Object – Oriented Design with UML and Java

Part XI: Design Patterns

Version 2023

Pattern Roots

A Pattern Language - Towns - Buildings - Construction

By Christopher Alexander <ISBN 0-19-501919-9>

- Patterns combine (one flushes out the details of another). Construction patterns are used within buildings, within neighborhoods, within towns…
- When a collection of patterns becomes rich enough to generate solutions across an entire domain, it may be referred to as a ***Pattern Language***.
- The goal is to guide the designer toward architectures which are both functional and aesthetically pleasing, for humans.
- A ***Design Pattern*** is a practical (not necessarily novel) solution to a recurring problem, occurring in different contexts, often balancing competing forces.

Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software

By the GoF: Gamma, Helm, Johnson, Vlissides <ISBN 0-201-63361-2>

– Best-selling Computer Science book of all time.

Design Patterns…

- Provide “tried and true” design solutions.
- Are examples of excellent OO design.
- Transfer design expertise from master practitioners.
- Reduce discovery costs and design complexity.
- Facilitate thinking and communicating about OO designs.

The 23 GoF Patterns, Categorized

- **Structural** (the composition of classes or objects):
  - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
- **Behavioral** (object interactions and responsibilities):
  - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor
- **Creational** (object creation):
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton

Design Pattern Templates

- The ***intent*** of the pattern.
- A motivating ***example***.
- General description of the ***problem***.
- A description of the ***forces*** which make the context challenging.
- A solution, with a ***name***.
- Where and how to apply the pattern.
- Benefits, trade-offs, consequences and costs.
- Implementation issues.

Software Forces

Forces are concerns, goals, constraints, trade-offs, and motivating factors.

Good designs balance competing forces with minimal complexity.

Some example forces that apply to software design:

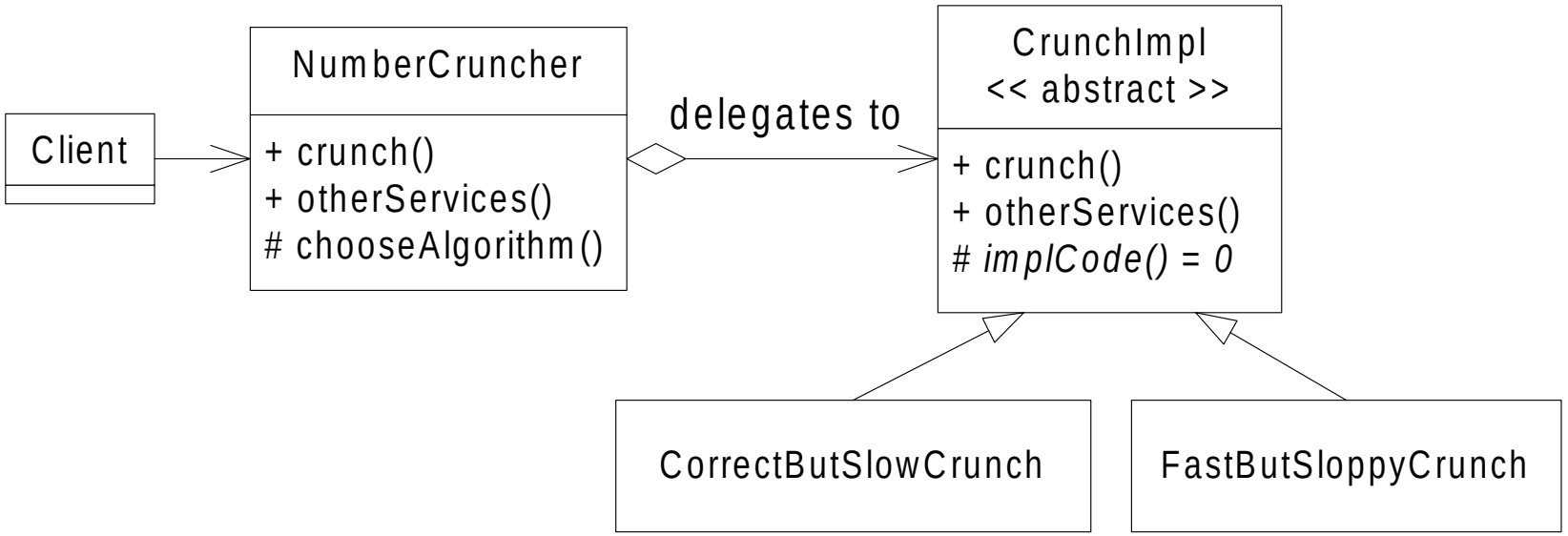
- Memory usage minimized while performance maximized.
- Simple, yet flexible.
- The code must never (never!) crash. OK, let it crash from time to time, but implement a robust disaster recovery plan.
- The server must support 1000 simultaneous connections.
- Initial time-to-market versus long-term quality.
- Faster? Or more reliable?
- Design a system that scales to 1,000,000,000 users...

Patterns in Software

- **Architectures**:
  - Layered Subsystems, MVC, Thin Client, SOA, Inversion of Control, Message Queue, Enterprise Service Bus, Load-Balanced Server, Multi-Zoned Network, . . .
- **Idioms**: Constructs specific to a programming language.
- **Anti-Patterns**:
  - Spaghetti Code, Big Ball of Mud, Analysis Paralysis, Toss it Over the Wall, Mythical Man Month, Death March, Hacking, Marketecture, Business Logic on UI Forms, . . .
- **Management Process**:
  - Waterfall, Agile, Hold Reviews, Engage Users, Engage QA, . . .

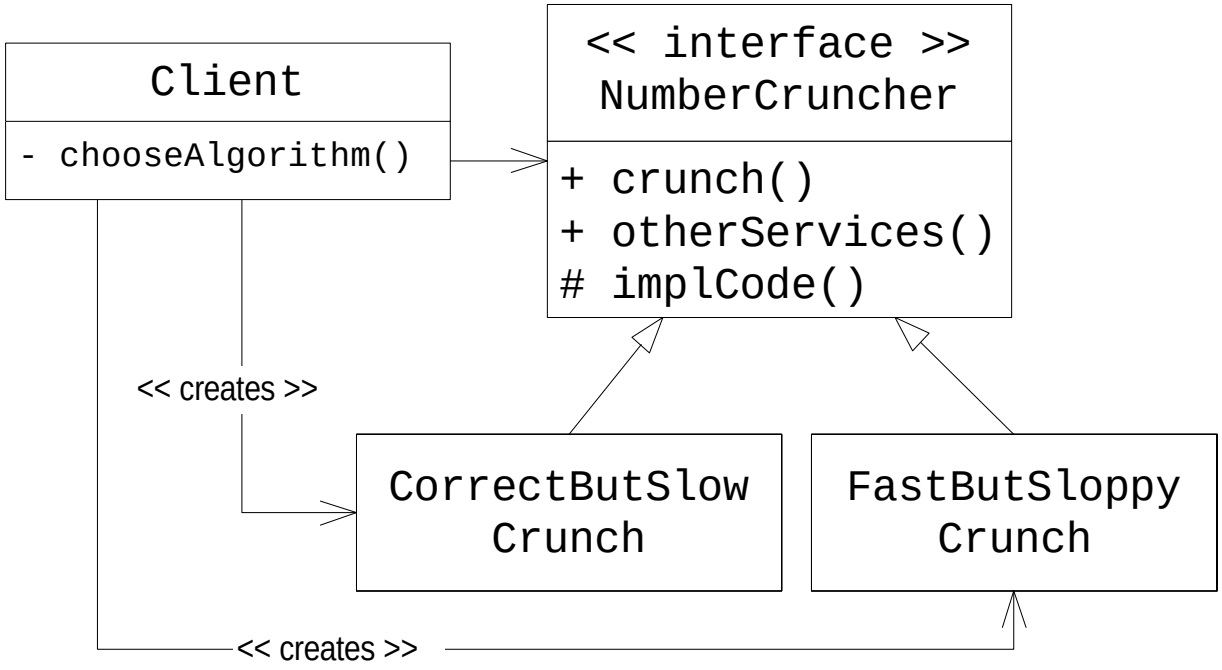
Design Pattern: *Strategy*

Intent: Allows multiple implementation strategies to be interchangeable, so that they can easily be swapped at run-time, and so that new strategies can be easily added.



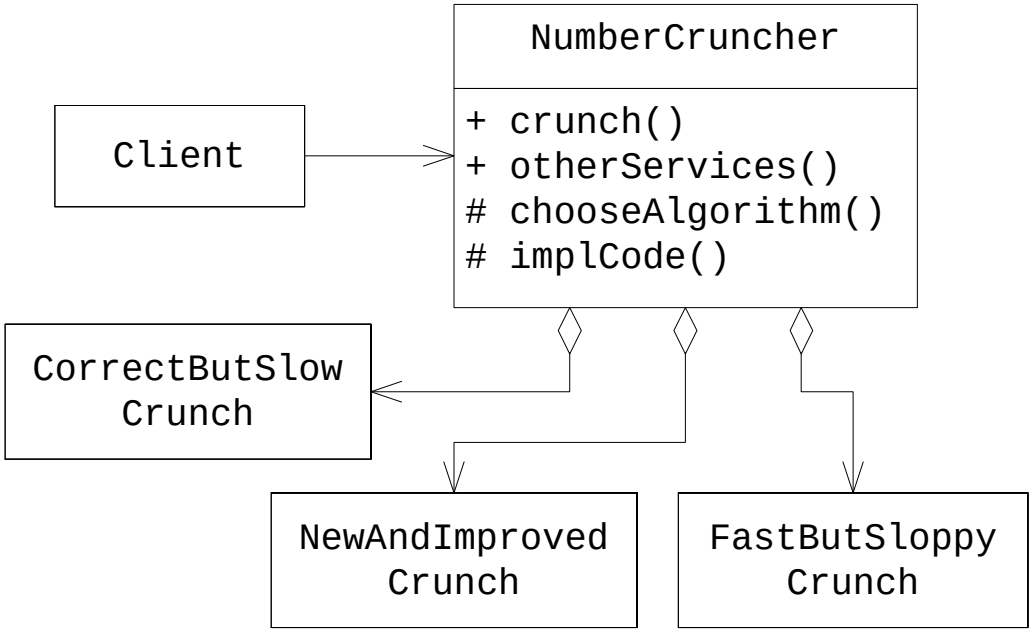
Strategy Alternative

- What if there were not a CrunchAlgorithm interface… suppose instead that NumberCruncher had two subclasses, CorrectButSlowNumberCruncher, and FastButSloppyNumberCruncher…? Why is this bad?



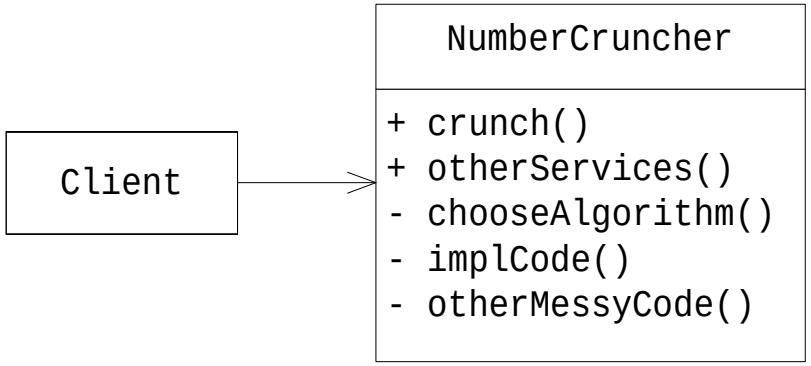
Another *Strategy* Alternative

Here's another "correct" design... Adding a NewAndImprovedCrunch would require adding *if-then-else* logic everywhere that the different Crunches are used. If the *Strategy* pattern were applied instead, the only place where the concrete CrunchImpls would get referred to specifically is the one place that they get instantiated.



Another *Strategy* Alternative

- All the NumberCruncher code is in one big class… Why is this bad?



- *Strategy* is similar to *Bridge*; same basic structure; very different intent.
- The *Strategy* pattern is also similar to *State*, which allows a class to be configured with different behaviors from which it can select whenever it makes a state transition.
- All 3 design patterns use “delegation to an abstract class or interface.”
- The difference lies in the patterns’ *intents*...

Cost vs. *time* vs. *convenience*…

Q: How should I travel to work tomorrow?

1: Bicycle?

2: Bus?

3: Car?

4: Taxi?

5: Friend?

6. Hitch-hike?

7. Walk?

Another example:

The Fractal Applet's FastColorsCalculator requires extra memory usage for every drawing.

Alternatives to *State* and *Strategy*

Many “procedural programmers” tend toward designs with lots of “decisional logic” - using “switch statements.”

Problems with this approach:

Increased code complexity.

Changes require multiple edits to the multiple switch statements.

Increases decisional logic, and thus, the likelihood for the code to have bugs - polymorphism can be used instead.

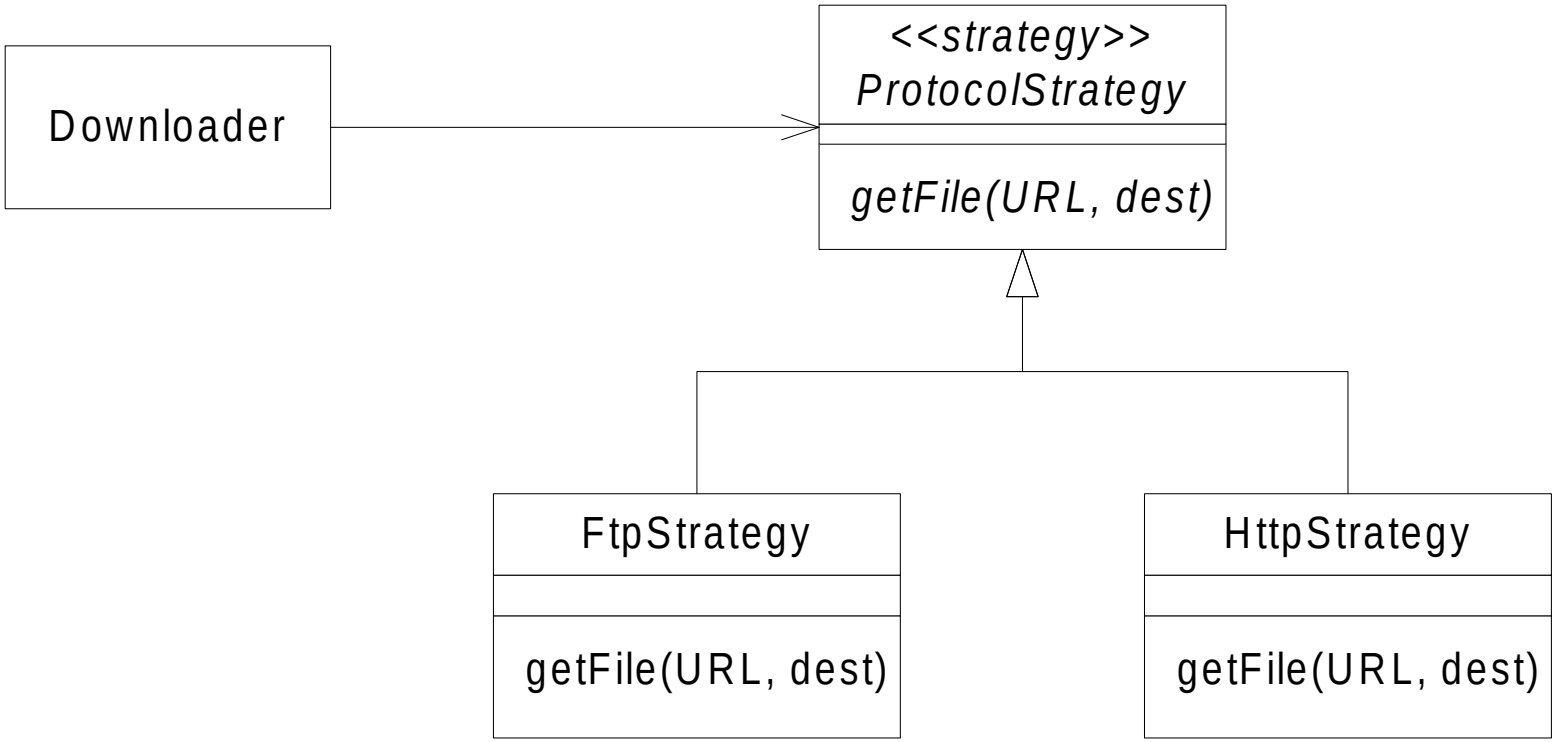
Tables are hard to understand by inspection.

The code ends up with if - else if - else if - else if . . .

Example: A File Downloader . . .

- We need a piece of software that will download files
  - We know of two protocols that must be supported, HTTP & FTP.
  - Other protocols will be used in the future
  - ***Let the design evolve…***
- Considerations:
  - We would like to encapsulate as much as possible regarding the handling of the protocol.
  - We would like to be able to switch protocols at run-time, depending on the download server.

Use the *Strategy* Pattern



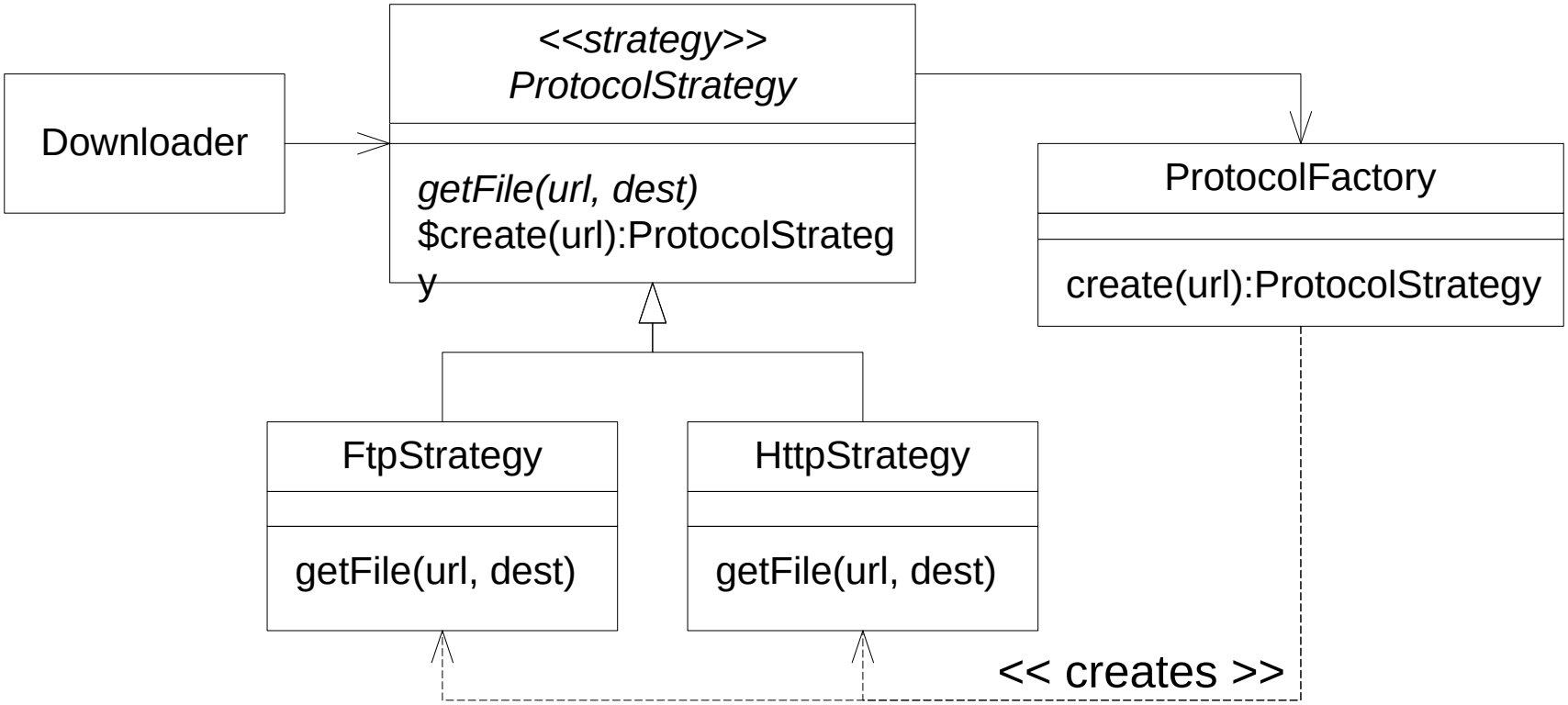
Using the Strategy

```
class Downloader {
    void download( String url, String dest )
    throws Exception {
        ProtocolStrategy ps;
        url = url.toLowerCase();
        if( url.startsWith( “ftp” ) ) {
            ps = new FtpStrategy();
        }
        else if( url.startsWith( “http” ) ) {
            ps = new HttpStrategy();
        }
        else {
            throw new Exception( “No can do” );
        }
        ps.getFile( url, dest );
    }
}
```

More Considerations

- Every time we add a protocol strategy, we have to modify `Downloader`.
- We would prefer that `Downloader` only has to know about `ProtocolStrategy`, remaining ignorant of the various concrete implementation classes.
- The `ProtocolStrategy` generalization breaks down at the point where new objects are instantiated.

Use a *Factory*

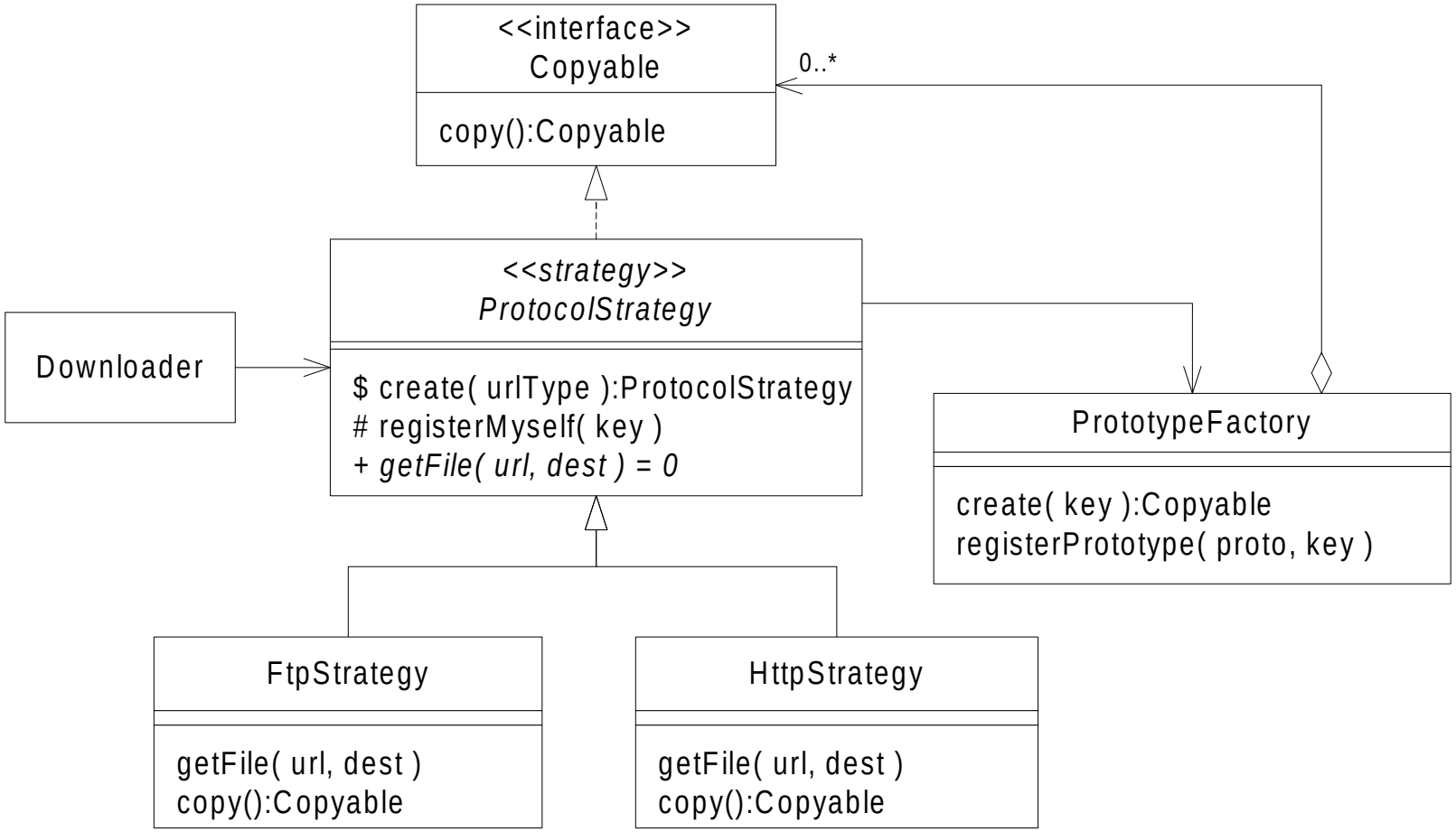


More Considerations

- We would like the system to be able to run 24x7 with the ability to add new strategies dynamically. To do this, we need to get away from specifying concrete class names.
- Note that the example code (in Java) assumes a mechanism to tell the running application to load a new class. There are various ways to do that.
- We want new strategies to register themselves:

```
class ShttpStrategy extends ProtocolStrategy {
  static { // This runs once upon loading the class
    ProtocolStrategy ps = new ShttpStrategy();
    ps.registerMyself( “shttp” );
  } // ...
}
```

Use *Prototypes*



Inside the Factory

```
class PrototypeFactory {
  private HashMap protoMap = new HashMap();

  public void registerPrototype( Copyable proto, String key ){
   protoMap.put( key.toLowerCase(), proto );
  }

  public Copyable create( String key )
  throws UnknownPrototypeKeyException {
    try {
      key = key.toLowerCase();
      Copyable proto = (Copyable) protoMap.get( key );
      return proto.copy();
    }
    catch( Throwable t ) {
      throw new UnknownPrototypeKeyException();
} } }
```

One Final Consideration

The ***Dependency Injection*** pattern can be a better choice than a ***Factory*** pattern

- Less code that is easier to configure and test
- Check out the ***Spring*** and ***Google Guice*** frameworks

The framework is responsible for creating the correct **`ProtocolStrategy`**.

- Annotate a constructor with a dependency
- Framework are often configured with XML files
- Notice how easy it would be to add a **`TestStrategy`**

```
@Inject // Guise
Downloader( ProtocolStrategy ps ) {
  this.ps = ps;
} // Easy, isn't it?
```

Why use *Creational* Design Patterns ?

- To separate concerns.
- To eliminate the use of hard-coded concrete class names when we want our code to use a generalization.
- To allow a program to be configurable to use “families” of concrete classes (as in the next example).
- Perhaps to recycle a retired instance using the “object pool” pattern?
- Or to reconstruct an object from a dormant state.
- Or to let a subclass decide which object to create.
- Or to encapsulate a complex construction process.
- Or to write less code that is easier to test.

Abstract Factory example

Design a portable GUI toolkit in C++ with Buttons, TextFields, etc... The code has to run on a Mac, Windows and Unix.

- Create an ***Abstract Factory*** with Mac, Windows and Unix "families" of subclasses. In one place (and one place only) in the code, determine which platform is being used, and thus determine the correct concrete subclass of the ***Factory*** to instantiate. Note that Button is also abstract.

```
Button b = factory.makeButton();
```

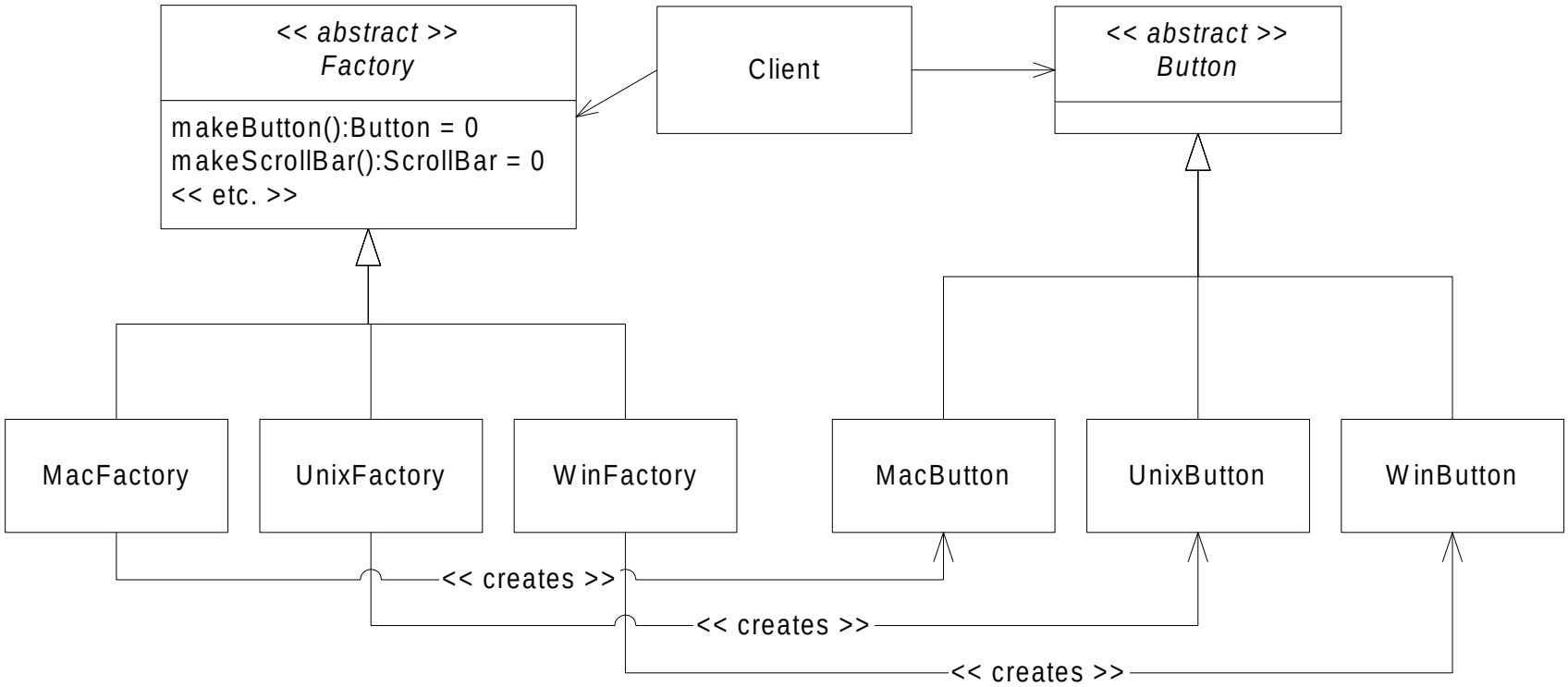
Factories are sometimes referred to as "virtual constructors"

- Note: The ***Prototype*** pattern provides yet another way to avoid hard-coding the name of the concrete subclass(es).

```
Button b = (Button) buttonPrototype.copy();
```

Abstract Factory Structure

- Client code remains blissfully unaware of the various concrete classes...



Patterns are everywhere...

- Human beings are great at “pattern recognition.”
- Our brains are hard-wired to have this capability.
- We are also good at thinking in terms of high-level abstractions.

As we study design patterns…

- we learn to *recognize* patterns when we see them
- we learn to use patterns *generatively* while designing

Two Similar Problems

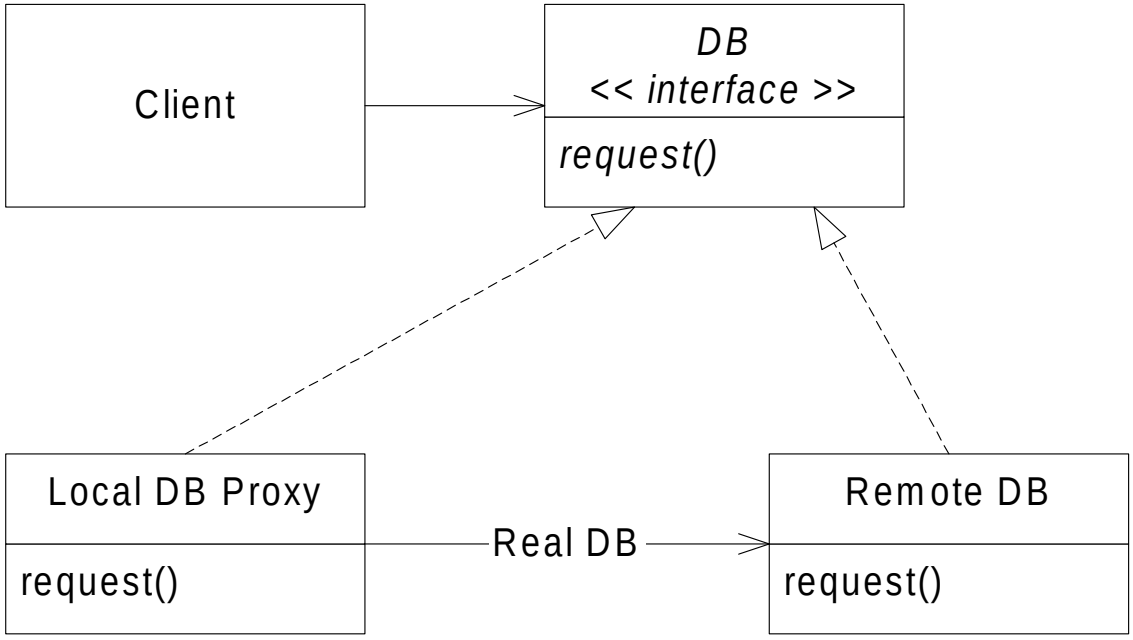
1. You have a *remote* database server, and you want a *local* object to encapsulate all requests to that remote server, so that the local application programmer can treat the remote server as if it were local.

   How can we do this easily?

2. You have a document viewer program which may embed large images which are slow to load; but you need the viewer itself to come up quickly. Therefore, images should get loaded only as needed (when they come into view), not at document load time.

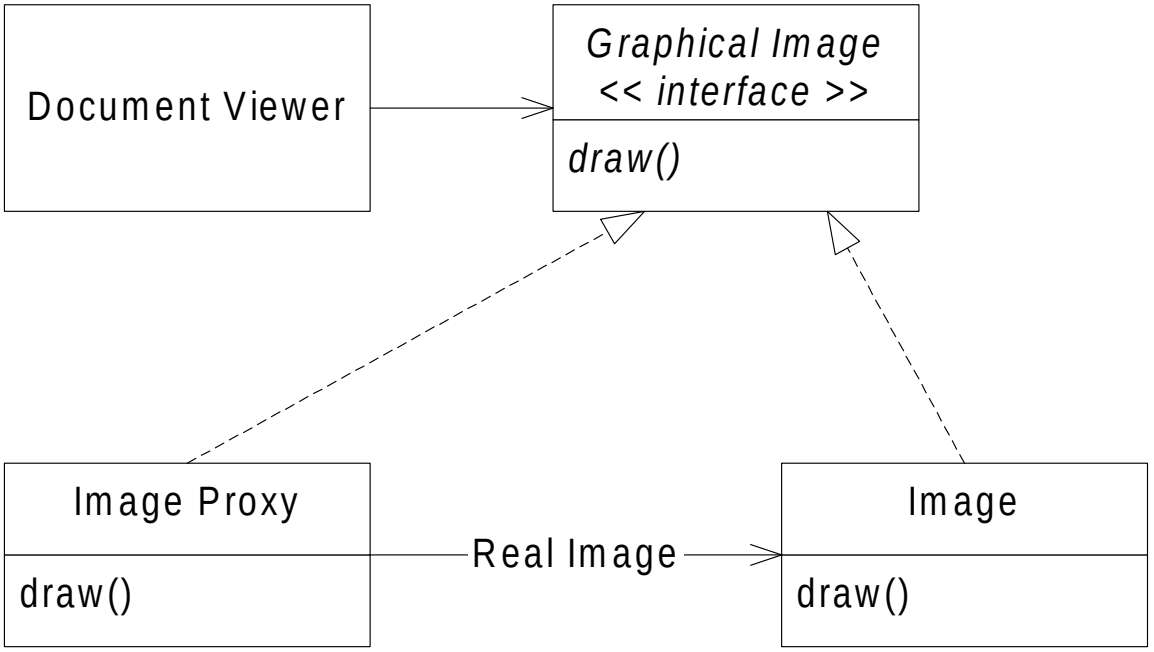
   How can we do this without complicating the design of the viewer?

Remote Database *Proxy*



The Local DB Proxy's request() encapsulates all network interactions as it invokes the request() on (delegates to) the Real (Remote) DB. Note: it is not required that the remote and local DB share the same interface.

Graphical Image *Proxy*



The Image Proxy's draw() will first load the Real Image from the disk if it hasn't been loaded already; then it will forward (delegate) the draw() request to the loaded Image.

Design Pattern: *Proxy*

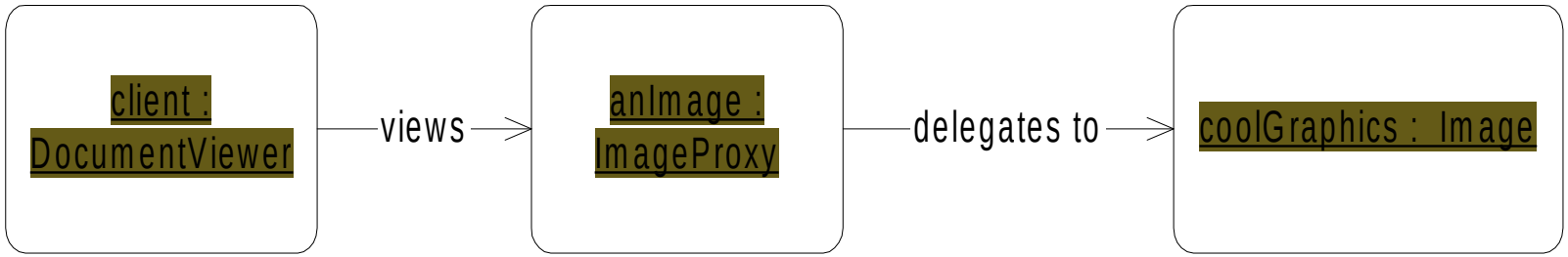
***Intent*: C**ommunicate with a representative of the server object, rather than with the server object directly...

***Applicability*:**

- As a ***gatekeeper*** to another object, to provide *security*, used when access to the real object must be protected.
- As an ***ambassador*** for a *remote* object.
- As a ***virtual*** stand-in for an object, used when it might be difficult or expensive to have a reference to the real object handy.

- If the *proxy* and the real thing have the same interface, then the client(s) can't tell them apart, which is sometimes what you want. Other times, the *proxy* might wish to *Adapt* the real thing's interface to be more convenient.
- Knowledge of this pattern is essential.

The Proxy object delegates



Design Pattern: *Adapter*

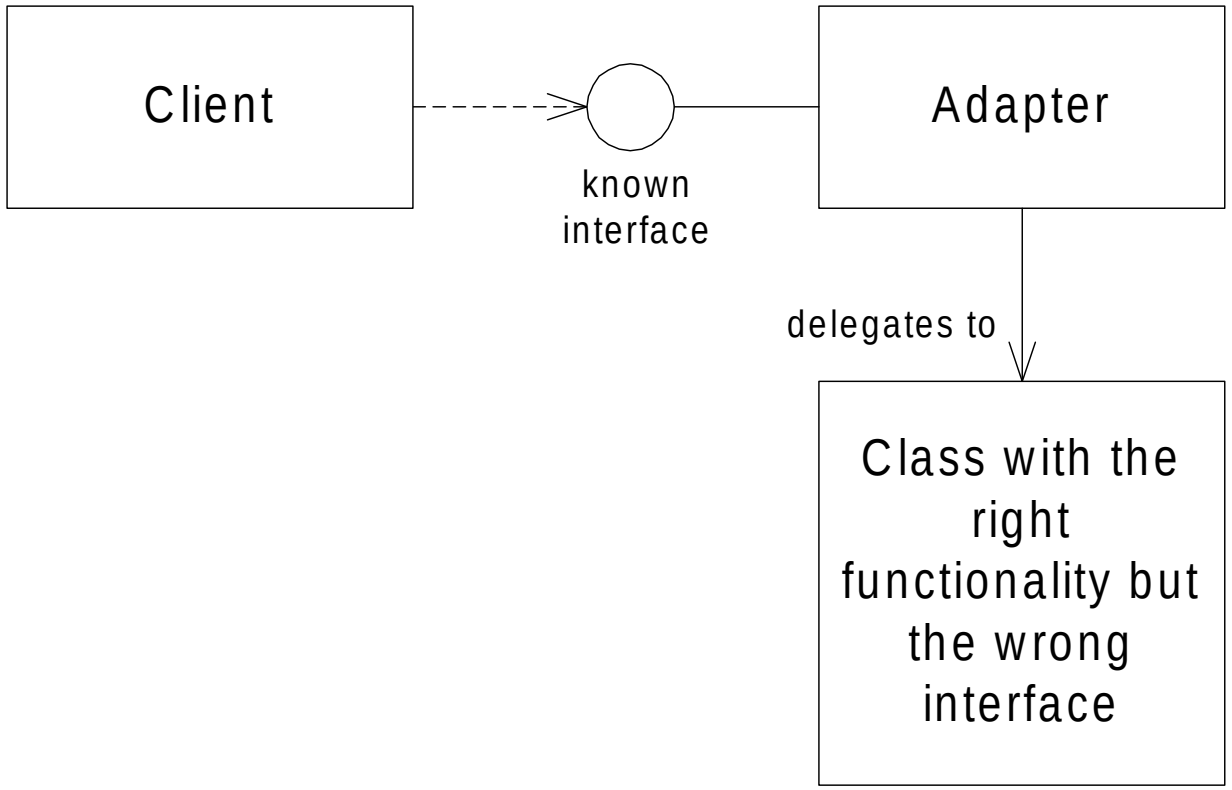
Intent: Convert the interface of a class into another interface clients expect. *Adapter* lets classes work together that couldn't otherwise because of incompatible interfaces.

Also known as a ***Wrapper***.

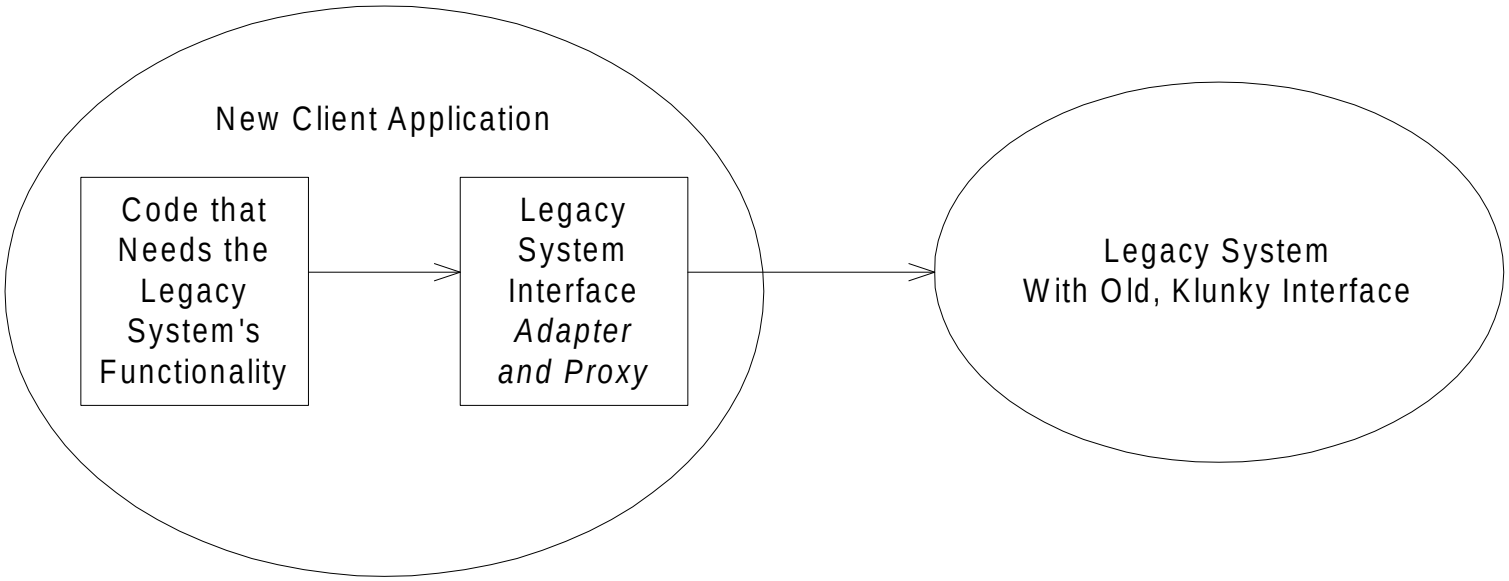
Use the *Adapter* pattern when:

- You want to use an existing class, and its interface doesn't match the one you need. This is sometimes the case with third party code, and also with machine-generated "stub" files.
- You are using a class or API that everyone on the team does not want to take the time to learn.

Adapter Example



Adapter & *Proxy* Together



Adapter Example

Wrap a native C call, using JNI (the **Java Native Interface**)…

- The following Java class encapsulates all of the details of JNI.
- This class is responsible for loading the .dll (or .so) file, calling the C function, and returning the results; it also deals with error conditions gracefully.

```
// Simple Java client code:
UniversalIDGenerator uidg = new UniversalIDGenerator();
byte[] theID = uidg.getNewID();
```

Adapter Code

```
public class UniversalIDGenerator
{
  static // Happens once upon loading the class
  {
    // Unix: libUidLib.so . . . Win32: UidLib.dll
    // Unix: LD_LIBRARY_PATH . . . Win32: PATH
    try {
      System.loadLibrary( “UidLib” );
      Log.info( "Loaded UidLib”, null );
    }
    catch( Throwable t ) {
      Log.error( "Unable to load UidLib”, t );
    }
  }
```

Adapter Code

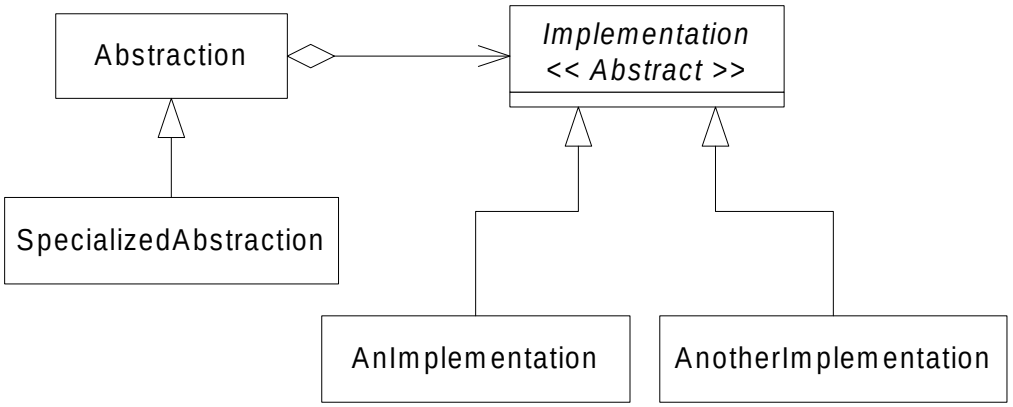
```
  private native void createUID( byte[] bytes ); // in C.

  public byte[] getNewID() {
    byte[] rawBytes = new byte[16];
    try {
      createUID( rawBytes );
    }
    catch( Throwable t ) {
      Log.error( “UniversalIDGenerator.getNewID()", t );
      rawBytes = null;
    }
    return rawBytes;
  }
}
```

Design Pattern: *Bridge*

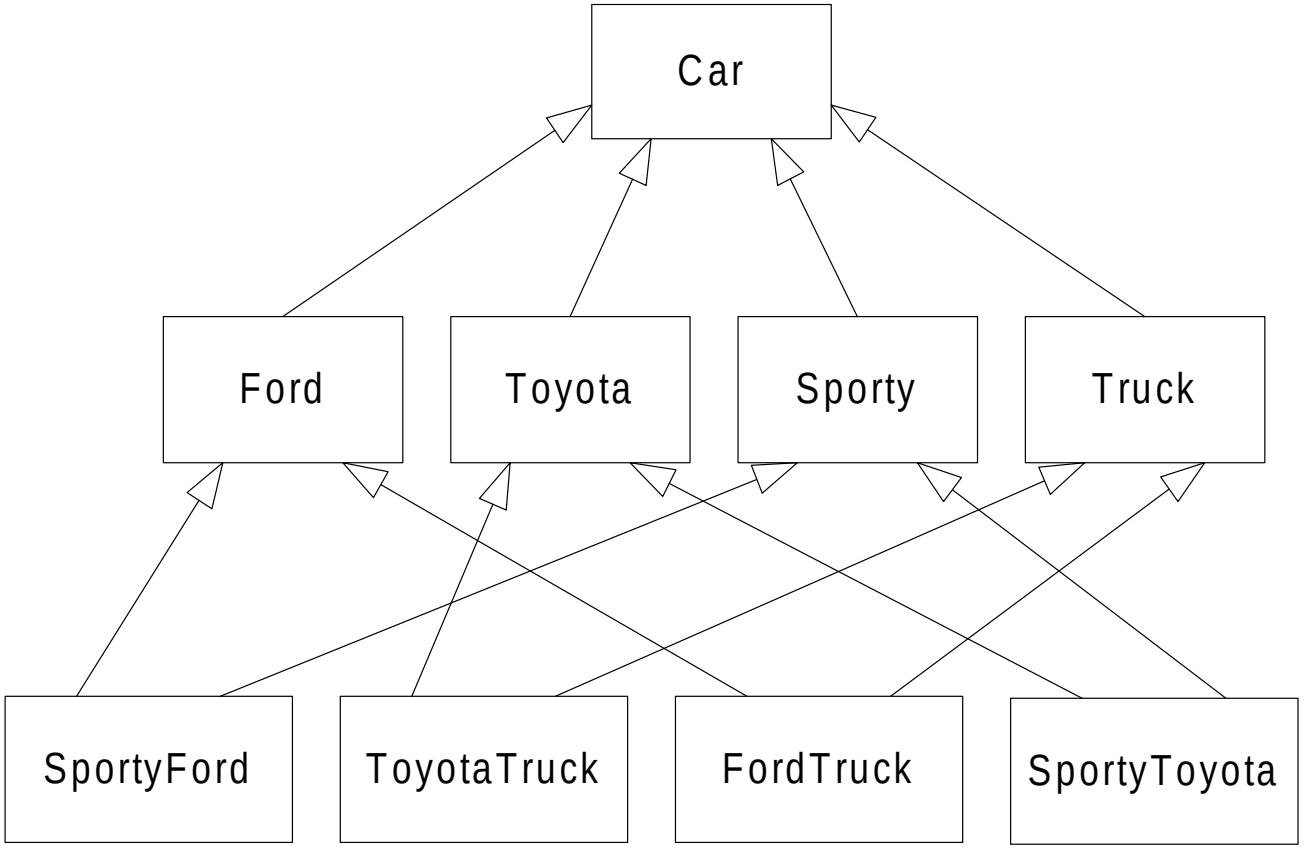
***Intent**:* Decouple an abstraction from its implementation.

- Allows the implementation to be selected at runtime.
- *A*llows separation of a “big” abstraction class into two smaller classes, one for the “abstraction” and one for the “implementation” - the two may vary independently.
- Also applicable to simplify a complex class hierarchy.



Bridge Example

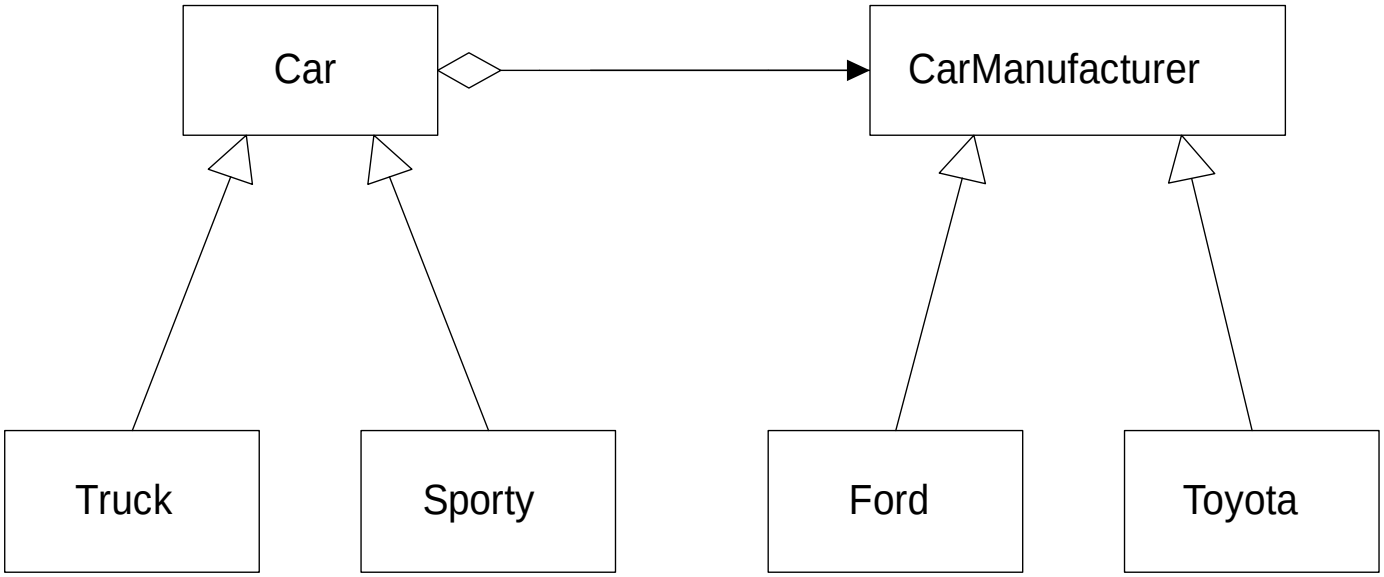
- How can we simplify this design?



Bridge Example

Apply the ***Bridge*** Design Pattern

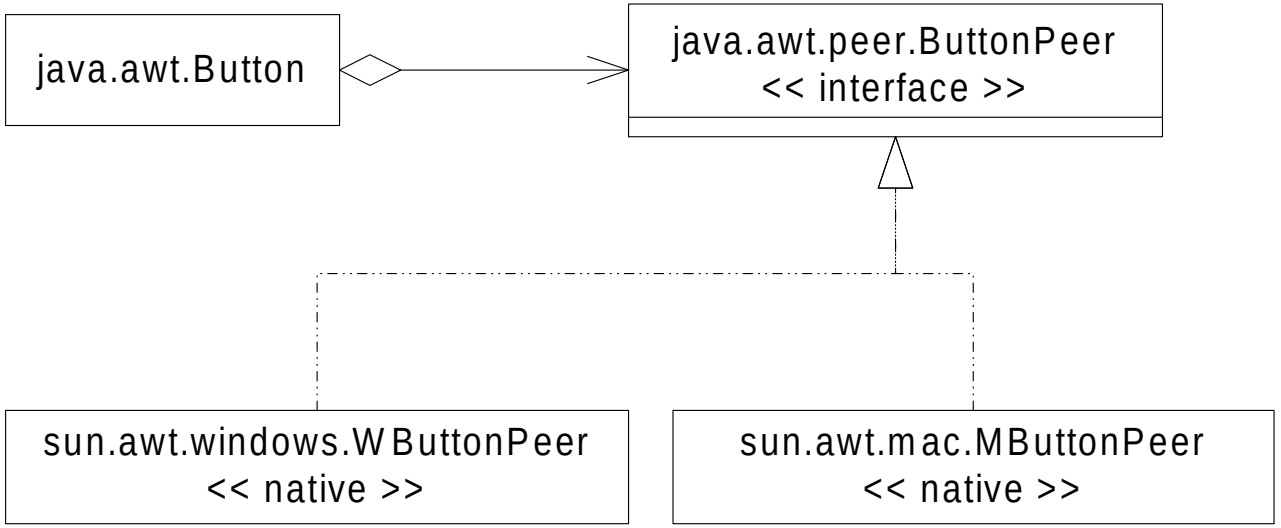
- You might use Bridge when you might otherwise be tempted to use multiple inheritance...



Bridge in the Java AWT

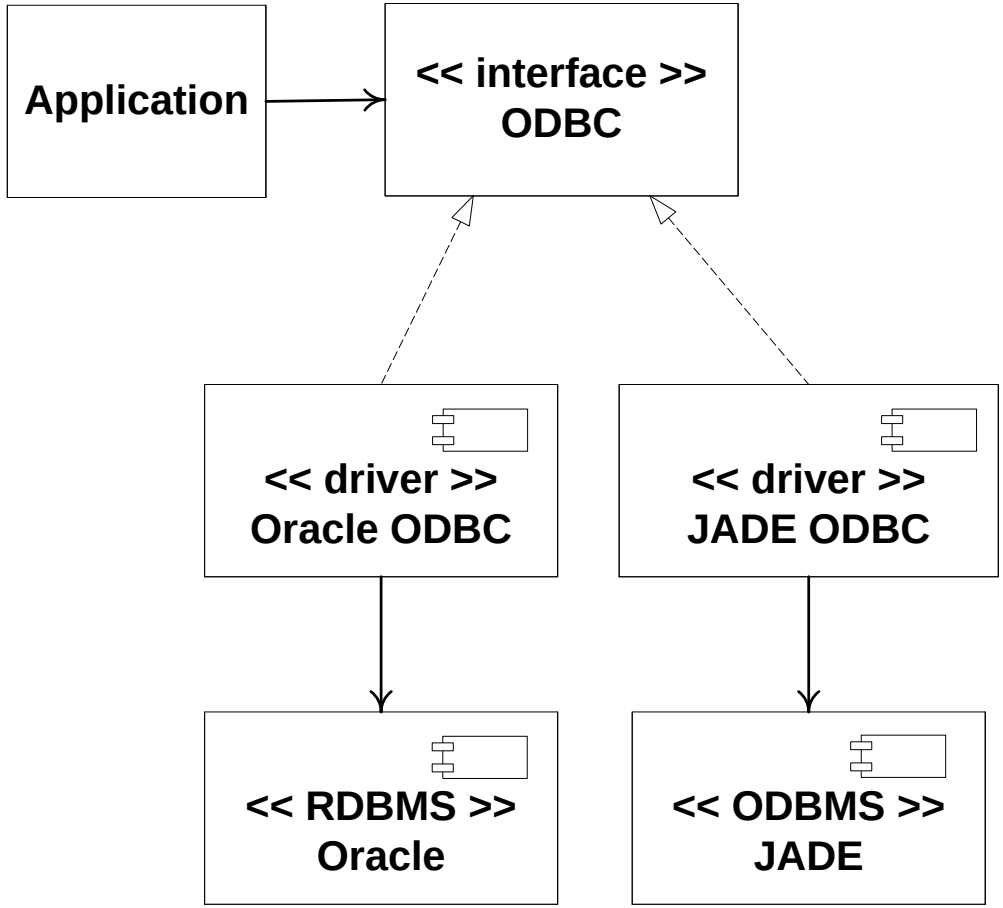
The Java AWT 1.1.x uses the *Bridge* pattern to separate component abstractions from the platform dependent “peer” implementations.

The **`java.awt.Button`** class is 100% pure Java, and is part of a larger hierarchy of GUI components. The **`sun.awt.windows.WButtonPeer`** class is implemented by native Windows code.



Adapter and *Bridge* Together

- ODBC specifies an abstract interface that clients expect. The ODBC driver for each specific database engine is an *Adapter*.
- The overall design that incorporates these drivers is an example of *Bridge*, separating application development from driver development.

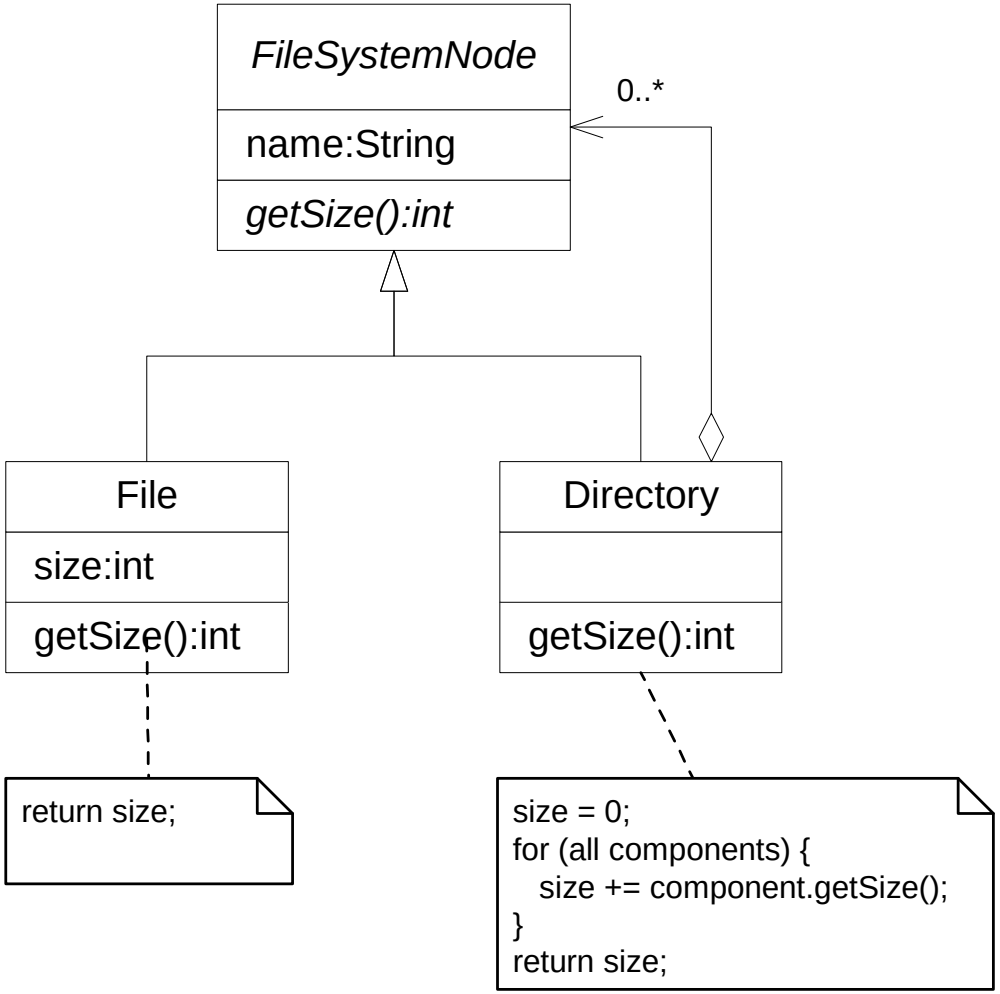


Example: New File System Feature . . .

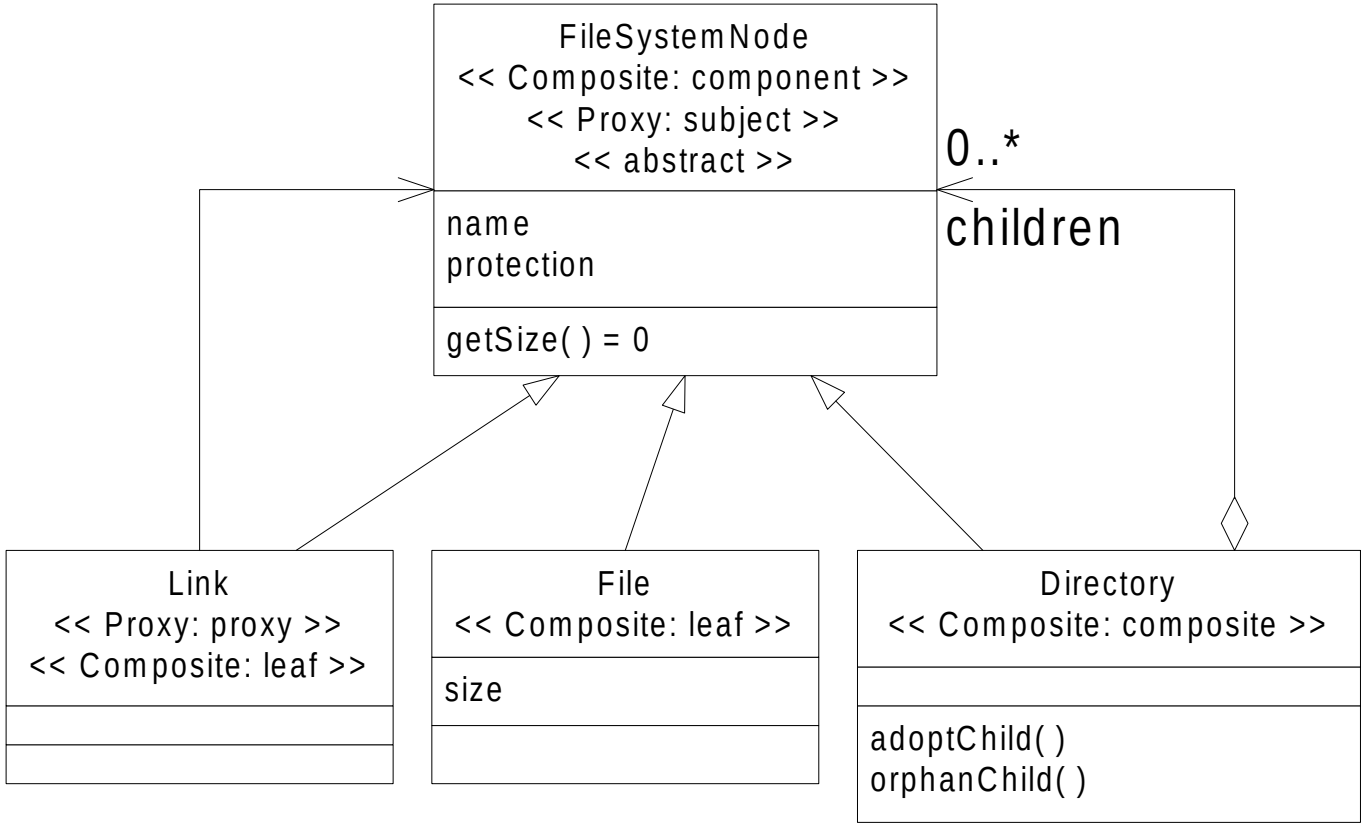
On UNIX systems, there is a feature in the File System called a Link that we wish to add to our design (like the Windows shortcut or the Macintosh alias). A Link is simply a navigational shortcut which allows a user to see a virtual copy of a file or a directory, as if it were local to the user's current location in the directory hierarchy. For most operations, the Link behaves exactly like the thing it is linked to, except that it may be deleted without deleting the actual directory or file.

- Draw a class diagram for the Link feature.

Composite Example: File System

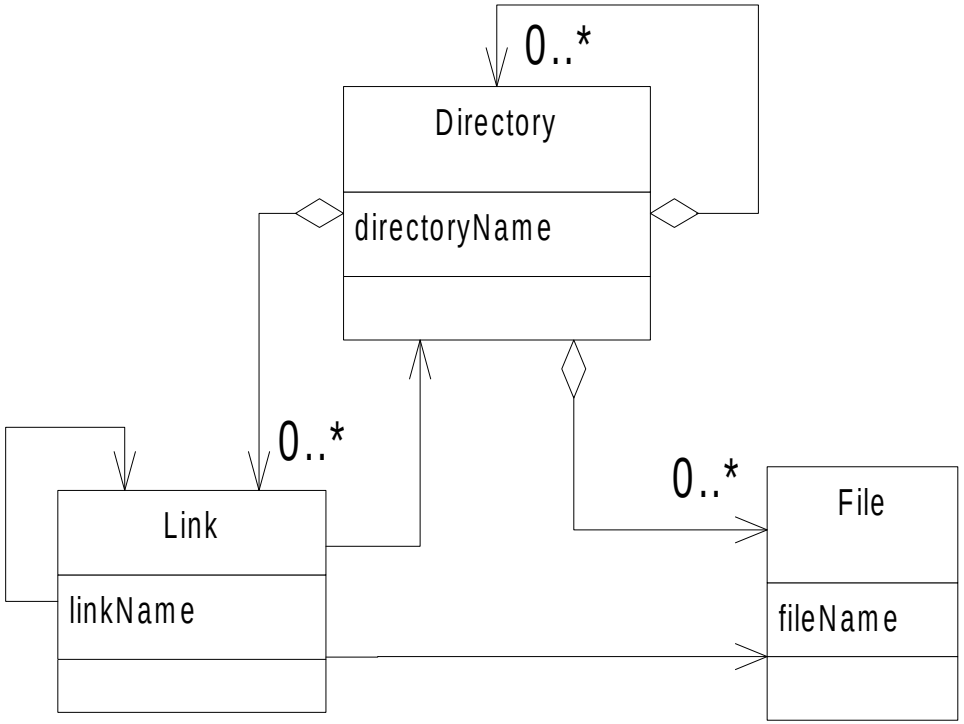


Composite & *Proxy* Together



Composite & *Proxy* Together

- Compare the previous slide with this “correct” design...
- An even worse “correct” design would be to have only two classes, Directory and File, with all of the Link code “hacked in” ...



Design Pattern: *Observer*

Intent: Allow objects to automatically notify each other upon changes in state, without tight coupling.

Also known as: “Publish / Subscribe” … “Source / Listener”

Applicability:

- Multiple views of a model (subject) need to stay in sync. No view should know about any other.

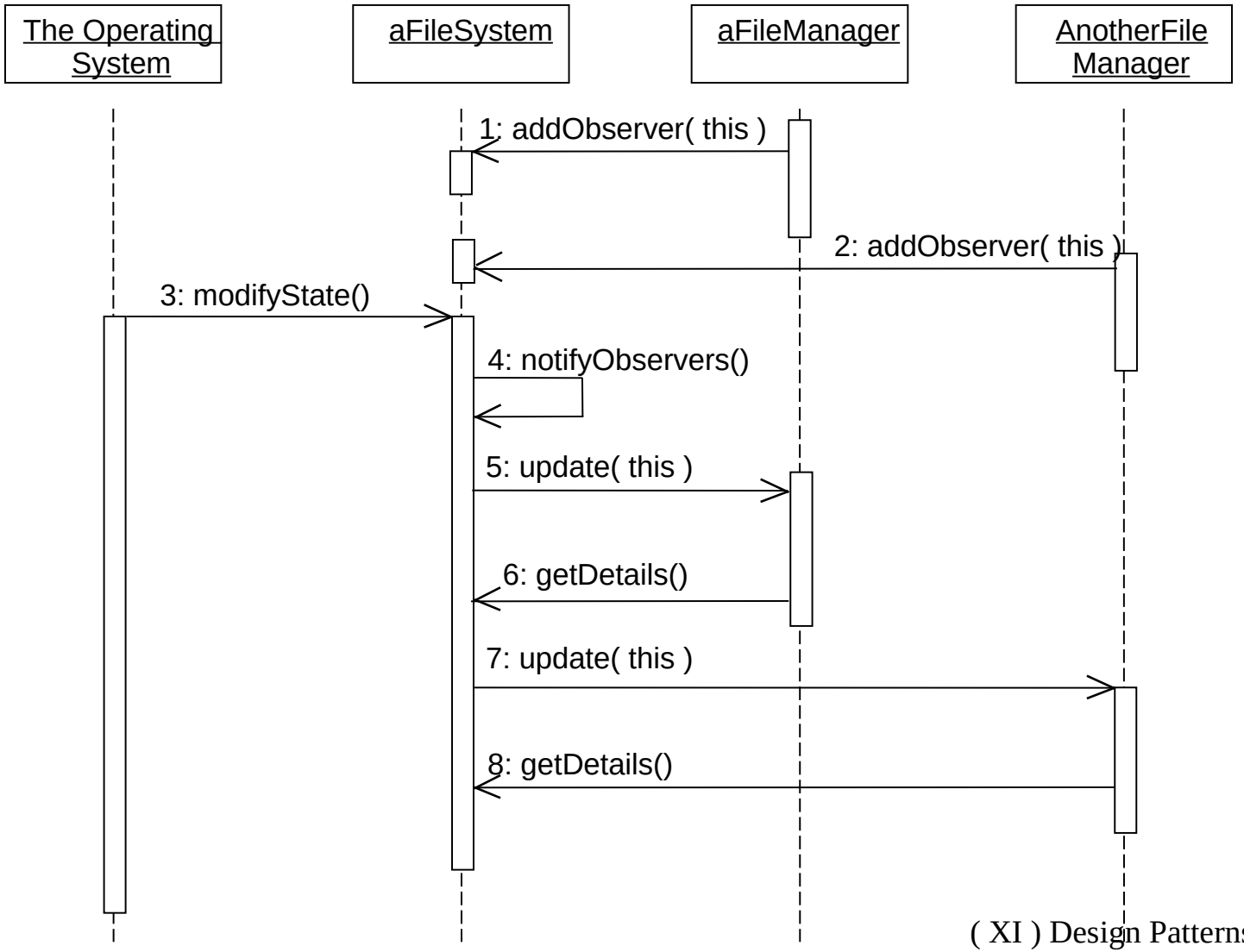
Pros:

- Excellent communication protocol.
- Avoids polling

Cons:

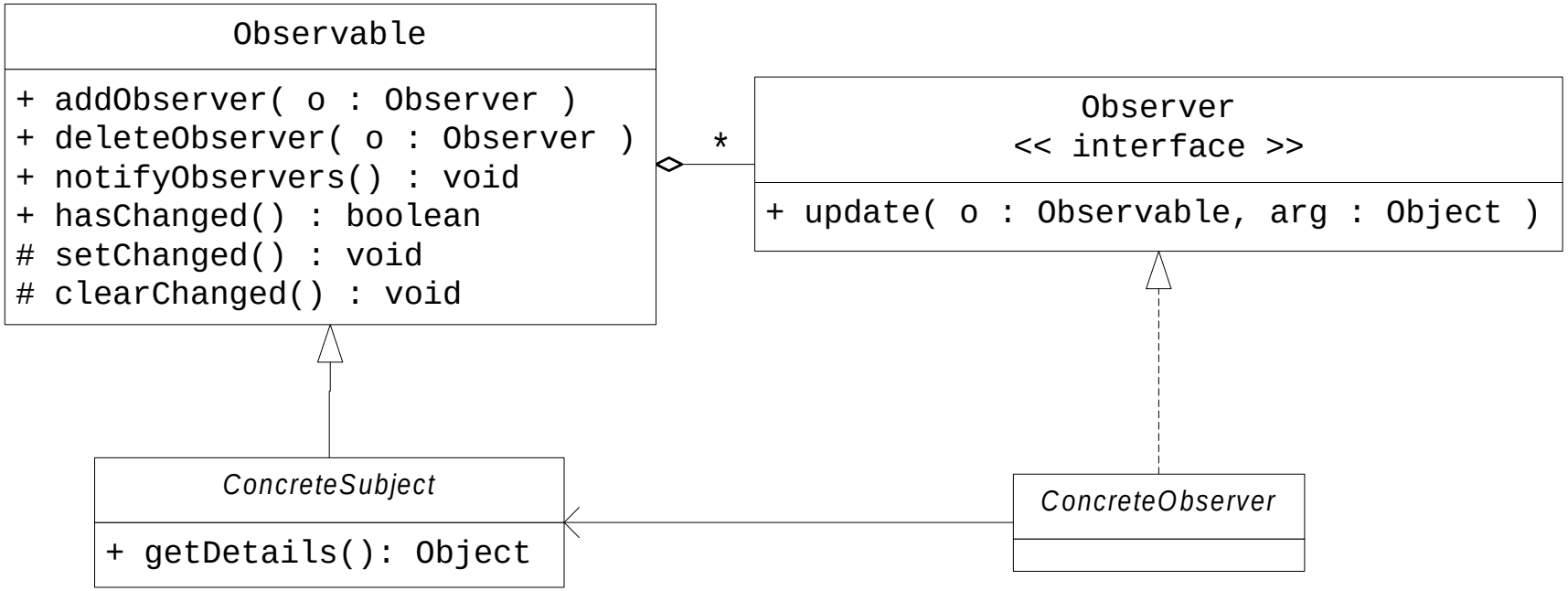
- None. Knowledge of this pattern is essential.

Observer Pattern Sequence Diagram

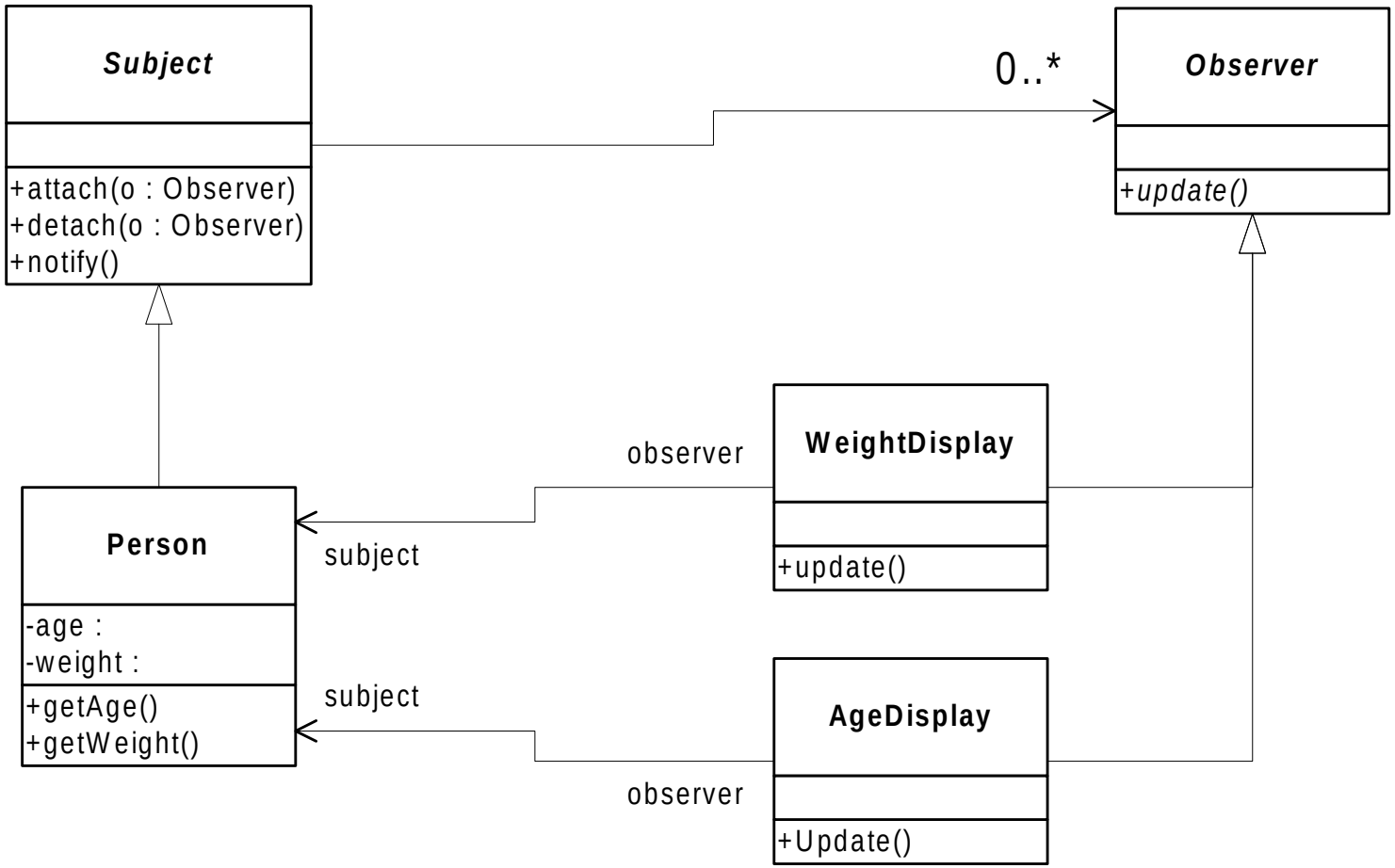


Java Support for *Observer*

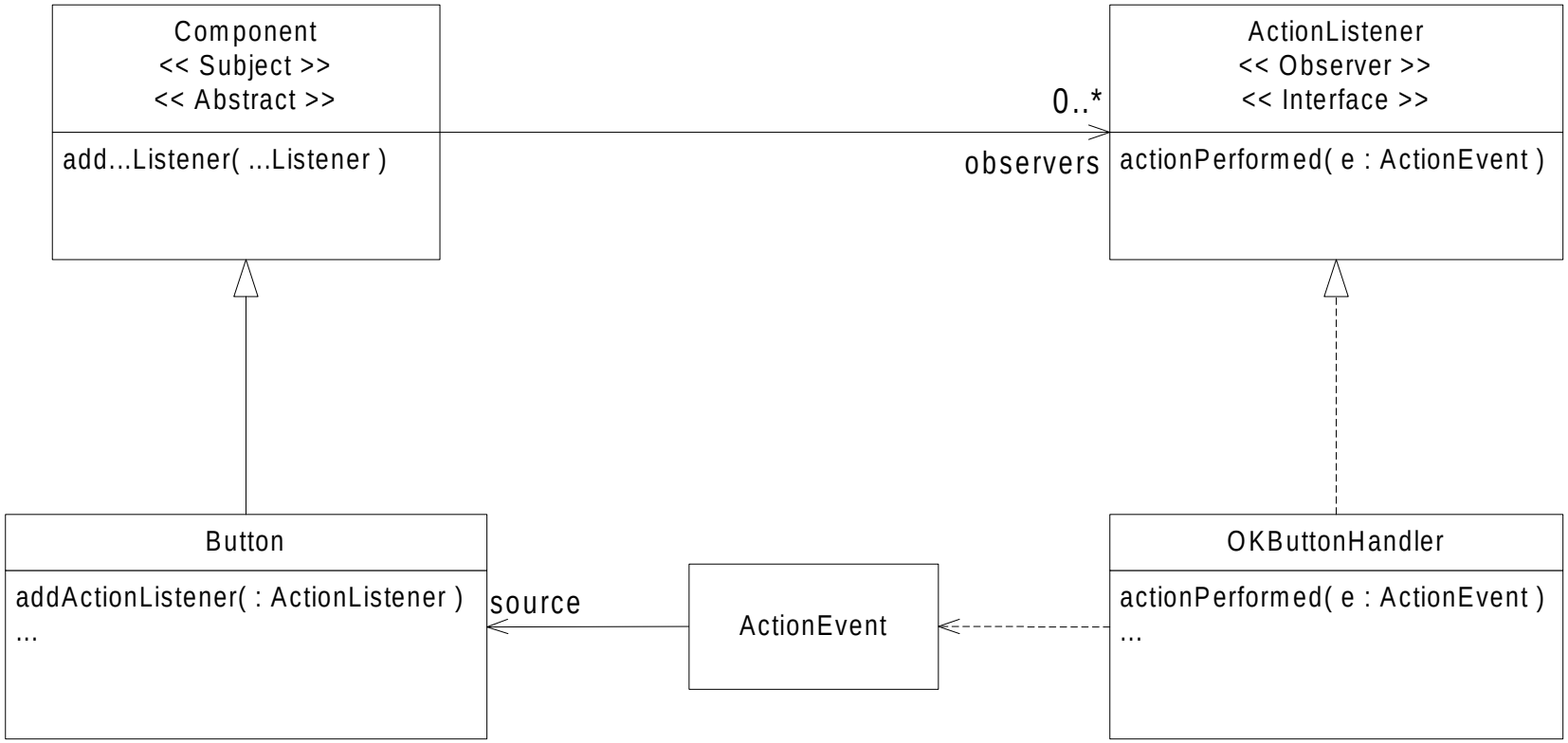
The java.util package provides an Observable class and an Observer interface:



Example: GUI displays *observe* a Person



Example: Java AWT 1.1 Event Model



- See example code on next slide...

Observer variant: Java AWT Events

```
class MyScreen extends java.awt.Frame {
  public void main( String[] args ) {
     java.awt.Button okButton = new java.awt.Button( “OK” );
     okButton.addActionListener( new OKButtonHandler() );
     /* … */
  }
  private void doOk() { /* … */ }
  // Inner Class...
  class OKButtonHandler implements ActionListener {
     public void actionPerformed( ActionEvent e ) {
       doOK(); // click the OK Button to invoke me.
     }
  }
```

Design Pattern: *Null Object*

***Intent*:** Simplify code by providing an object that does nothing.

- Not one of the GoF patterns.

```
interface ILog {
  public void log( String msg );
}
// subclasses: FileLog, ScreenLog, DBLog
// N.B.: Use Log4J instead.
```

Code without a Null Object

```
class Client
{
  private ILog il = null; // initialized elsewhere

  public void code( )
  {
    if( il != null ) il.log( “1” );
    // ...
    if( il != null ) il.log( “2” );
  }
}
```

- How can the Client’s code() be simplified using a “*Null-Object*”?

Simpler Code with Null Object

```
class NullLog implements ILog
{
  public void log( String msg ) { ; } // Does nothing
}

class Client
{
  private ILog il = new NullLog( );
  public void code( )
  {
    il.log( “1” ); // No conditionals required.
    il.log( “2” );
  }
}
```

Null Object variation: *Stub / Mock*

***Intent*:** Provide a trivial (or "null" or "mock") object that stands in for a complex object or subsystem whenever the complex object is unavailable.

- In some cases, a ***Stub*** object behaves just like a *Null Object*.
- In other cases, the ***Mock*** might provide a fake, hard-coded, or trivial version of the service in question.
- Mock objects are great for ***Unit Testing***.

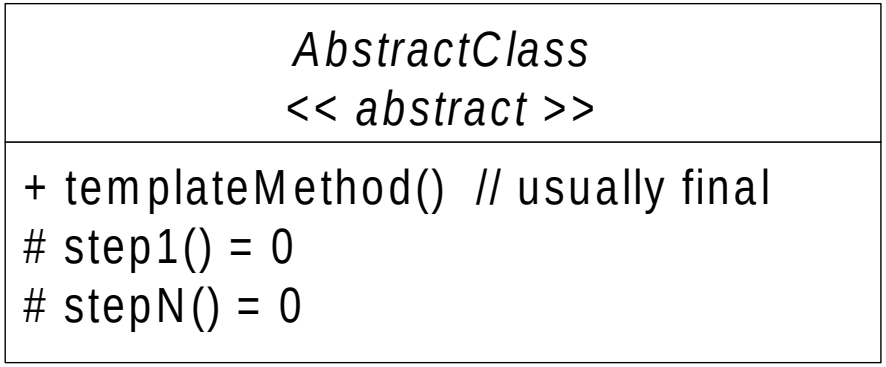
Very common example: A stubbed-out database:

- Useful at the beginning of a project when the real database is still under construction.
- Also useful for "demo" versions of software that must run without being connected to the real database.

Design Pattern: *Template Method*

***Intent*:** Have an abstract class define the invariant parts of an algorithm, deferring certain steps to its subclasses.

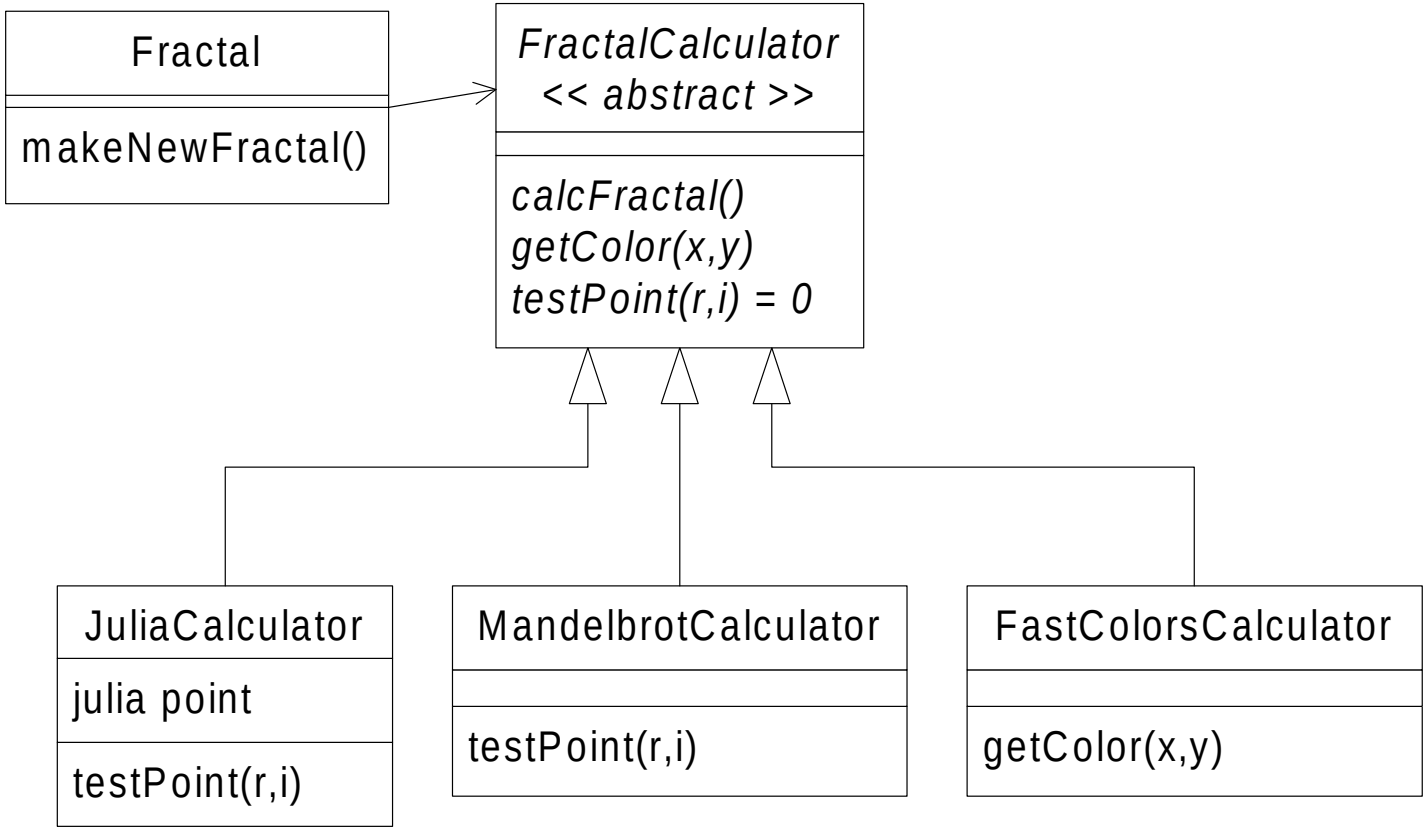
- The class with the templateMethod() is designed to have subclasses which implement certain steps of the process.
- Example: The Fractal Applet's FractalCalculator.



Template Method + Observer

```
abstract class ProcessManager extends java.util.Observable {
  protected final void process( ) {
    try {
      initProcess( );
      doProcess( );
      setChanged( );
      notifyObservers( ); // my state changed.
    }
    catch( Throwable t ) {
      Log.error( “ProcessManager.process():”, t );
    }
  }
  abstract protected void initProcess( );
  abstract protected void doProcess( );
}
```

Strategies w/ Templete Method



Layers

MVC implementations often use ***layers***:

- Presentation Layer / The **View**
- Application Layer / The **Controller**
- Business Layer / The **Model**
- Data Layer / object to relational mapping

Each layer provides a coherent set of services through a well defined interface. This helps to confine change, encourages reuse, and facilitates unit testing.

- The View knows about the Model but not vice-versa – use ***Observer***.

Design Pattern: *Command*

***Intent**:* Allow requests to be modeled as objects, so that they can be treated uniformly, queued, etc...

- The Command pattern is often used in distributed applications using message-oriented middleware. The messages are essentially commands.
- In Model – View – Controller designs, the View might determine the user's intent and send commands to the Controller. This facilitates undo, and scripted testing.
- A Command should be responsible for creating its own *undo* Command.

Cut – Paste Example

- **Before Cut**: Some text is selected.
- **After Cut**: There is no selected text. The cursor position is set.
- **Before Paste**: There is possibly some selected text. There is text in clipboard.
- **After Paste**: The previously selected text, if any, is deleted. Text from clipboard, if any, is inserted. The cursor position is set. There is no selected text.

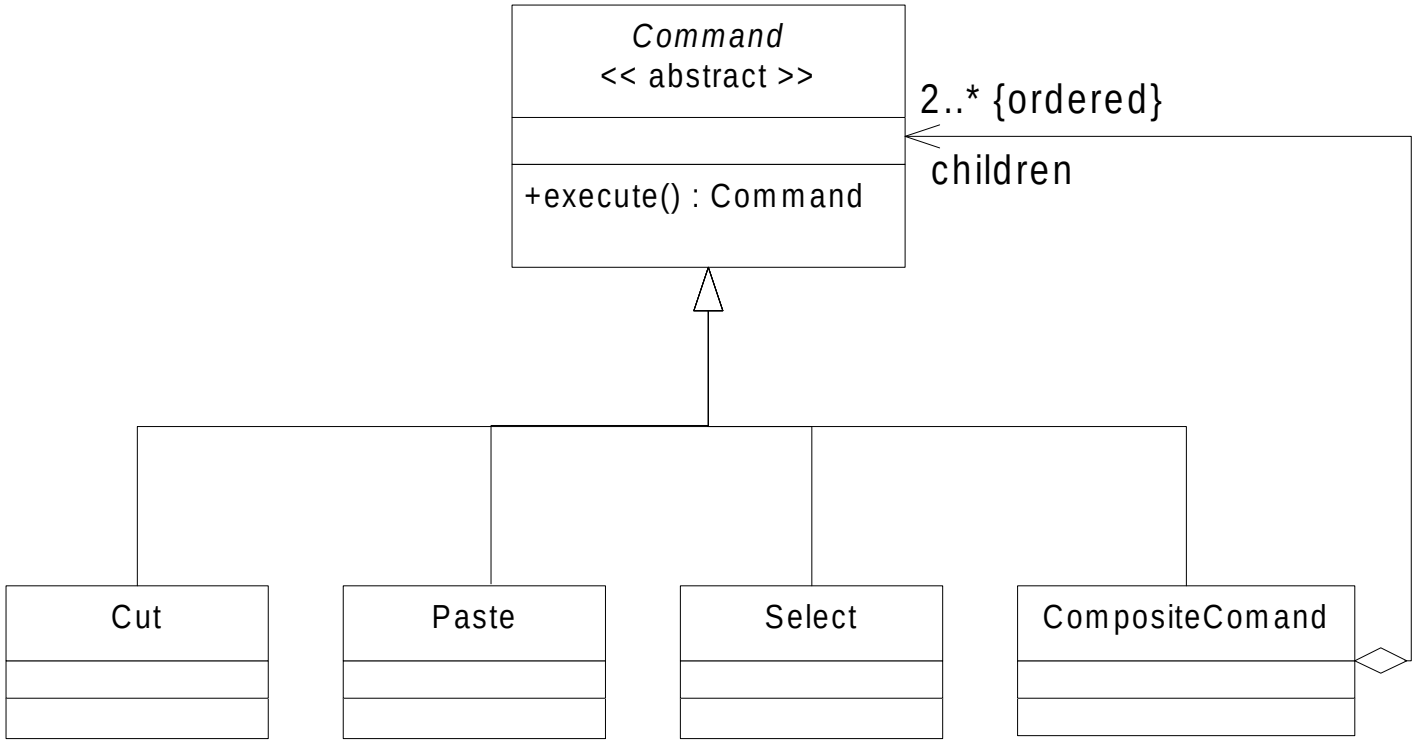
Cut's execute() method returns an inverse (undo) *Command* with the following functionality:

1) PASTE the previously CUT text.

2) SELECT the previously CUT text.

- Note that this is a ***Composite*** Command.

Composite Command



Design Pattern: *Memento*

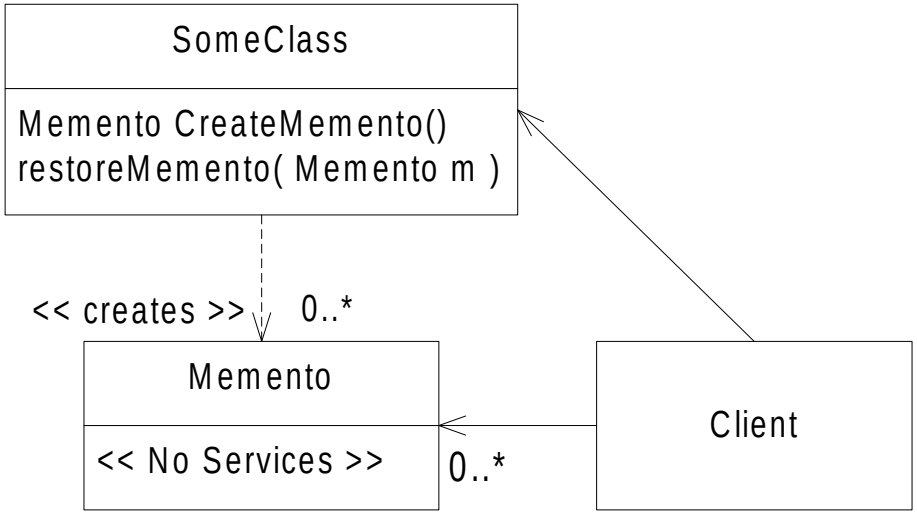
***Intent*:** Save an object's state without violating encapsulation.

***Applicability*:** The state of an object must be saved so that it can be restored later. The *Memento* object contains the necessary state information.

This is another way to implement "undo."

Example: Java Beans save their state to a **`.ser`** file after being configured, using Java *serialization*.

How is it possible for data in the *Memento* to be available to *SomeClass*, but not to *Clients*?



Design Pattern: *Decorator*

Intent: Extend the responsibilities of an object without subclassing. Decorators may be chained together dynamically, so that each performs some function and then delegates to the next object for further processing.

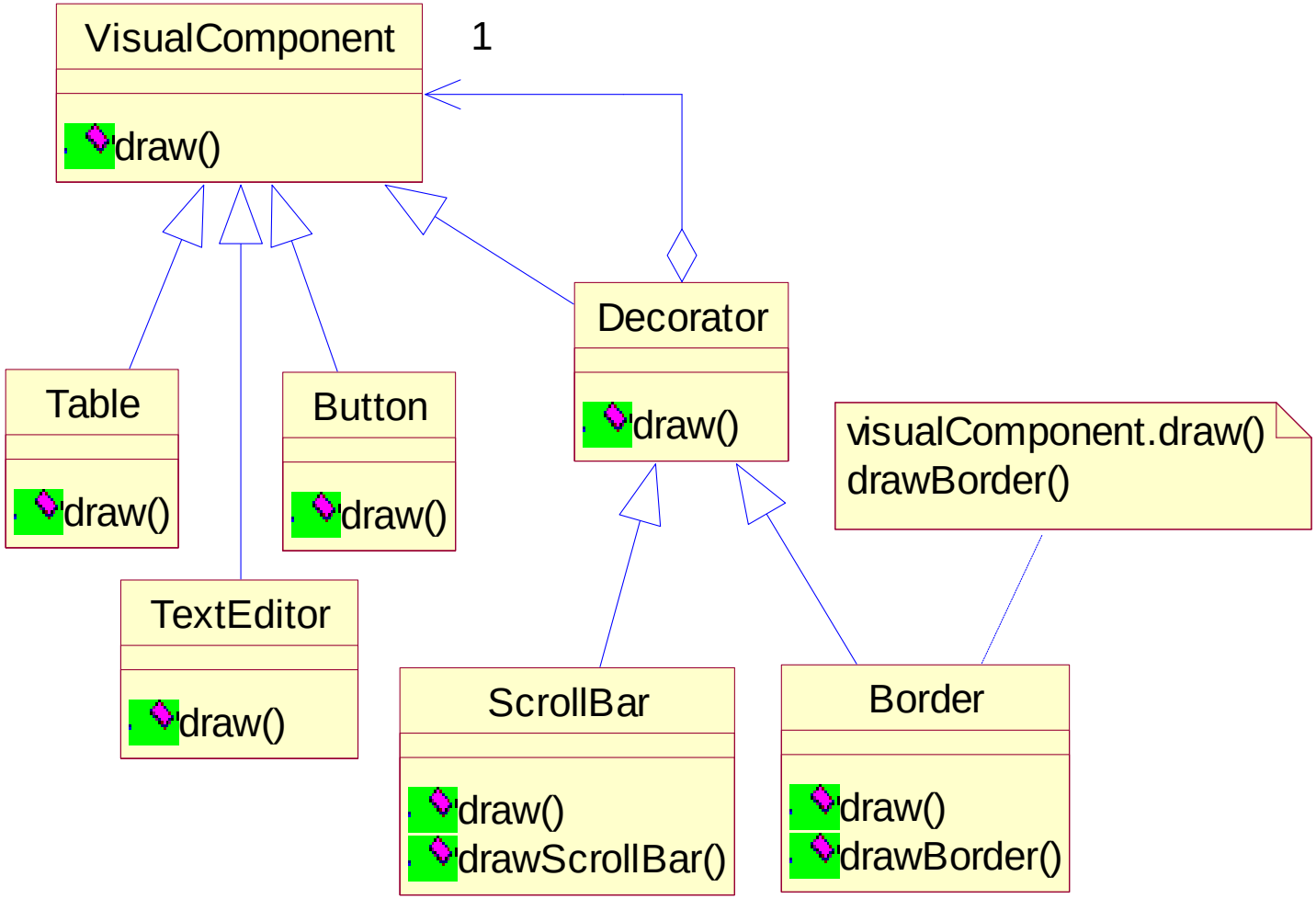
Example: Consider a GUI toolkit withy Buttons, Tables, and Text Editors. These are *components* which can be *decorated*.

```
VisualComponent vc = new ScrollBar(
  new Border( new TextEditor() ));
vc.draw();
```

- Note the distinct lack of classes of the type:

`TextEditorWithScrollBar` and **`ButtonWithBorder`**, etc…

Decorator Design Pattern



Design Pattern: *Visitor*

Intent: Separate each chunk of system functionality into its own class, independent from the system's structure. Define new operations on a collection of classes without changing those classes.

Issues:

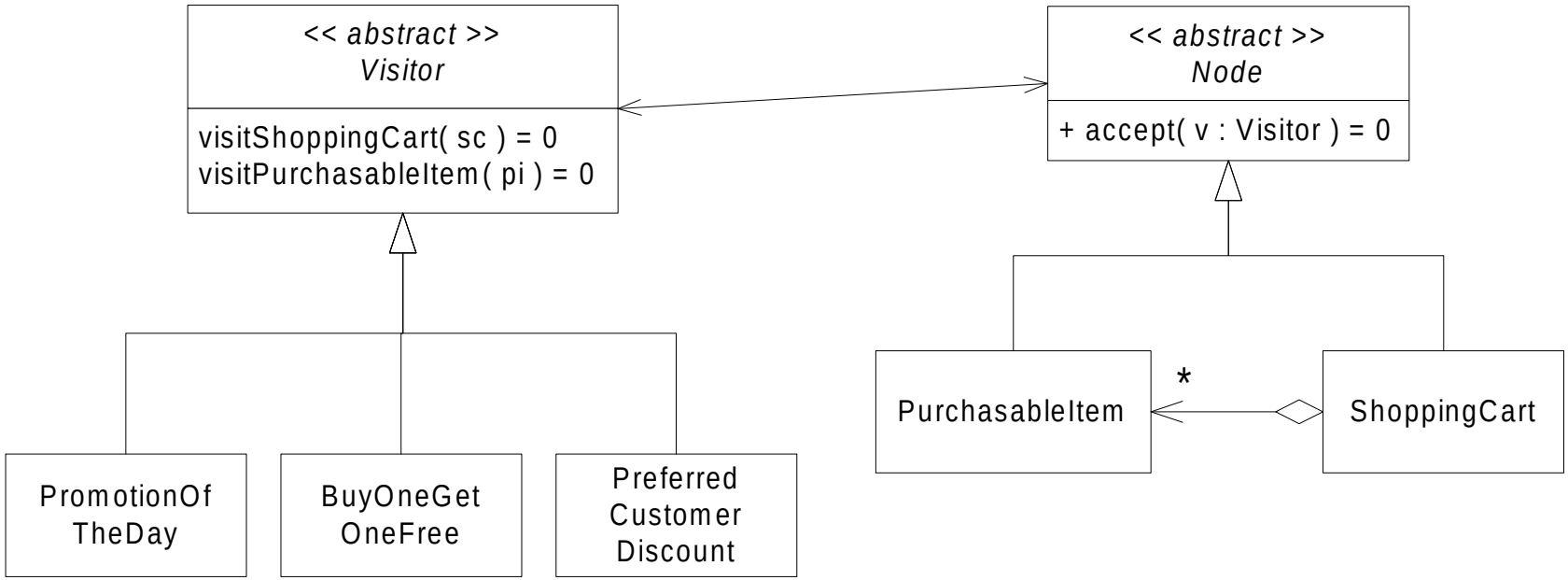
- Visitors may be used in conjunction with Commands, which also represent chunks of system functionality; Commands might create Visitors.
- The Visitor pattern helps to prevent "polluting" structural classes with behavioral code.
- If the system structure is also not stable, then this design becomes difficult to manage.

E-Commerce Web Site Example

- A e-commerce web site uses two reusable abstractions: `ShoppingCart` & `PurchasableItem`.
- There are constantly changing promotions & discounts.
- Avoid “hacking” at the `ShoppingCart` & `PurchasableItem` code every time there’s a new promotion.
- Give each promotion its own class that “visits” all of the `PurchasableItems` in the `ShoppingCart` to compute the total price.
- The client code should be something like this:

```
PromotionOfTheDay visitor = new PromotionOfTheDay();
shoppingCart.accept( visitor );
SalesPrice price = visitor.getTotalPrice();
```

E-Commerce Web Site Participants

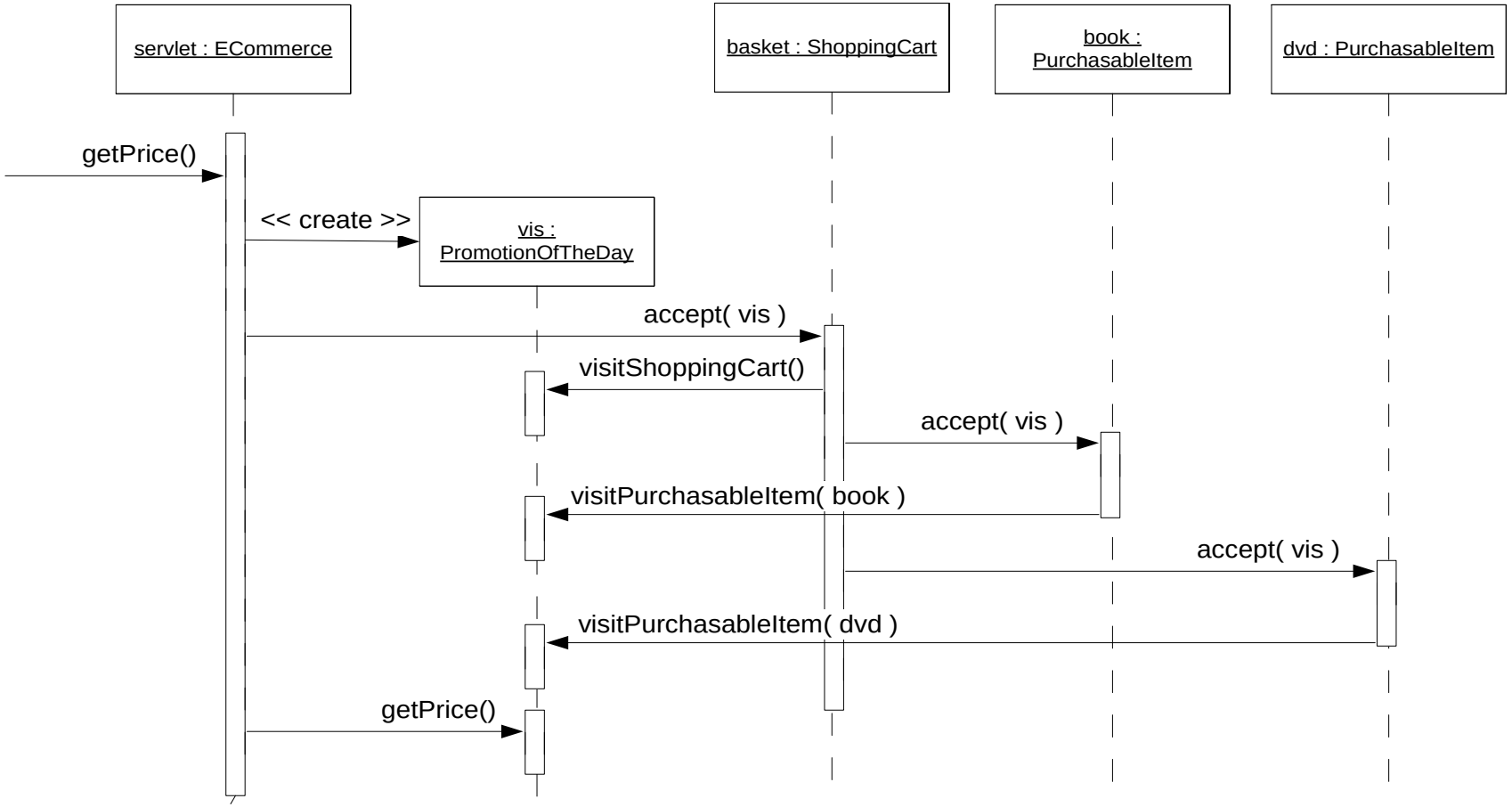


E-Commerce Web Site

- `ShoppingCart` knows nothing about `PromotionOfTheDay`.
- `ShoppingCart.accept()` simply passes the `visitor` along to all of the `PurchasableItems` in the cart, calling `accept()`.

```
class PurchasableItem {
  private Money price;
  private String description;
  public Money getPrice() {
    return price;
  }
  public void accept( Visitor v ) {
    v.visitPurchasableItem( this );
  }
}
```

Visitor Sequence Diagram



Double Dispatch

There are two cases where polymorphism is used since there are two generalizations at work: the Nodes and the Visitors.

This leads to the notion of “double dispatch.”

- The first “dispatch” is:

```
node.accept( visitor );
```

- Enter **`ConcreteNodeA’s accept()`** method:

```
accept( Visitor vis ) {
  vis.visitConcreteNodeA( this );
  continueAccepting( vis );
}
```

- The second “dispatch” is:

```
vis.visitConcreteNodeA( this );
```

Double Dispatch Detail

Polymorphism does not work on overloaded method signatures. Consider the following counter-example:

```
public class NotDoubleDispatch {
  public static void main( String[] args ) {
    Cmd cmd = new SubCmd();
    handle( cmd ); // not polymorphic
    cmd.execute(); // polymorphic
  }
  public static void handle( Cmd cmd ) {
    System.out.println("CmdHandler");
  }
  public static void handle( SubCmd cmd ) {
    System.out.println("SubCmdHandler");
} }
```

Double Dispatch Detail

Thus, in order for the visitor to know which kind of node it is visiting, the following code will ***not*** work:

```
class TenPercentOffVisitor {
  public void visit( ShoppingCart sc ) {}
  public void visit( PurchasableItem pi ) {}
}
```

Instead, we must use double dispatch, and ***not*** use method name overloading:

```
class TenPercentOffVisitor {
  public void visitShoppingCart( ShoppingCart sc ) {}
  public void visitPurchasableItem( PurchasableItem pi ) {}
}
```

More and More Patterns ...

Design Patterns: Elements of Reusable Object-Oriented Software

by Gamma et al., Addison-Wesley, 1994_<ISBN 0-201-63361-2>

Check online. Patterns are everywhere!

As a solution design professional, you must study pattern-oriented architecture.

Google: ***Cloud Design Patterns***, for example.