Object Oriented Programming and Design in Java

Session 21
Instructor: Bert Huang

Announcements

- Homework 4 due now
- Homework 5 out now. Due last day of class: Mon. May 3rd
- Mon. May 3rd: Final review
- Mon. May 10th, Final exam. 9 AM - noon
  - closed-book/notes, focus on post-midterm material, but material is inherently cumulative

Review

- Applications of queues, stacks, maps, sets
  - Queues/stacks: producer/consumer, method calls
  - Maps and Sets: word search, word count
- Binary search trees:
  - SortedMap, SortedSet interfaces
  - O(log N) for add/get, fast range search
- Priority Queues (Heaps)
  - O(1) findMin, O(log N) insert and deleteMin

Today's Plan

- Threadsafe wrappers for Collections
- Leftover Design Patterns
    - ADAPTER
    - COMMAND
    - FACTORY METHOD
    - PROXY
    - SINGLETON
    - VISITOR

Comparison

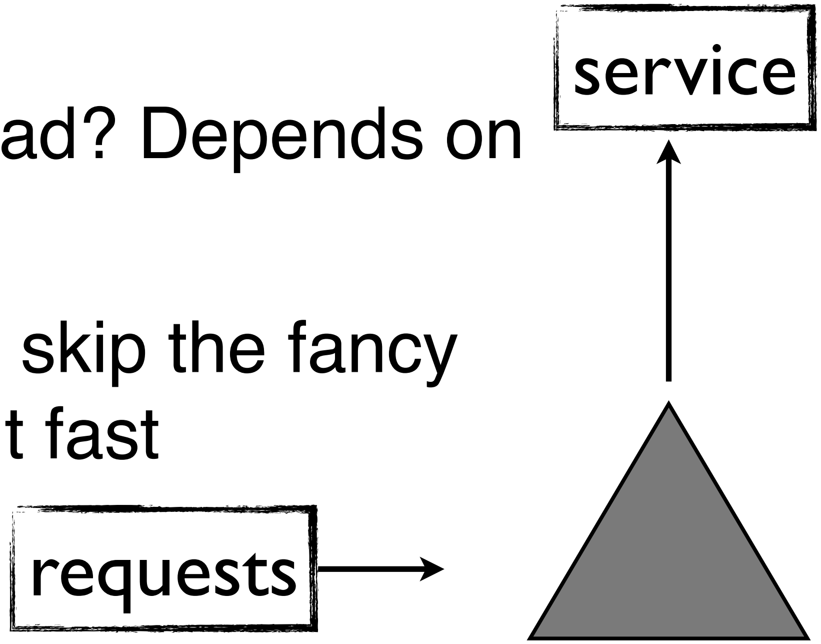
	insert	findMin	get	get range
lists	O(1)	O(N)	O(N)	O(N)
hashmap	O(1)	O(N)	O(1)	O(N)

BST	O(log N)	O(log N)	O(log N)	O(log N + k)
				k = # elements in range

heap	O(log N)	O(1)	O(N)	O(N)

Producer Consumer with Priority Queues

- Natural extension to using a simple queue, assign priority to all requests
- Consumer grabs the highest (lowest) priority element
- Is it worth the log N overhead? Depends on application
  - If consuming is very fast, skip the fancy prioritization and just do it fast



Thread Safe Data Structures

- Since data structures are designed to be extremely fast, thread safety is omitted to avoid overhead
- Java has interface ConcurrentMap, implemented by ConcurrentHashMap
- and interface BlockingQueue, implemented by ArrayBlockingQueue, LinkedBlockingQueue

Threadsafe Wrappers

- Collections has static method
  `Collection synchronizedCollection(Collection c)`
  - returns synchronized wrapper of c
- synchronizedSet, List, Map, SortedMap
- Returns *decorated* object of anonymous class
- Each unsafe method is wrapped with an object lock

Programming Patterns

MVC

SINGLETON

FACTORY-METHOD

VISITOR

COMPOSITE

PROXY

DECORATOR

ADAPTER

COMMAND

STRATEGY

TEMPLATE-METHOD

Pattern: Adapter

- When reusing code, we often find interfaces that do the same thing
- Maybe uses different method names, parameter order, etc
- Don't rewrite any concrete classes, create an adapter
  - implement one interface using the other

ADAPTER

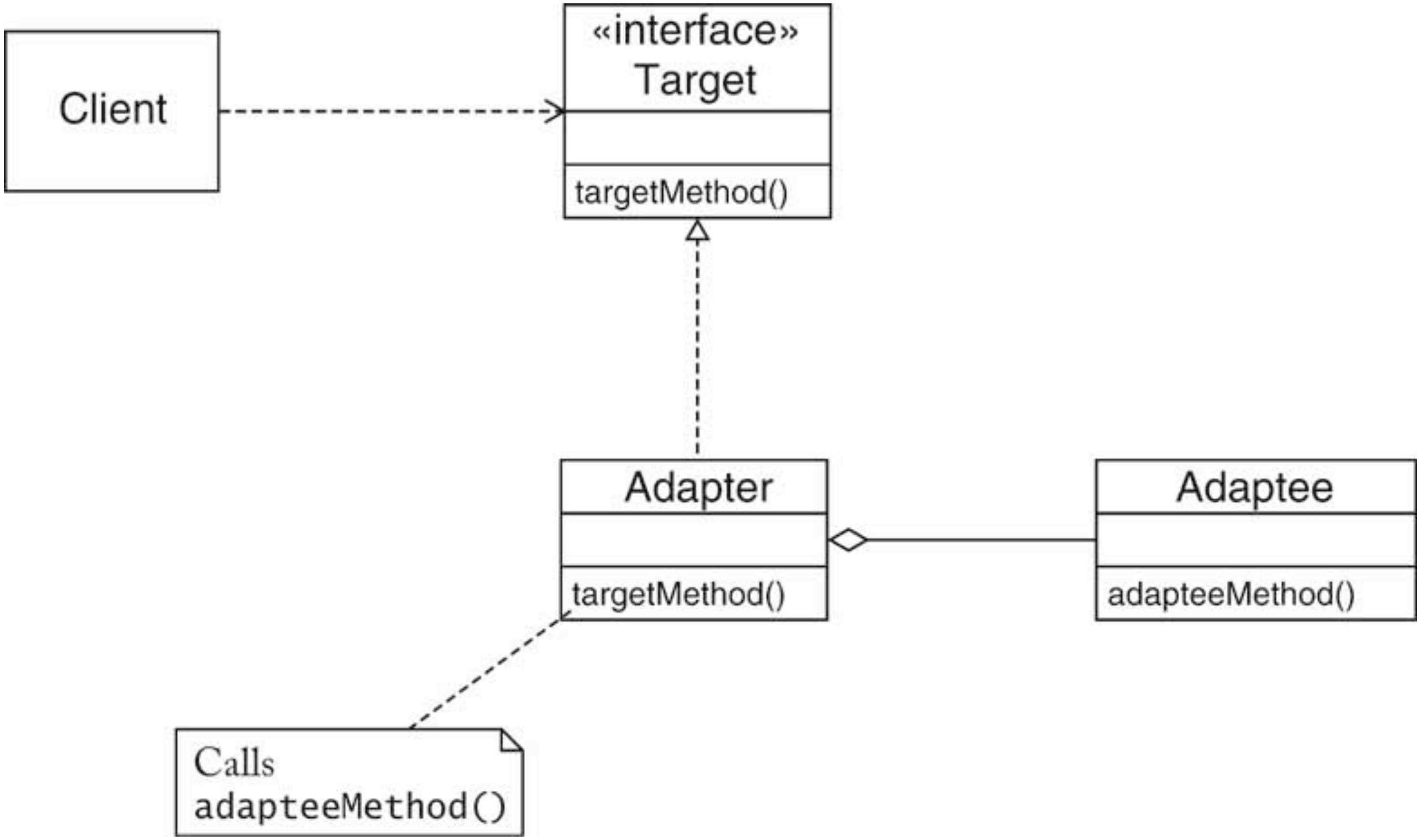
Context

- You want to use an existing *adaptee* class without modifying it.
- The context in which you want to use the class requires conformance to a *target* interface
- The target interface and the adaptee interface are conceptually related

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

Adapter Diagram

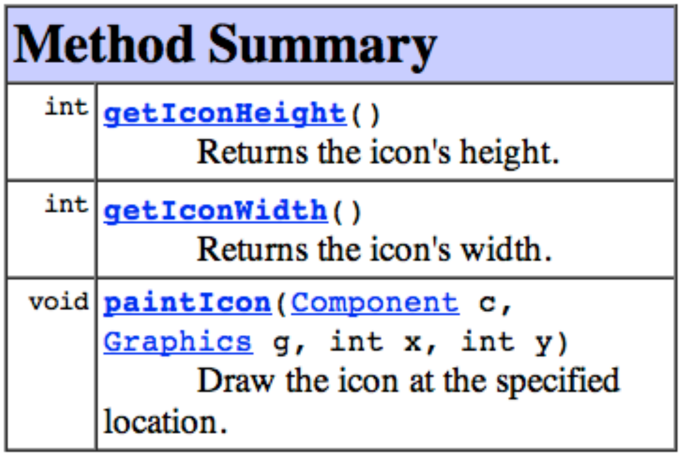


```
/**
   An adapter that turns an icon into a JComponent.
*/
public class IconAdapter extends JComponent {
   /**
      Constructs a JComponent that displays a given icon.
      @param icon the icon to display
   */
   public IconAdapter(Icon icon) {
      this.icon = icon;
   }

   public void paintComponent(Graphics g) {
      icon.paintIcon(this, g, 0, 0);
   }

   public Dimension getPreferredSize() {
      return new Dimension(icon.getIconWidth(),
            icon.getIconHeight());
   }

   private Icon icon;
}
```



Method Summary

int	getIconHeight() Returns the icon's height.
int	getIconWidth() Returns the icon's width.
void	paintIcon(Component c, Graphics g, int x, int y) Draw the icon at the specified location.

Pattern: Command

- It is sometimes useful to be able to manipulate commands as objects
  - command history, undo, macros, etc.
  - states for commands, e.g., estimated-duration, Icon for GUI, etc.
- Executing commands by just calling methods does not allow us to do these

COMMAND

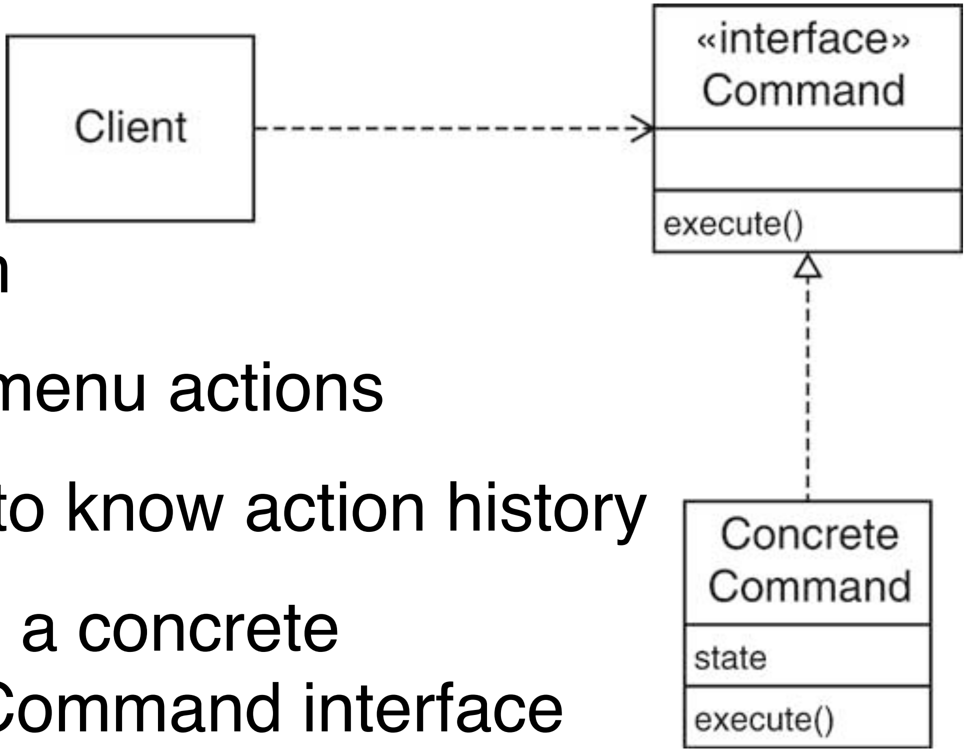
Context

- You want to implement commands that behave like objects, either because
  - you want to store additional information with commands,
  - or you want to collect commands

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

Command Example



- Client: painting program
- User performs various menu actions
- Multi-level undo needs to know action history
  - Each type of action is a concrete implementation of a Command interface
  - Each action also implements an undo() method
- Client program stores stack of commands; pop().undo() to undo most recent command

Pattern: Factory Method

- `list.iterator()` returns an Iterator object
- If we know concrete class of list, could use `Iterator iter = new LinkedListIterator(list)`
- but that's not polymorphic; client shouldn't need to know concrete classes
- The `iterator()` method is a *factory method*

FACTORY-METHOD

Context

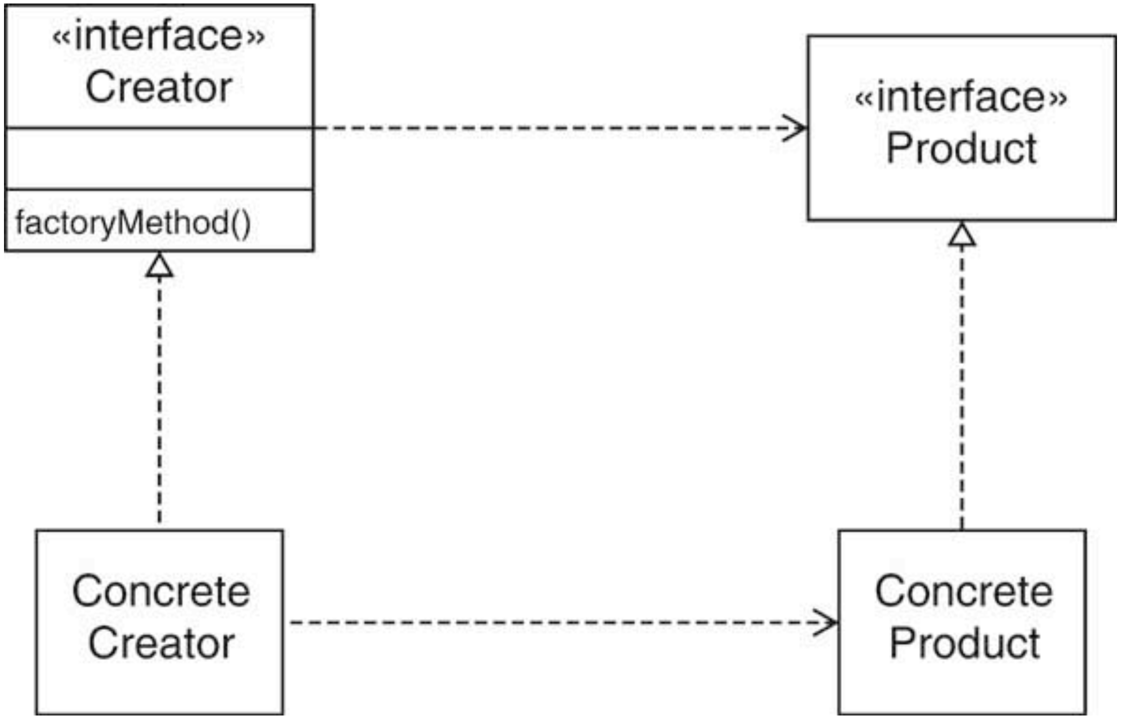
- A creator type creates objects of another *product* type
- Subclasses of the creator type need to create different kinds of product objects
- Clients do not need to know the exact type of product objects

---

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 *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

Example Factory-Method



- Creator: Collection
- Concrete Creator: LinkedList
- factoryMethod(): iterator()
- Product: Iterator
- ConcreteProduct: LinkedListIterator

Pattern: Proxy

- A proxy acts on behalf of someone else
- In the proxy pattern, an object represents another object,
- is treated exactly as the represented object
- but modifies the under-the-hood behavior in some way
- A Proxy is like a Decorator you never notice
- e.g., threadsafe wrappers could use the Proxy pattern

PROXY

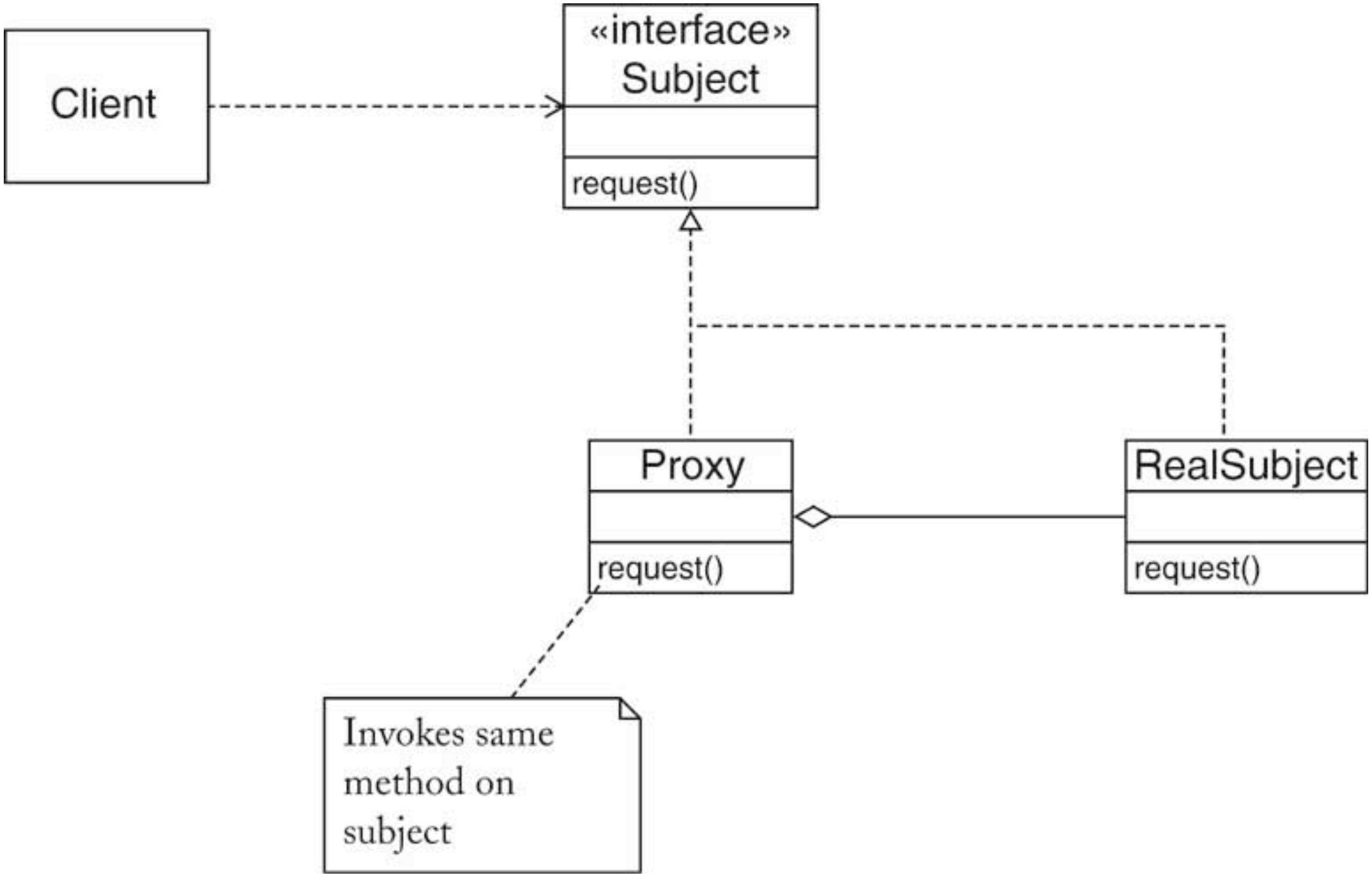
Context

- A *real subject class* provides a service that is specified by an *subject interface* type
- There is a need to modify the service in order to make it more versatile
- Neither the client nor the real subject should be affected by the modification

Solution

- Define a *proxy* class that implements the subject interface type. The proxy holds a reference to the real subject
- The client uses a proxy object
- Each proxy method invokes the same method on the real subject and provides the necessary modifications

Proxy Diagram



Proxy Example

- Normally, you can add an Icon to a Label
  JLabel label = new JLabel(new ImageIcon(imageName))
  - loads the image on construction, may waste memory/time
- Use proxy instead: label = new JLabel(new ImageProxy(imageName))
- ImageProxy doesn't load the image until it is painted

```
public void paintIcon(Component c, Graphics g, int x, int y)
{
   if (image == null) image = new ImageIcon(name);
   image.paintIcon(c, g, x, y);
}
```

Pattern: Singleton

- We often have classes that never need more than one instance
  - e.g., a utility class that everyone shares
- One approach is to have the class have only static methods,
  - but a static class can't implement an interface, can't be passed as a parameter

SINGLETON

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

---

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

Example Singleton

- Pseudo-random number generators
- I often find my code riddled with redundant Random objects;
  I really only need one

```
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();
}
```

Pattern: Visitor

- You're building a hierarchy of classes, and you want to allow new functionality
- but don't want to have clients modify code
- STRATEGY is inadequate if new functionality depends on concrete types
- e.g., file system: DirectoryNode and FileNode
  - want to allow client to add operations, e.g., printing operation, disk-space computation

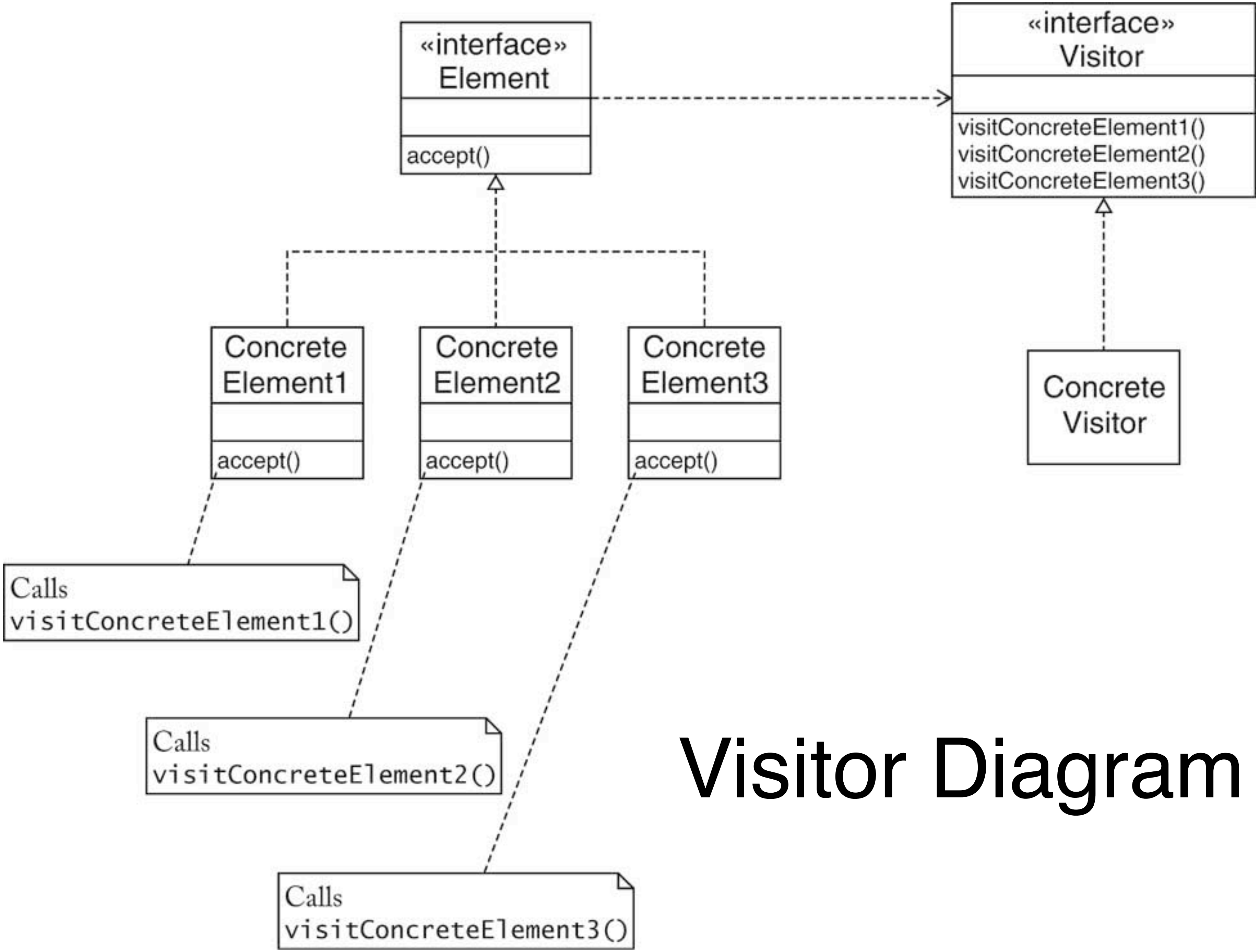
VISITOR

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

Solution

- Define a *visitor* interface 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

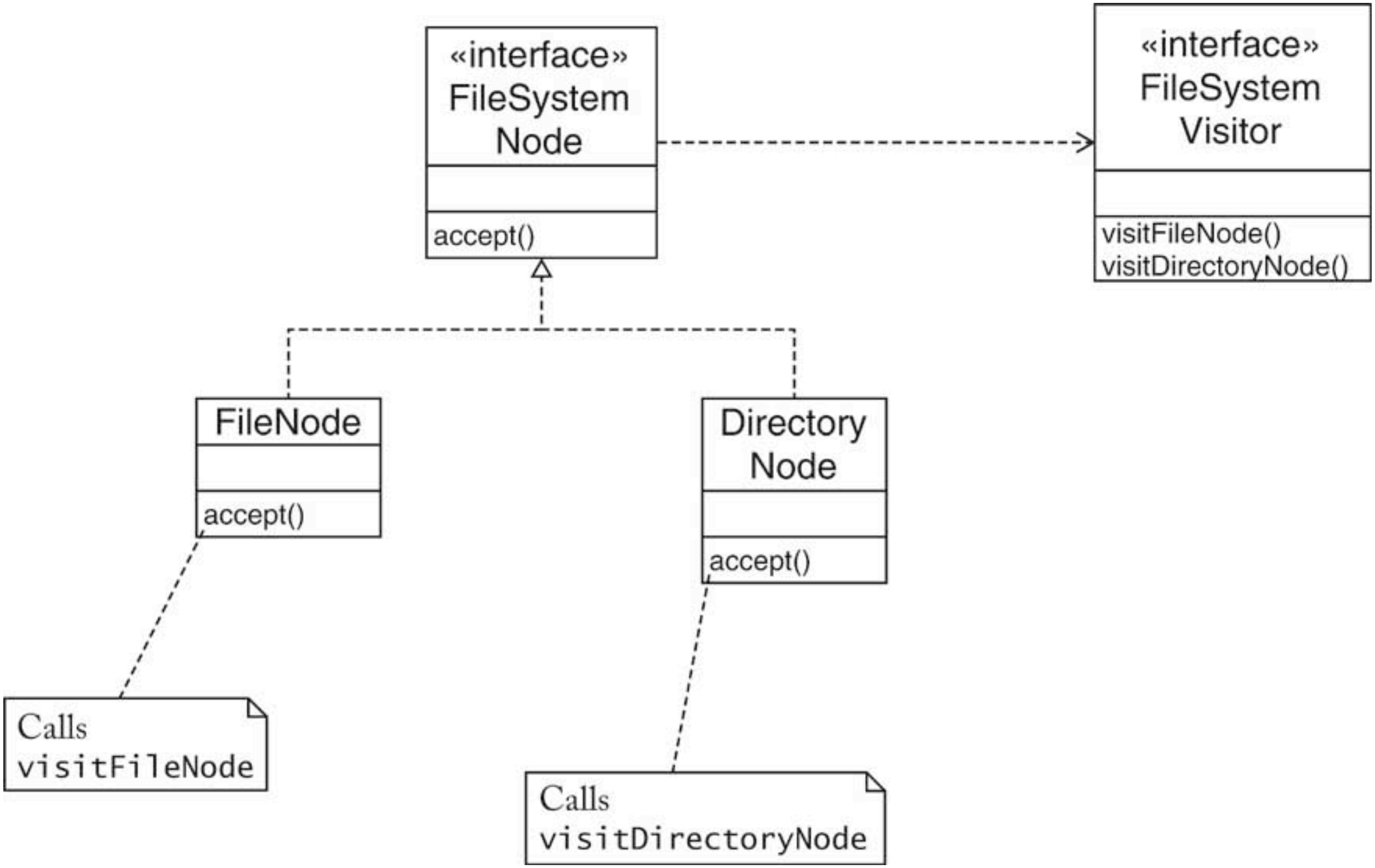


Visitor Diagram

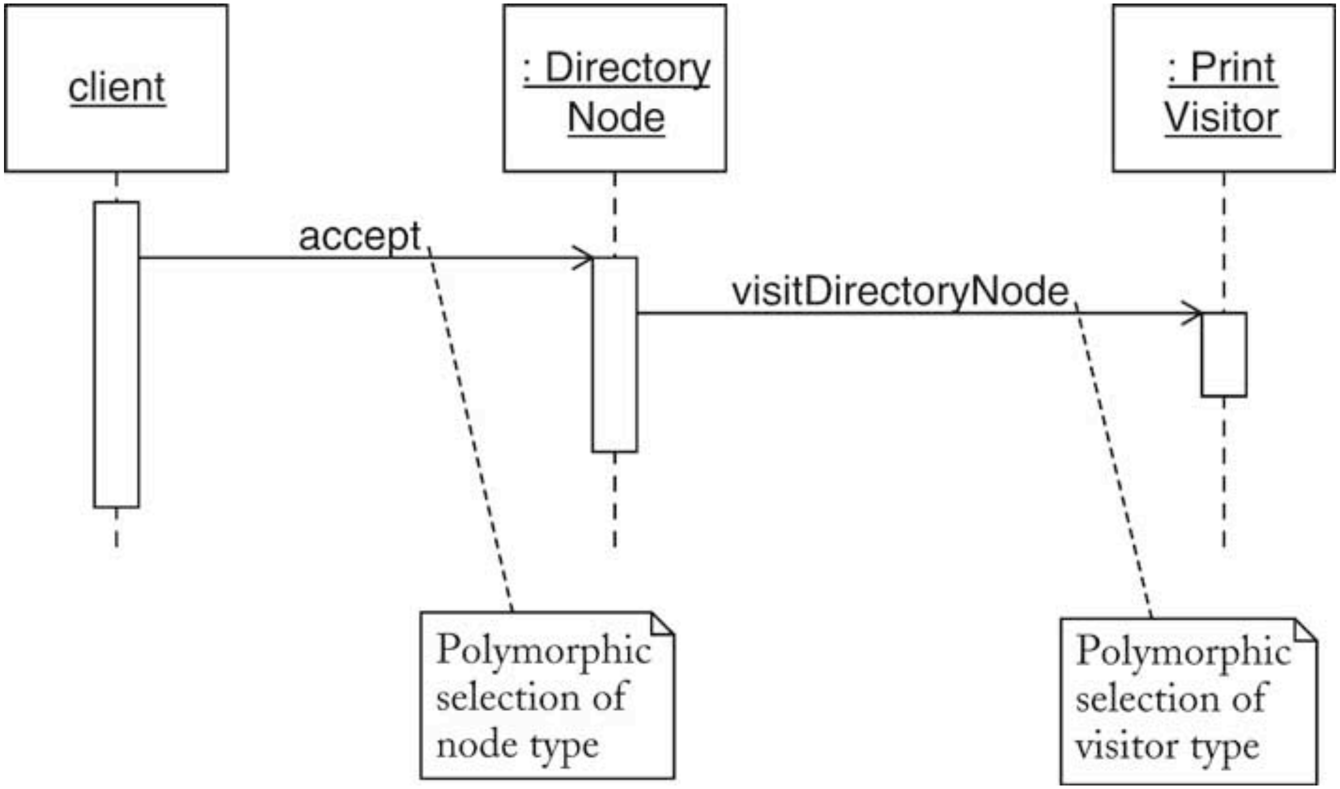
Double Dispatch

- This pattern uses polymorphism twice to make code very general
  - 1st, element.accept() calls Visitor method based on type of element
  - 2nd, the Visitor method performs operation based on type of Visitor
- Both actions called through interfaces
- Concrete classes need not be known at runtime

Example Visitor



Double Dispatch in FileSystemNode


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

Programming Patterns

MVC

VISITOR

COMPOSITE

PROXY

DECORATOR

ADAPTER

SINGLETON

COMMAND

STRATEGY

FACTORY-METHOD

TEMPLATE-METHOD

Reading

- Horstmann Ch. 10