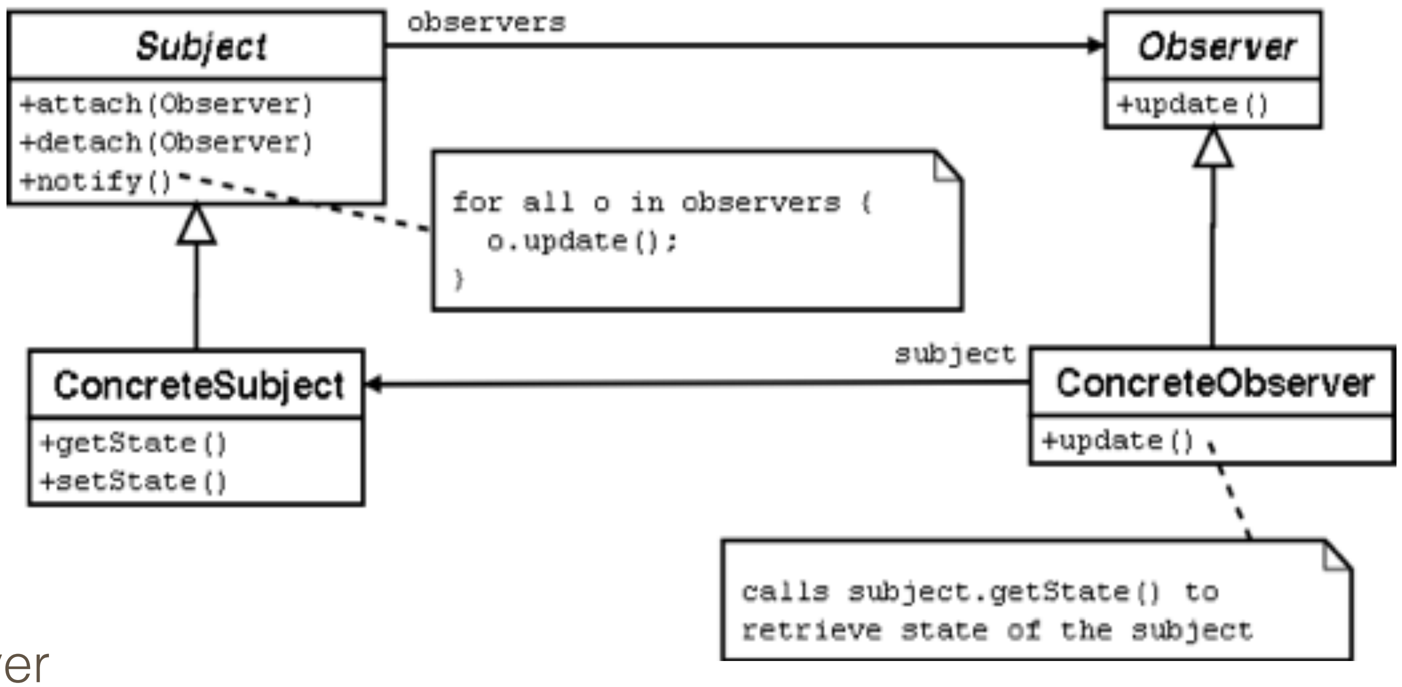
OBSERVER

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

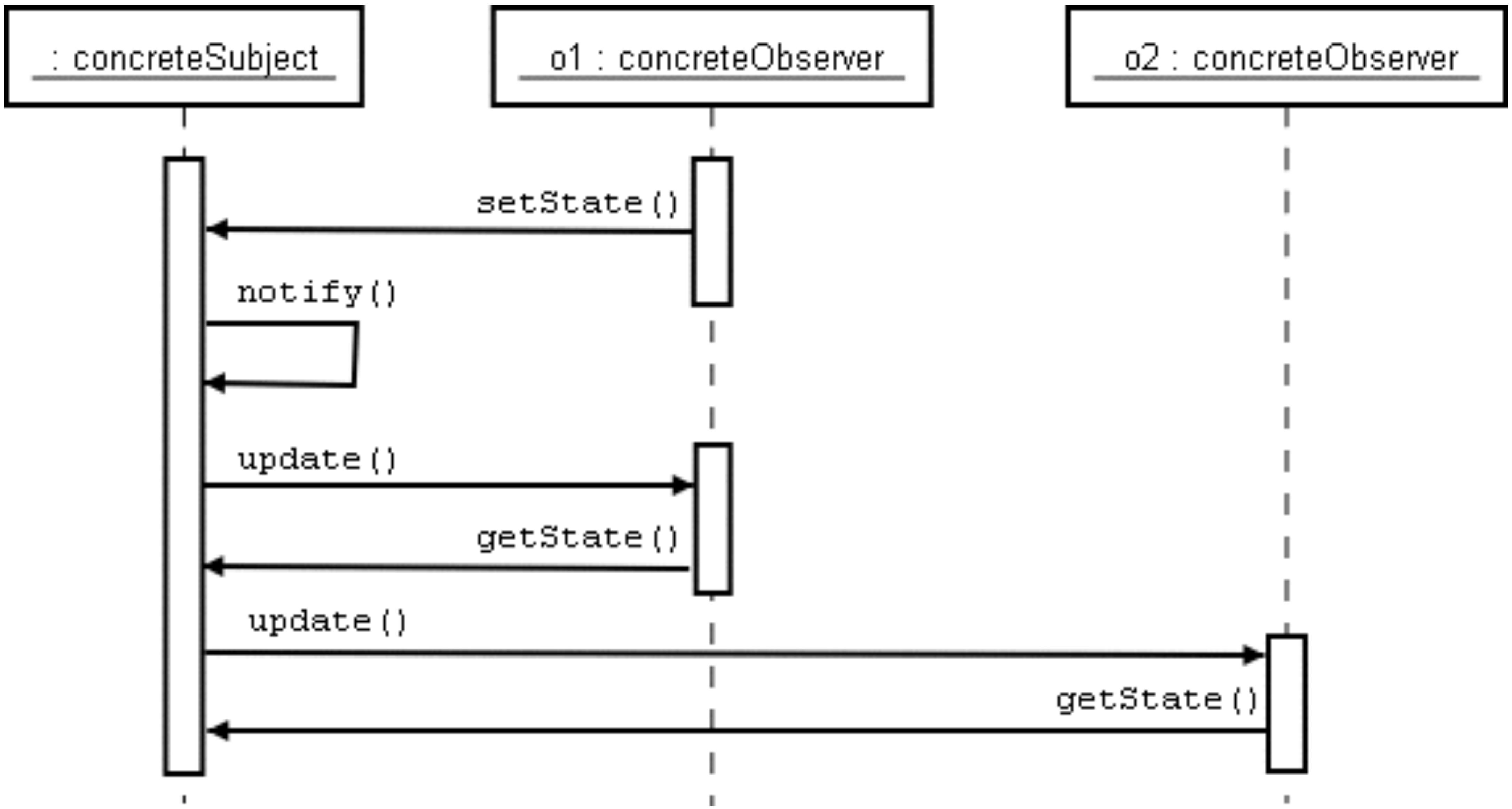
- apply **Observer** when
  - when an abstraction has two aspects, one dependent on the other.
  - when a change to one object requires changing others
  - when an object should be able to notify other objects without making assumptions about the identity of these objects

OBSERVER: PARTICIPANTS

- **Subject**
  - knows its observers. any number of observers may observe a subject
  - provides an interface for attaching/ detaching observers
- **Observer**
  - defines an updating interface for objects that should be notified of changes
- **ConcreteSubject**
  - stores state of interest to ConcreteObserver objects
  - sends a notification to its observers when state changes
- **ConcreteObserver**
  - maintains reference to a ConcreteSubject object
  - stores state that should stay consistent with subject's
  - implements the Observer updating interface to keep its state consistent with the subject's



OBSERVER: SEQUENCE DIAGRAM



OBSERVER: EXAMPLE

- add FileObservers to our FileSystem example.
  - add a method **`write(String)`** to class File to model operations that change a File's contents
  - associate FileObservers with Files; notify these after each write
  - FileObservers print a warning message that the file has changed

```java
interface Observer {
  public void update();
}
```

```java
class FileObserver implements Observer {
  FileObserver(File f){
    f.attach(this);
    _subject = f;
  }
  public void update(){
    System.out.println("file " +
     _subject.getAbsoluteName() + " has changed.");
  }
  private File _subject;
}
```

ATTACHING AND NOTIFYING OBSERVERS

```java
class File extends Node {
  File(String n, Directory p){
    super(n,p);
  }
  public void attach(Observer o){
    if (!_observers.contains(o)){
      _observers.add(o);
    }
  }
  public void detach(Observer o){
    _observers.remove(o);
  }
  public void notifyObservers(){
    for (Observer obs : _observers){
      obs.update();
    }
  }
  public void write(String s){
    notifyObservers();
  }

  private List<Observer> _observers = new ArrayList<Observer>();
}
```

UPDATED CLIENT

```java
public class Main {
  public static void main(String[] args){
    Directory root = new Directory("");
    File core = new File("core", root);

    // create observer for file core
    FileObserver obs = new FileObserver(core);
    core.write("hello");
    core.write("world");
  }
}
```

prints

```
file /core has changed.
file /core has changed.
```

OBSERVER: CONSIDERATIONS

- who triggers the update?
  - state-changing methods call notify() method, or
  - make clients responsible for calling notify()

- avoiding observer-specific update protocols
  - **push model**: subject sends its observers detailed information about the changes
  - **pull model**: subject only informs observers that state has changed; observers need to query subject to find out what has changed

- specifying modifications of interest explicitly
  - when observer is interested in only some of the state-changing events

- encapsulating complex update semantics
  - for highly complex relationships between subject and observer, introduce a ChangeManager class to coordinate

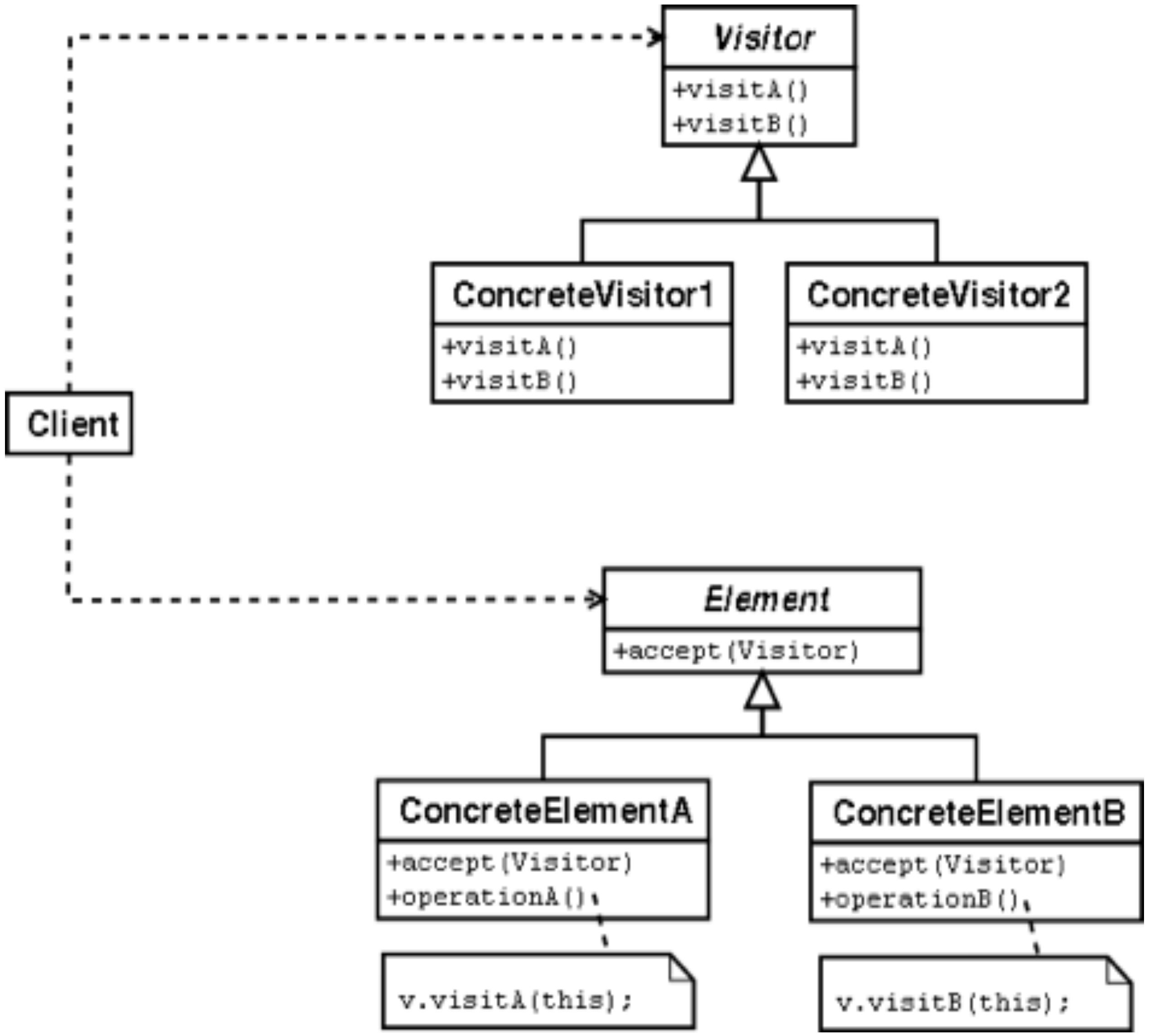
VISITOR

- represent an operation to be performed on a set of “related classes” without changing the classes.

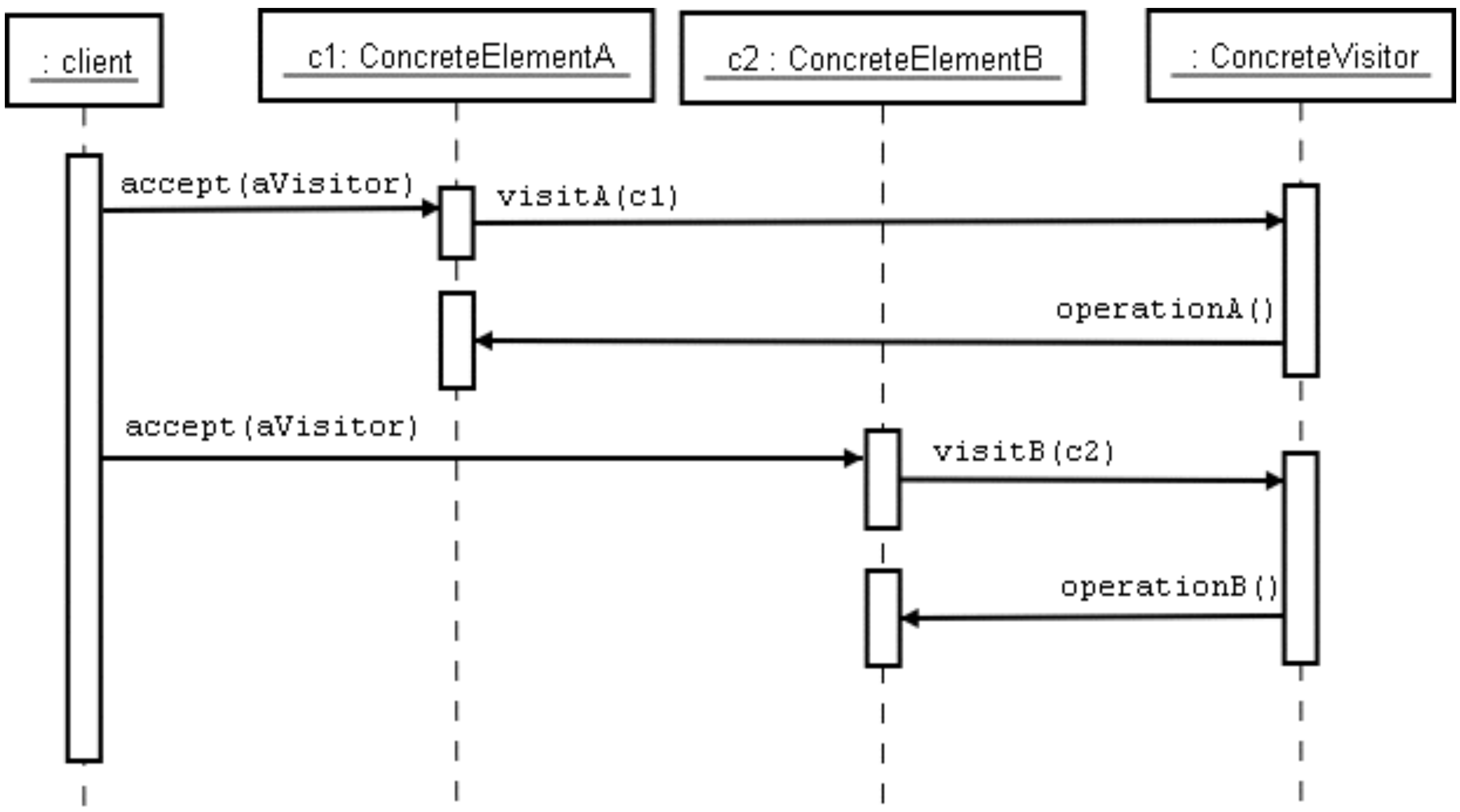
- apply **Visitor** when:
  - a hierarchy contains many classes with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes
  - many distinct and unrelated operations need to be performed on objects, and you want to avoid polluting their classes with these operations
  - the classes in the object structure rarely change, but you frequently want to add new operations on the structure

VISITOR: PARTICIPANTS

- **Visitor**
  - declares a **`visit()`** operation for each class of ConcreteElement in the object structure
- **ConcreteVisitor**
  - implements each operation declared by Visitor
- **Element**
  - defines an operation **`accept(Visitor)`**
- **ConcreteElement**
  - implements operation accept(Visitor)



VISITOR: SEQUENCE DIAGRAM



VISITOR: EXAMPLE

- final variation on the FileSystem example
  - based on solution with links, iterators
  - use Visitor to implement variant of Unix "**`du`**" command (**`du`** counts the size of a directory and its subdirectories, usually in 512-byte blocks)

- steps:
  - create interface **`Visitor`** with methods **`visit(File)`**, **`visit(Directory)`**, **`visit(Link)`**
  - create class **`DuVisitor`** that implements **`Visitor`**
  - declare **`accept(Visitor)`** method in class **`Node`**, implement in **`File`**, **`Directory`**, **`Link`**

STEP 1: ADDING ACCEPT() METHODS

```
class File extends Node {
  …
  public void accept(Visitor v){
    v.visit(this);
  }
  …
}

class Link extends Node {
  …
  public void accept(Visitor v){
    v.visit(this);
  }
  …
}
```

```
class Directory extends Node {
  …
  public void accept(Visitor v){
    v.visit(this);
  }
  …
}
```

STEP 2: DEFINE A VISITOR

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


class DuVisitor implements Visitor {
  DuVisitor(){
    _nrFiles = 0; _nrDirectories = 0;
    _nrLinks = 0; _totalSize = 0;
  }
  public void visit(File f){
    _nrFiles++;
    _totalSize += f.size();
  }
  public void visit(Link l){
    _nrLinks++;
  }
  …
```

```
…
public void visit(Directory d){
  _nrDirectories++;
  Iterator<Node> it = d.iterator();
  for (it.first(); !it.isDone(); it.next()){
    Node n = it.current();
    if (n instanceof File){
      visit((File)n);
    } else if (n instanceof Directory){
      visit((Directory)n);
    } else if (n instanceof Link){
      visit((Link)n);
    }
  }
}
public void report(){
  System.out.println("files:       " + _nrFiles);
  System.out.println("directories: " + _nrDirectories);
  System.out.println("links:       " + _nrLinks);
  System.out.println("total size:  " + _totalSize);
}
  int _totalSize; int _nrFiles; int _nrLinks; int
_nrDirectories;
}
```

CLIENT

```
public class Main {
  public static void main(String[] args){
    Directory root = new Directory("");
    new File("core", root, "hello");
    Directory usr = new Directory("usr", root);
    new File("adm", usr, "there");
    new Directory("foo", usr);
    new File("bar1", usr, "abcdef");
    new File("xbar2", usr, "abcdef");
    new File("yybarzz3", usr, "abcdef");
    Link link = new Link("link", usr, root);
    new Link("link2", link, root);

    DuVisitor visitor = new DuVisitor();
    root.accept(visitor);
    visitor.report();
  }
}
```

prints:

```
files:        5
directories:  3
links:        2
total size:   28
```

VISITOR: CONSIDERATIONS

- requires **ConcreteElement** classes to expose enough state so Visitor can do its job
  - breaks encapsulation

- adding new operations is easy
  - by defining new **ConcreteVisitor**

- adding new **ConcreteElement** classes is hard
  - gives rise to new abstract operation on **Visitor**
  - ...and requires implementation in every **ConcreteVisitor**

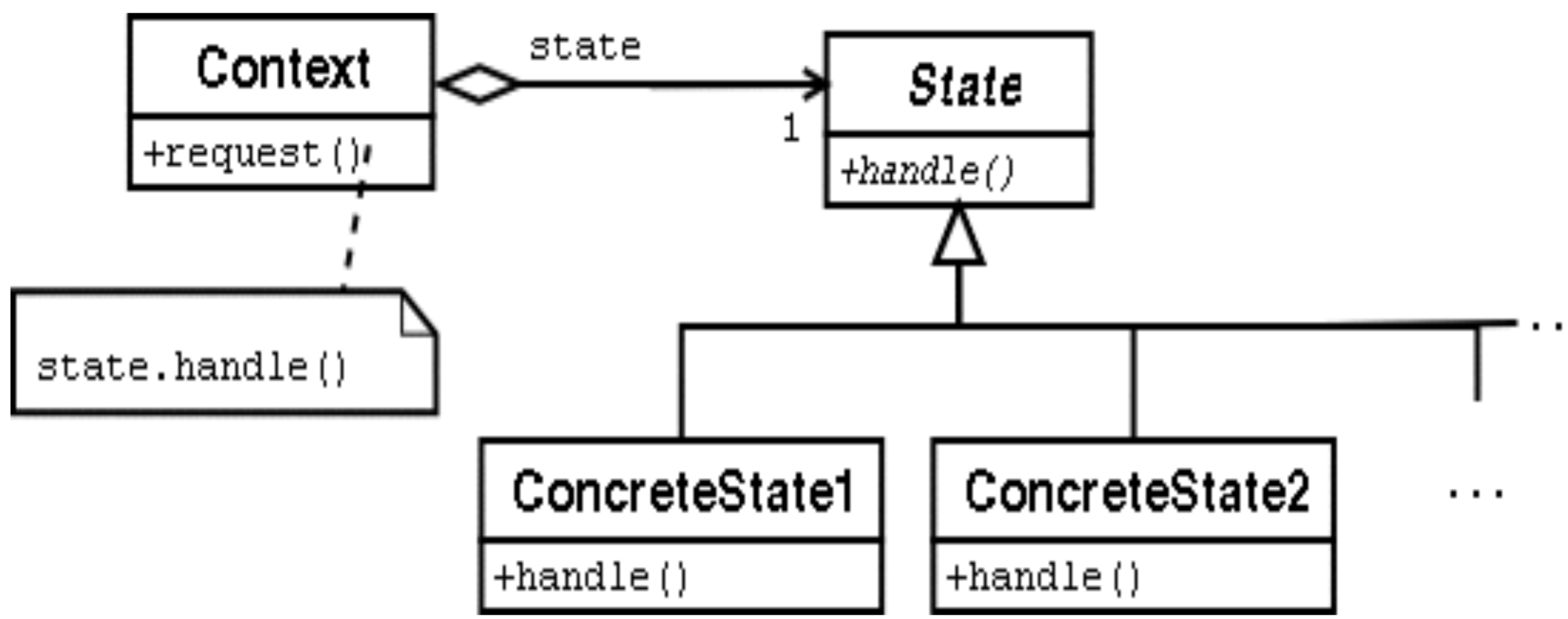
- Visitor not limited to a class hierarchy, can be applied to any collection of classes
  - provided they define **`accept()`** methods

STATE

- allow an object to change its behavior when its internal state changes
- use **State** when:
  - an object's behavior depends on its state
  - operations have large conditional statements that depend on the object's state (the state is usually represented by one or more enumerated constants)

STATE: PARTICIPANTS

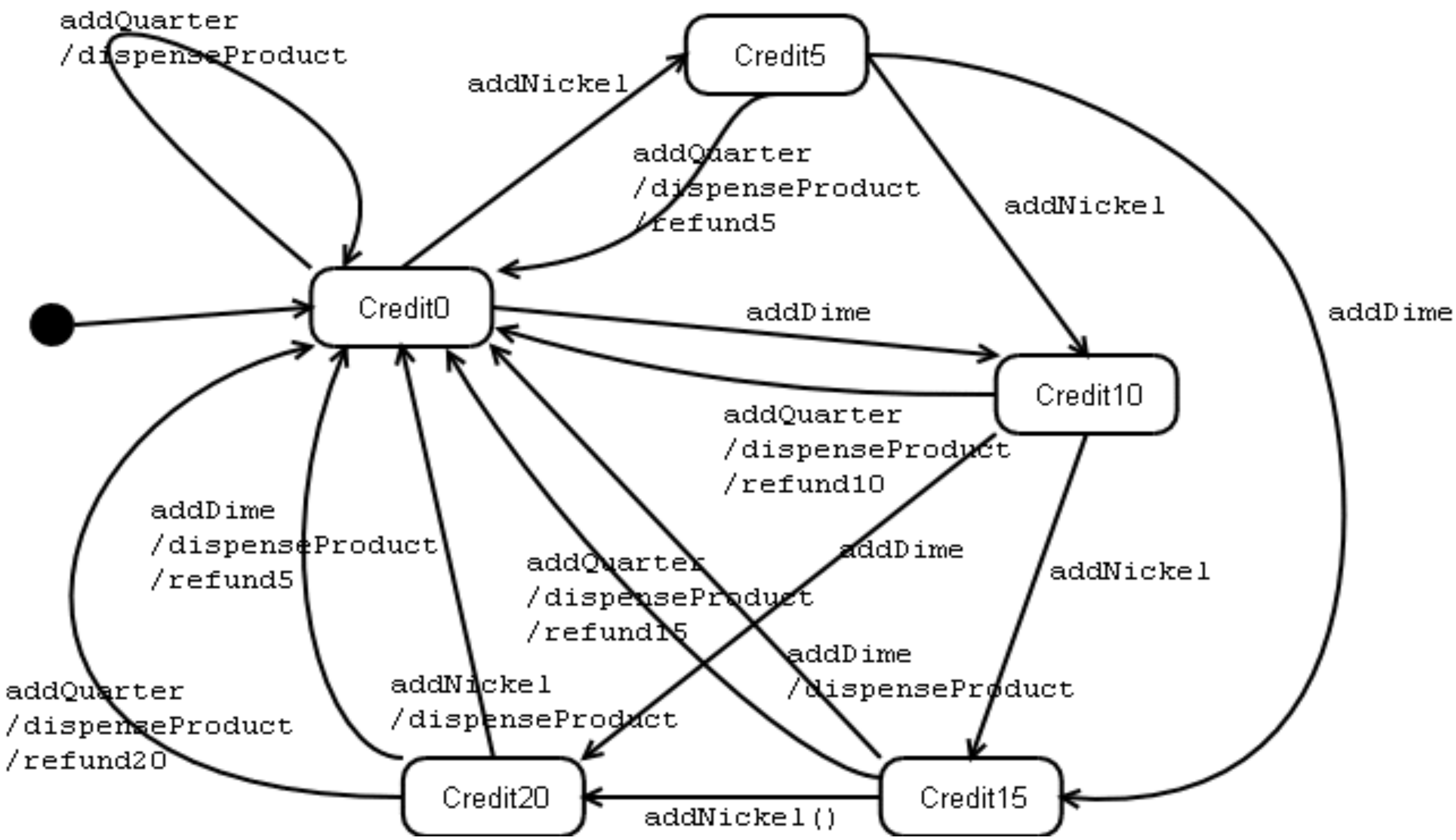
- **Context**
  - defines interface of interest to clients
  - maintains reference to a ConcreteState subclass that defines the **current** state
- **State**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context
- **ConcreteState** subclasses
  - each subclass implements a behavior associated with a state of the Context (by overriding methods in State)



STATE: EXAMPLE

- example of a **vending machine**:
  - product price is $0.25
  - machine accepts any combination of nickels, dimes, and quarters
  - customer enters coins; when credit reaches $0.25 product is dispensed, and refund is given for the remaining credit.
  - machine has display that shows the current balance

VENDING MACHINE: UML STATECHART DIAGRAM



“TRADITIONAL” IMPLEMENTATION

- use an integer value to represent the states
  - more complex situations may require an enum or object
- methods **`addNickel()`**, **`addDime()`**, and **`addQuarter()`** model user inserting coins
- methods **`refund()`**, **`displayBalance()`**, and **`dispenseProduct()`** model system’s actions
- conditional logic (with **`if`**/**`switch`** statements) depending on current state

“TRADITIONAL” IMPLEMENTATION

```
class VendingMachine {
  private int _balance;
  public VendingMachine() {
    _balance = 0; welcome();
  }
  void welcome() {
    System.out.println("Welcome.
    Please enter $0.25 to buy product.");
  }
  void dispenseProduct() {
    System.out.println("dispensing product...");
  }
  void displayBalance() {
    System.out.println("balance is now: " +
                       _balance);
  }
  void refund(int i) {
    System.out.println("refunding: " + i);
  }
  …
```

```
public void addNickel() {
  switch (_balance) {
  case 0 : { _balance = 5;
  displayBalance();
  break;  }
  case 5 : { _balance = 10;
  displayBalance();
  break; }
  case 10 : { _balance = 15;
  displayBalance();
  break; }
  case 15 : { _balance = 20;
  displayBalance();
  break; }
  case 20 : { dispenseProduct();
  _balance = 0; welcome();
  break; }
  }
}

…
```

“TRADITIONAL” IMPLEMENTATION (2)

```
…
public void addDime() {
  switch (_balance) {
  case 0 : { _balance = 10;
  displayBalance();
  break; }
  case 5 : { _balance = 15;
  displayBalance();
  break; }
  case 10 : { _balance = 20;
  displayBalance();
  break; }
  case 15 : { dispenseProduct();
  _balance = 0; welcome();
  break; }
  case 20 : { dispenseProduct();
  refund(5); _balance = 0; welcome();
  break; }
  }
}
…
```

```
…
public void addQuarter() {
  switch (_balance) {
  case 0 : { dispenseProduct();
  _balance = 0; welcome();
  break; }
  case 5 : { dispenseProduct();
  refund(5); _balance = 0; welcome();
  break; }
  case 10 : { dispenseProduct();
  refund(10); _balance = 0; welcome();
  break; }
  case 15 : { dispenseProduct();
  refund(15); _balance = 0; welcome();
  break; }
  case 20 : { dispenseProduct();
  refund(20); _balance = 0; welcome();
  break; }
  }
}
}
```

CLIENT CODE

```java
public class Client {
  public static void main(String[] args) {
    VendingMachine v = new VendingMachine();
    v.addNickel();
    v.addDime();
    v.addNickel();
    v.addQuarter();
  }
}
```

```
Welcome. Please enter $0.25 to buy product.
balance is now: 5
balance is now: 15
balance is now: 20
dispensing product...
refunding: 20
Welcome. Please enter $0.25 to buy product.
```

PROBLEMS WITH THIS CODE

- state-specific behavior *scattered* over different conditionals
  - changing one state's behavior requires visiting each of these
- inflexible: adding a state requires invasive change
  - would need to edit each conditional
- approach tends to lead to large monolithic classes

STATE-BASED VENDINGMACHINE

```java
interface VendingMachineState {
  void addNickel(VendingMachine v);
  void addDime(VendingMachine v);
  void addQuarter(VendingMachine v);
  int getBalance();
}


public class VendingMachine {
  public VendingMachine() {
    _state = Credit0.instance(this);
  }
  // methods welcome(), displayBalance() etc. as before

  void changeState(VendingMachineState s) {
    _state = s; displayBalance();
  }
  public void addNickel() { _state.addNickel(this); }
  public void addDime() { _state.addDime(this); }
  public void addQuarter() { _state.addQuarter(this); }
  private VendingMachineState _state;
}
```

CONCRETE STATE CLASSES

```
class Credit0 implements VendingMachineState {
  private Credit0(){ }
  private static Credit0 _theInstance;
  static Credit0 instance(VendingMachine v) {
    if (_theInstance == null) {
      _theInstance = new Credit0();
    }
    v.welcome(); return _theInstance;
  }
  public void addNickel(VendingMachine v) {
    v.changeState(Credit5.instance());    }
  public void addDime(VendingMachine v) {
    v.changeState(Credit10.instance());   }
  public void addQuarter(VendingMachine v) {
    v.dispenseProduct();
    v.changeState(Credit0.instance(v));   }
  public int getBalance() { return 0; }
}
```

```
class Credit20 implements VendingMachineState {
  private Credit20(){ }
  private static Credit20 _theInstance;
  static Credit20 instance(){
    if (_theInstance == null){
      _theInstance = new Credit20();
    }
    return _theInstance;
  }
  public void addNickel(VendingMachine v) {
    v.dispenseProduct();
    v.changeState(Credit0.instance(v));     }
  public void addDime(VendingMachine v) {
    v.dispenseProduct(); v.refund(5);
    v.changeState(Credit0.instance(v)); }
  public void addQuarter(VendingMachine v) {
    v.dispenseProduct(); v.refund(20);
    v.changeState(Credit0.instance(v));   }
  public int getBalance(){ return 20; }
}
```

STATE: BENEFITS

- **localizes state-specific behavior**, and partitions behavior for different states
  - leads to several small classes instead of one large class
  - natural way of partitioning the code
- **avoids (long) if/switch statements** with state-specific control flow
  - also more extensible---you don't have to edit your switch statements after adding a new state
- **makes state transitions explicit**
  - simply create a new ConcreteState object, and assign it to the state field in Context
- state-objects can be shared
  - and common functionality can be placed in abstract class State

STATE: IMPLEMENTATION ISSUES

- who defines the state transitions?
  - not defined by the pattern
  - usually done by the various ConcreteStates
- when to create ConcreteStates?
  - on demand or ahead-of-time
  - choice depends on how often ConcreteStates get created, and cost of creating them
- can use Singleton if ConcreteStates don’t have any fields

OTHER BEHAVIORAL PATTERNS

Chain of Responsibility	avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
Interpreter	given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language
Mediator	define an object that encapsulates how objects interact
Memento	without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
Strategy	define a family of algorithms, encapsulate each one, and make them interchangeable
Template Method	define the skeleton of an algorithm in an operation, deferring some steps to subclasses

DESIGN PATTERNS: GENERAL REMARKS

- design patterns are not the solution to all problems!

- in general, don't try to apply as many patterns as possible. Instead, try to:
  - recognize situations where patterns are useful
  - use key patterns to define global system architecture

- document your use of patterns, use names that reflect participants in patterns

- reusable software often has to be refactored
  - design patterns are often the "target" of refactorings that aim at making the system more reusable
  - next week: more about refactoring…