Object - Oriented Programming & Design

Part XI: Design Patterns

CSCI 4448 - Fall 2001

Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software

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

- Often referred to as the *Gang of Four* (GoF) book.
- **Best-selling computer science book of all time.**

- This classic book is the first of many books about "Patterns" in software.
- These patterns facilitate our thinking and communicating about OO designs.
- They provide "tried and true" design solutions worth considering.
- The patterns can also be thought of as examples of excellent OO design.
- The "pattern template" is a common format for transferring design expertise, experience and wisdom from master software professionals.
- Once learned, a pattern can be applied over and over again in different contexts.
- They reduce discovery costs and unnecessary design complexity.
- A *Design Pattern* is a practical (not necessarily novel) solution to a recurring problem, occurring in different contexts, often balancing competing *forces*.

Pattern Roots

Design Patterns have roots in the work of Christopher Alexander.

His classic book describes hundreds of patterns:

A Pattern Language - Towns - Buildings - Construction

Oxford University Press, 1977 <ISBN 0-19-501919-9>

- Examples: Shielded Parking, Pedestrian Street, South Facing Outdoors, Entrance Room, Couple's Realm, Children's Realm, Alcoves, Ceiling Height Variety, Closets Between Rooms, Waist-High Shelf, Different Chairs, A Place To Wait, Six-Foot Balcony, Bulk Storage, Fruit Trees, Vegetable Garden, ...
- Patterns often combine together in interesting ways to solve larger problems than any one pattern can do alone (one pattern flushes out the details of another). For example, construction patterns are used within buildings, within neighborhoods, within towns… When a collection of patterns becomes rich enough to be able to *generate* good solutions across some entire domain of design, it may be referred to as a *pattern language*. A pattern language also includes guidelines as to how and when to apply its patterns, given the particular needs and constraints of the context at hand, guiding the designer toward architectures which are both functional and aesthetically pleasing.

Design Patterns We Have Already Seen

- ***Mediator***: microwave oven / game referee
- ***Façade***: compiler / data server / customer service representative
- ***Strategy***: computer player
- ***Composite***: file system / menus / arithmetic expressions
- ***Iterator***: Java collections
- ***Decorator***: Java IO Streams & Readers
- ***State***: implementation for UML state diagrams
- ***Proxy***: database server proxy
- ***Singleton***: database server proxy / inventory / security manager

And don't forget the ***GRASP*** responsibility assignment patterns….

- ***Expert*** / ***Creator*** / ***High Cohesion*** / ***Low Coupling*** / ***Controller***

Design Pattern Templates

The Design Pattern "Template" indicates (more or less):

- The ***intent*** of the pattern.
- General description of the problem, including a description of the ***context*** in which the problem occurs.
- A description of the ***forces*** which make the context challenging.
- A motivating example.
- A generic solution, with a ***name***. These names become part of our shared vocabulary for communicating about design.
- Where and how to apply the pattern as a solution to the problem.
- Why the generated design structure is desirable.
- The benefits, trade-offs, consequences and costs of using the pattern.
- Implementation issues.
- Other useful information...

Software Forces

We should understand the *forces* that exist in our design space.

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

One property of good designs is that they tend to balance competing forces in an elegant way.

Some example forces that apply to software design:

- Memory usage must be minimized while performance is maximized.
- The design must be as simple, yet also as flexible as possible.
- The code must interoperate with existing components.
- The code must communicate asynchronously with a remote server.
- The code must never (never!) crash.
- The server must support 1000 simultaneous client connections.
- A Java applet must download to the browser quickly.
- Time to market & initial cost vs. features & long term quality.

Types of Patterns (GoF)

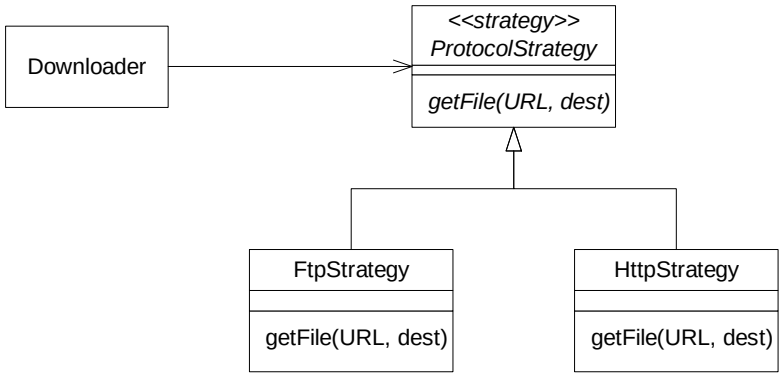
The GoF classifies its 23 design patterns into three categories:

- **Structural** (the composition of classes or objects)
  - *Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy*
- **Behavioral** (object interactions and the distribution of 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*

Example: Downloading Files...

- We need a piece of software that will download files from servers...
  - We currently know of two protocols that must be supported, HTTP & FTP.
  - Other protocols will be used in the future
- 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



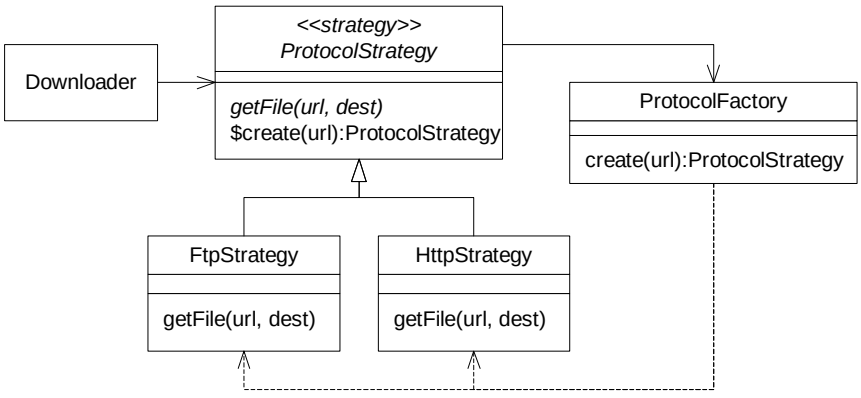
Using the Strategy

```
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 completely ignorant of the various concrete implementation classes.
- The ProtocolStrategy generalization breaks down at the point where new objects are instantiated.

Use a *Factory*



Inside the Factory

```
class ProcotolFactory {
    ProtocolStrategy create( String url )
    throws ProtocolNotSupportedException {
        ProtocolStrategy ps;
        url = url.toLowerCase();
        if( url.startsWith( "ftp" ) ) {
            ps = new FtpStrategy();
        }
        else if( url.startsWith( "http" ) ) {
            ps = new HttpStrategy();
        }
        else {
            throw new ProtocolNotSupportedException();
        }
        return ps;
    }
}
```

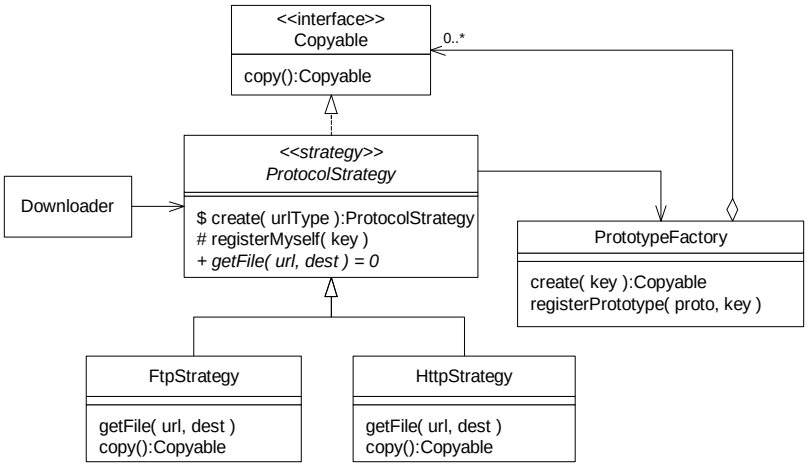

Using the Factory

```
class Downloader {
    public void download( String url, String dest )
    throws ProtocolNotSupportedException {
      ProtocolStrategy ps = ProtocolStrategy.create( url );
      ps.getFile( url, dest );
    }
}

abstract class ProtocolStrategy {
    static ProtocolFactory factory = new ProtocolFactory();

    public static ProtocolStrategy create( String url )
    throws ProtocolNotSupportedException {
      return factory.create( url );
    }
}
```

Yet More Considerations

- Consider: if ProtocolStrategy were an interface rather than an abstract class, we would have to move the static code out of ProtocolStrategy and into the factory class, causing clients to have to know about the factory class, an extra burden.
- It could also be argued that there is no need for a factory class at all, with the factory logic moved into the ProtocolStrategy class.
- We would like to generalize the Factory so that it can be configured dynamically; to do so, we need to get away from specifying the concrete class names that the Factory can instantiate, and the logic for choosing the right one.
- Ideally, we would like the system to be able to run 24x7.

Use *Prototypes*



Inside the Prototype 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();
} } }
```


So Where Are We?

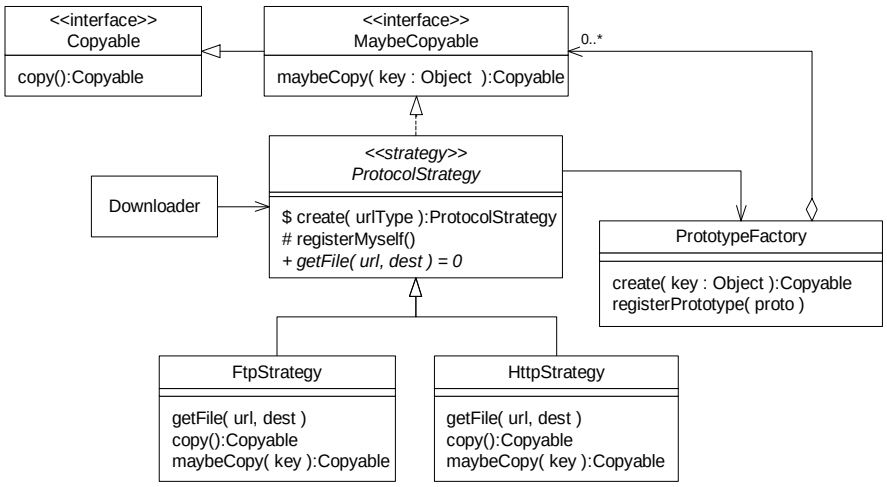
- The client knows only about the abstract class: `ProtocolStrategy`.
- `ProtocolStrategy` knows how to ask a `PrototypeFactory` to make a new strategy, given a key.
- `PrototypeFactory` knows very little about the things it makes (only that they are Copyable), and it can be configured dynamically.
- New strategy classes can easily and dynamically register themselves:

```
class ShttpStrategy extends ProtocolStrategy {
  static {
    ProtocolStrategy ps = new ShttpStrategy();
    ps.registerMyself( “shttp” );
  }
  // ...
}
```

Still More Considerations ...

- We still need information somewhere regarding the specific concrete prototype classes, and code that forces them to load, *adding themselves* to the factory's prototype registry...
- The PrototypeFactory can only make instances of classes that can be chosen by using simple `equals()` tests on the key, since this is how HashMap works. This is actually not much of a problem for the example at hand, where the protocol String ("ftp", "http", "shttp") makes a great key, but not all designs have such an obvious key.
- The Factory implicitly makes the choice of which Prototype is to be replicated. This perhaps would be better determined by the prototype objects themselves… We can make a yet more flexible Factory by using another pattern...

Use *Chain of Responsibility* Variant



How it Works

- Each subclass of `MaybeCopyable` has its own test, inside of `maybeCopy(key)` to determine if it should replicate itself. The prototype object with either return null, or call `copy()`.
- The `PrototypeFactory` asks each of its registered Prototypes to `maybeCopy(key)` until one of them does, or all have failed, in which case, null is returned.
- The key is no longer limited to be a String.
- In the GoF Chain Of Responsibility pattern, each Prototype calls the next one in the list itself; control does not pass back to the Factory between Prototypes; therefore, this example does not strictly follow the GoF's Chain Of Responsibility pattern; we might call it the "volunteer" variant.

Inside the new PrototypeFactory

```
class PrototypeFactory {
    private ArrayList protoList = new ArrayList();

    public void registerPrototype( MaybeCopyable proto ) {
       protoList.add( proto );
    }

    Copyable create( Object key ) { // no longer String
        MaybeCopyable proto = null;
        Iterator it = protoList.iterator();
        while ( it.hasNext() && proto == null ) {
          proto = (MaybeCopyable) it.next();
          proto = (MaybeCopyable) proto.maybeCopy( key );
        }
        return proto;
    }
}
```

Downloading Files Example Summary

Patterns We've Used:

- **Creational**
  - Factory
  - Prototype
- **Behavioral**
  - Strategy
  - Chain of Responsibility variant - "volunteer"
  - Iterator

Why use *Creational* Design Patterns ?

- To eliminate the use of hard-coded concrete class names at the one point where they are generally used: wherever a new object gets created.
- To delegate the choice of how to create the object to the object's class. In many designs, the class itself should decide if it wants to create a new object; or perhaps recycle a retired instance using the "object pool" design pattern; or perhaps deserialize the object from a file; or issue a database query; or perhaps the desired object is already in memory? Clients should not have to deal with this extra complexity! Hide it from them!
- To ease code maintenance by providing a mechanism whereby concrete class names do not need to be hard-coded anywhere! It is possible, for example, for a program, upon starting up, to scan the disk for any .class files where the class implements a given interface, and then load the class, and add it to the factory's prototype registry. Such concrete class names can also be specified in a textual .ini file. It is even possible for new concrete classes to be added to a 7x24 program while it remains running! For example, a message can be sent to a running server through a socket connection, where the message contains the name of a new class to load.
- To allow a program to be easily configurable to use any one of a number of "families" of different concrete classes (as in the next example).

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. Clients will use the ***Abstract Factory*** interface for creating new objects. Note that the Button class is also abstract.

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

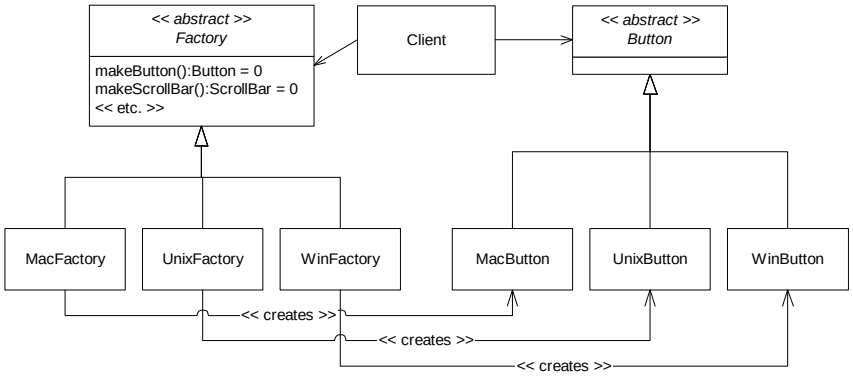
Factories are sometimes referred to as "virtual constructors"

- 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 (cont.)

- 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.
- So it is natural for us to use this innate ability while designing.

As we study design patterns…

- we learn to *recognize* patterns when we see them in existing code.
- we learn to use patterns *generatively* while creating new class structures.

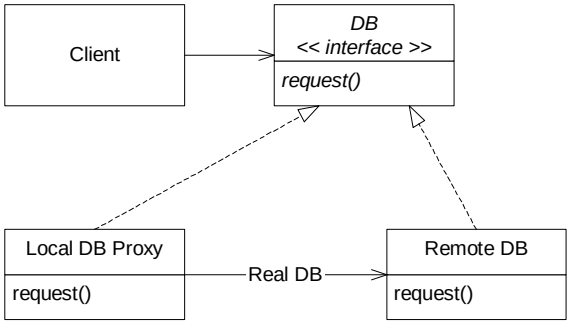
More Design Patterns Example(s)

Two design problems with "similar" solutions:

- 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?
- You have a document viewer program which may embed very large graphical images which are slow to load; but you need the viewer itself to come up very quickly when the document is selected. Therefore, images in the document 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?

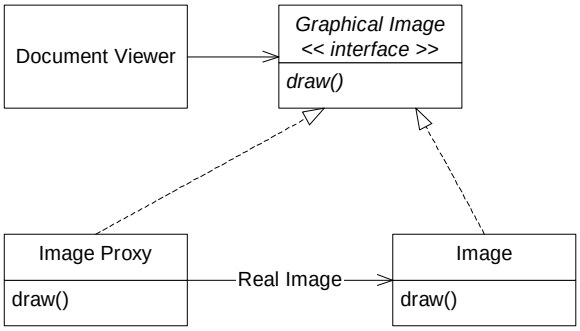
Describe one solution strategy that works for both problems.

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*

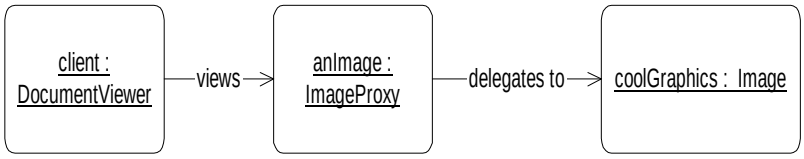
***Intent*:** Have a client object communicate with a representative of the server object, rather than with the server object directly...

***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.
- As a ***cache*** which can be shared by multiple clients.

- This is one of the most widely used patterns, for a reason!
- 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.
- The performance hit from the extra level of indirection is minimal.
- Knowledge of this pattern is essential.

Proxy (cont.)



Design Pattern: *Adapter*

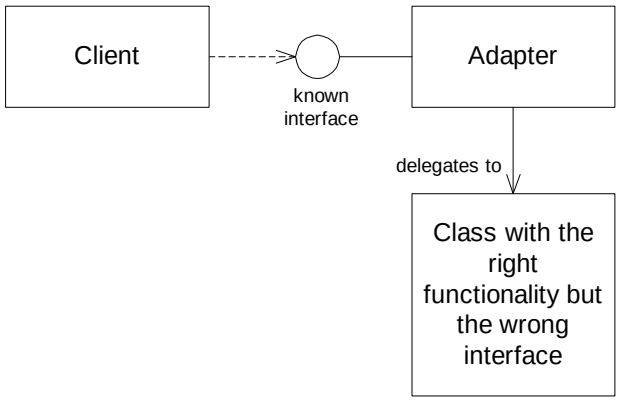
Intent: Convert the interface of a class into another interface clients expect. *Adapter* lets classes work together that couldn't otherwise because of incompatible interfaces.

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

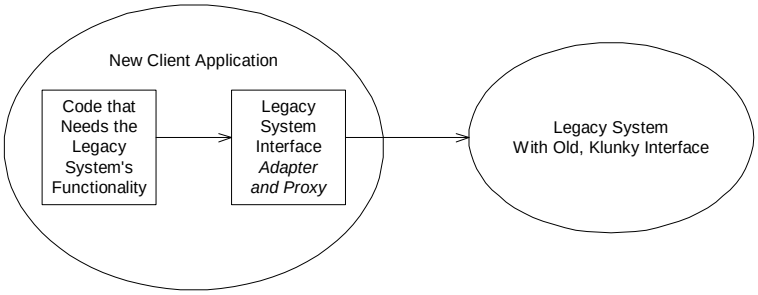
Use the *Adapter* pattern when:

- 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 that you get whenever you compile CORBA (IDL) or RMI interfaces.
- 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)…

- It would be very bad style if all of the clients of the C function that (quickly) generates a "universal ID" were to have to know about JNI. The following Java class provides the "universal ID" service at a simpler, higher level, completely encapsulating 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 in a graceful manner.

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

Adapter (cont.)

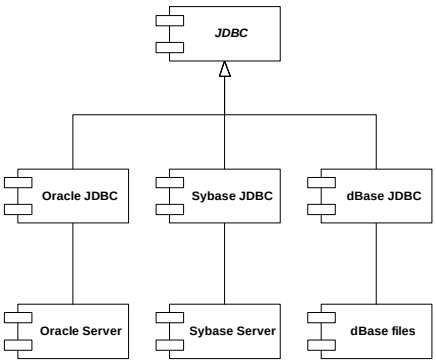
```
public class UniversalIDGenerator
{
  static // Happens once! the first time the class gets referenced
  {
    // Load the Library: Unix: libUidLib.so - Win32: UidLib.dll
    // Using Environment Variable: Unix: LD_LIBRARY_PATH - Win32: PATH
    try {
      System.loadLibrary( “UidLib” );
      Log.log( "Loaded Library: UidLib”, Severity.INFO, null );
    }
    catch( Throwable t ) {
      Log.log( "Unable to load Library: UidLib”, Severity.ERROR, t );
    }
  }
```


Adapter (cont.)

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

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

Adapter Example

- **JDBC specifies a common interface that clients expect.**
- **The JDBC "driver" for each specific database engine is an Adapter.**

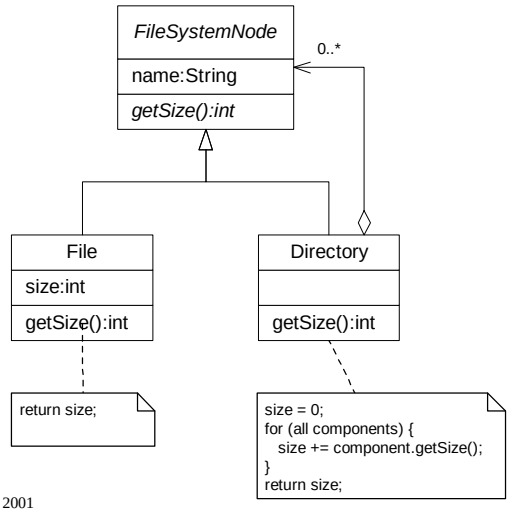


Example: New File System Feature . . .

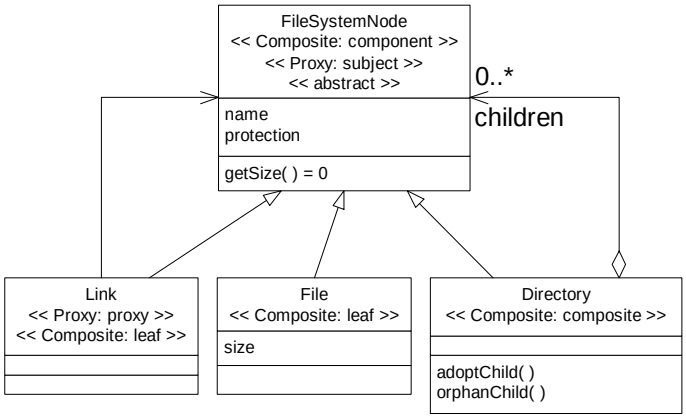
On UNIX systems, there is a feature in the File System called a Link that we wish to add to our design (like the Windows shortcut or the Macintosh alias). A Link is simply a navigational shortcut which allows a user to see a virtual copy of a file or a directory, as if it were local to the user's current 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

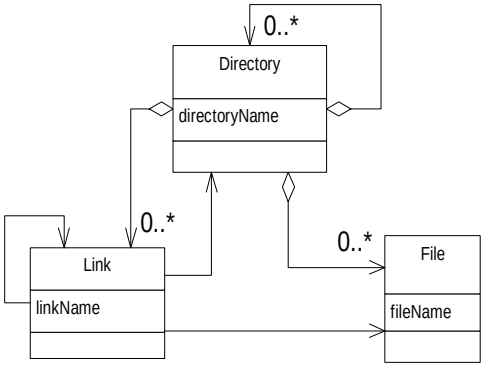


Composite & *Proxy* Together



Composite & *Proxy* Together (cont.)

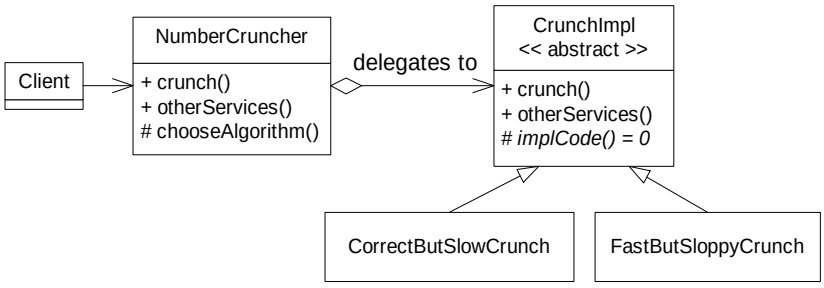
- Compare the previous slide with this "correct" design...
- An even worse "correct" design would be to have only two classes, Directory and File, with all of the Link code "hacked in" ...



Design Pattern: *Strategy*

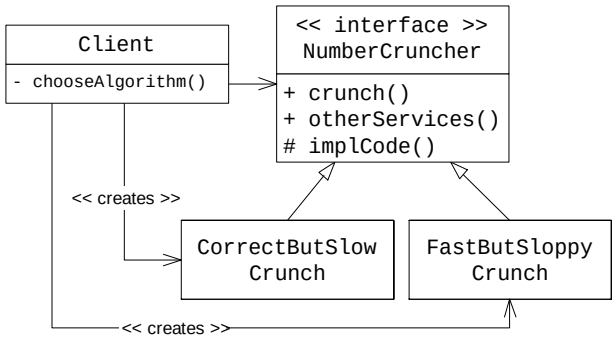
Intent: Allows multiple implementation strategies to be interchangeable, so that they can easily be swapped at run-time, and so that new strategies can be easily added.

- In this example, notice that client's of NumberCruncher do not know about the different crunch algorithms. The NumberCruncher.crunch() method is free to decide which CrunchImpl to use at any time; new algorithms can be easily added.



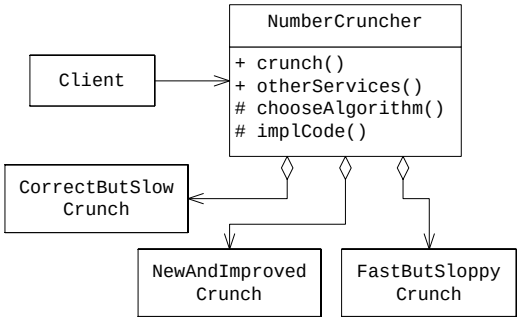
Strategy (cont.)

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



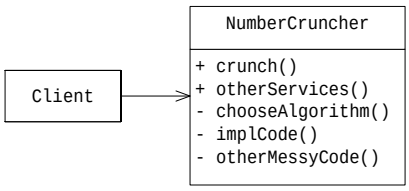
Strategy (cont.)

- Here's another "correct" design... But there can be no polymorphism in the chooseAlgorithm() or implCode() methods, leading to maintenance difficulties. 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.



Strategy (cont.)

- In this *inflexible* example, 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 an interesting state transition.
- All 3 design patterns use "delegation to an abstract class or interface."
- The difference lies in the patterns' *intents*...

Strategy (cont.)

Q: How should I travel to CU tomorrow?

1: Bicycle?
2: Bus?
3: Car?
4: Taxi?
5: Friend?
6. Hitch-hike?
7. Walk?

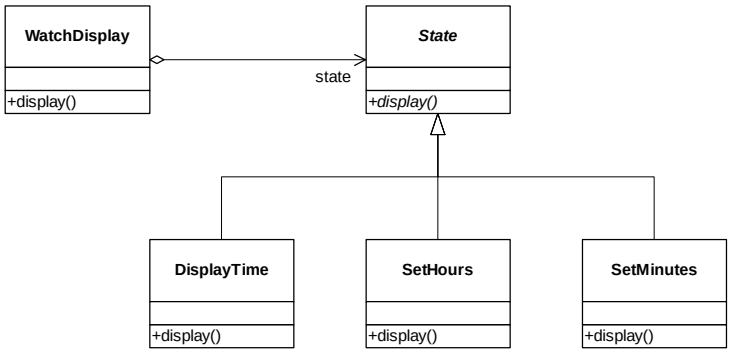
There are trade-offs in ***cost*** vs. ***time*** vs. ***convenience***…

Other examples:

- Fractal Applet Calculators
- Computer Game Players

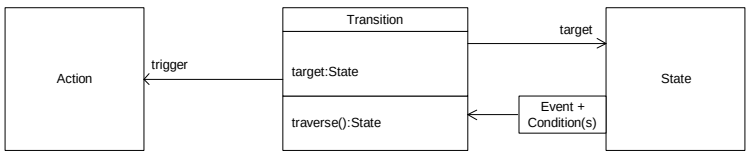
Design Pattern: *State*

Intent: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class from the point of view of the calling client (meaning: its behavior will change).



State Pattern (cont.)

- How does a *ConcreteState* know what state to go to on a transition?
  - Each class can have its own table or switch statement, or a hash table of transitions keyed by their trigger Events (and guard conditions).
  - Consider using *State*, *Action*, *Event* and *Transition* classes.
  - Note: The *Action* class might be implemented using the *Command* pattern.



Alternatives to *State* and *Strategy*

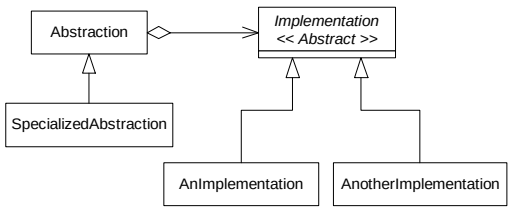
- Many "procedural programmers" tend toward designs that have lots of "decisional logic" - frequently using "switch statements." Often, the states of some "state machine" are represented in some kind of table.

- Problems with this approach:
  - Increased code complexity.
  - Changes require multiple edits to the multiple switch statements.
  - Makes the decision at run-time rather than design-time.
  - Increases decisional logic, and thus, the likelihood for the code to have bugs - polymorphism can be used instead.
  - Tables are very hard to understand by inspection.
  - Switch statements can only make decisions based on integers, so the code ends up with if - else if - else if - else if . . .

Design Pattern: *Bridge*

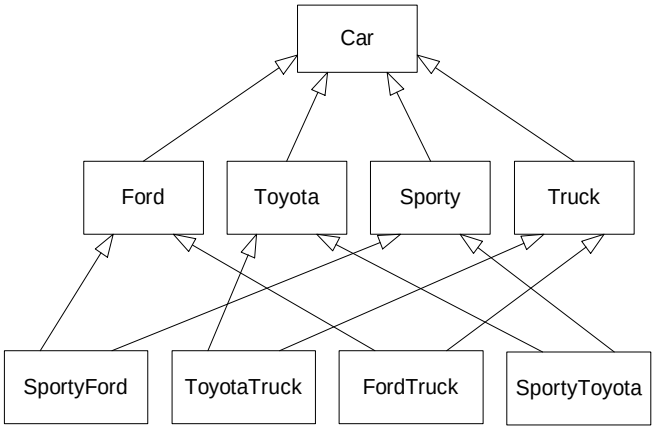
Intent: Decouple a class abstraction from its implementation.

- Like *Strategy*, *Bridge* allows the implementation to be selected at runtime.
- *Bridge* allows multiple Abstraction objects to share one instance of an Implementation object (eg: A C++ String class with copy-on-update semantics).
- *Bridge* allows separation of a "big" Abstraction class into two smaller classes, one for the "abstraction" and one for the "implementation" - so that the two may vary independently.
- Also applicable to simplify a complex class hierarchy.



Bridge Example

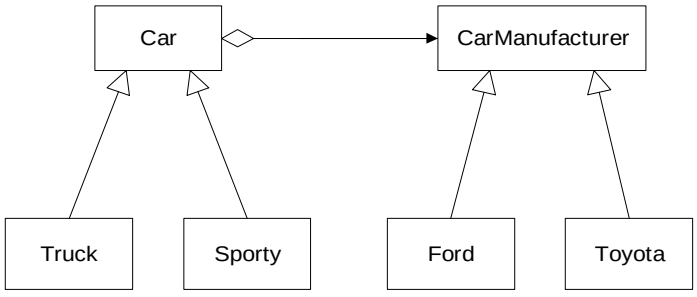
- How can we simplify this design?



Bridge Example (cont.)

Apply the ***Bridge*** Design Pattern

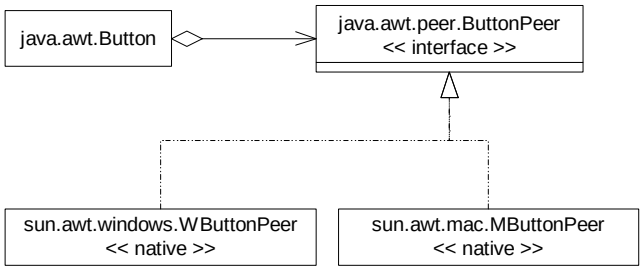
- You might use Bridge when you might otherwise be tempted to use multiple inheritance...



Bridge in the Java AWT

The Java AWT 1.1.x (pre-Swing) uses the *Bridge* pattern to separate the widget (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.



Design Pattern: *Observer*

***Also known as*:** "Publish / Subscribe," "Model / View," and "Source / Listener."

***Motivation*:** Two File Managers are both observing the same Directory; the user deletes a subdirectory using File Manager A; we want File Manager B to immediately and automatically get updated, reflecting the change...

***Applicability*:**

- When there are multiple views of a model (subject) that need to stay in sync. No view should know about any other.
- When an object needs to communicate to other objects of unknown type (but known *Observer* interface) it can notify them.

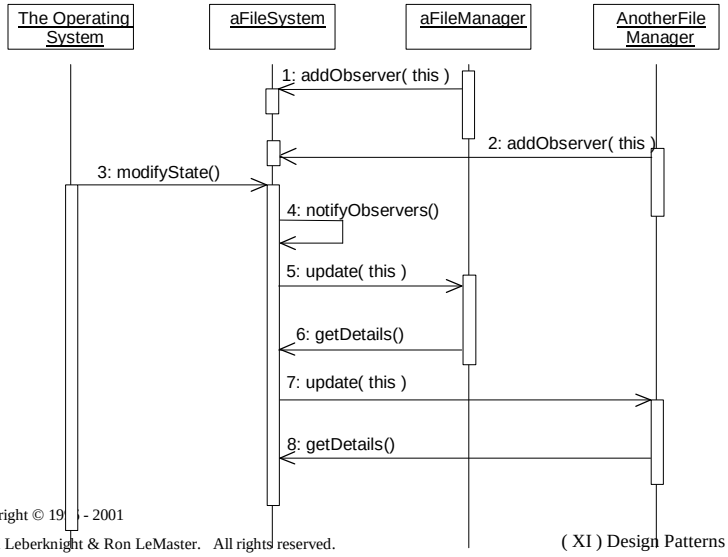
***Pros*:**

- Promotes loose coupling between objects.
- Excellent communication protocol.
- Avoids polling

***Cons*:**

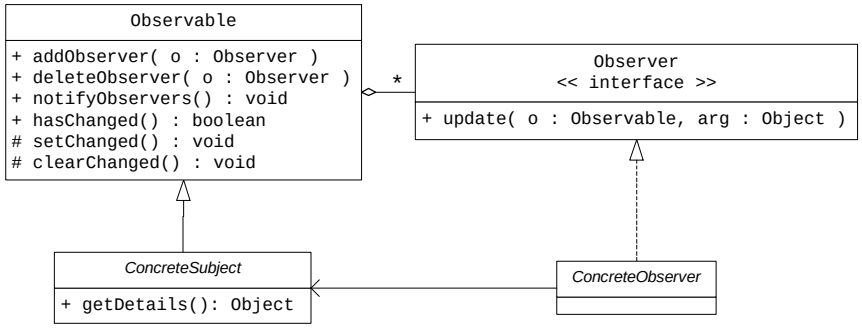
- None. Knowledge of this pattern is essential.

Observer Pattern (cont.)

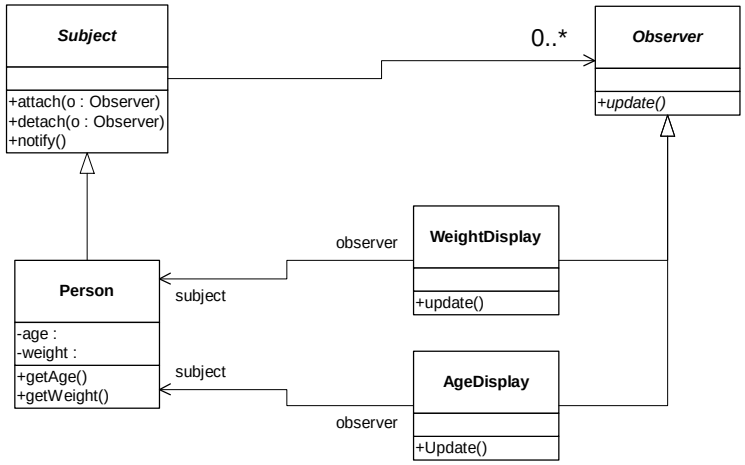


Java Support for *Observer*

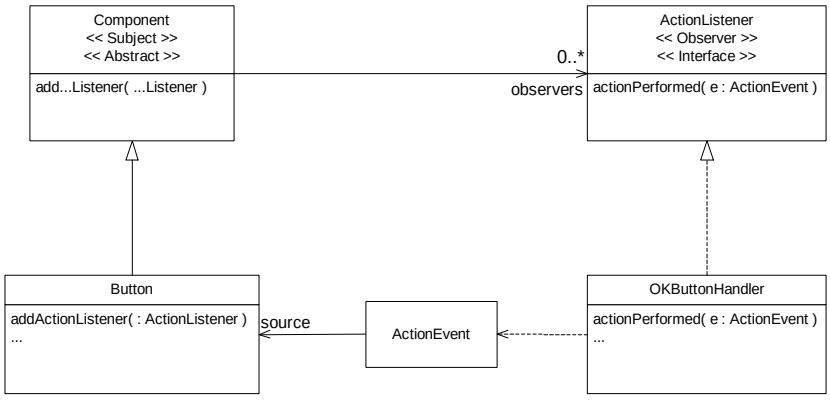
- The java.util package provides an Observable class and an *Observer* interface:



Example: A GUI Observes a Person



Java AWT 1.1 Event Model



- **See example code on next slide...**

Java AWT 1.1 Event Model (cont.)

```
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();
     }
  }
  // Any mouse click event over the OK Button will invoke the
  // OKButtonHandler.actionPerformed() method.
```


Design Pattern: *Null Object*

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

- Not one of the GoF patterns.

```
interface ILog {                    // Code to demonstrate
  public void log( String msg );    // a Null Object ...
}
class FileLog implements ILog {
  public void log( String msg ) { …; }
}
class ScreenLog implements ILog {
  public void log( String msg )  {
    System.out.println( msg );
  }
}
class DBLog implements ILog { …; }
```

Null Object (cont.)

```
class Client
{
  private ILog il = null; // initialized elsewhere
  // ...
  public void code( )
  {
    if( il != null ) il.log( "1" ); // conditionals required !
    // ...
    if( il != null ) il.log( "2" );
  }
}
```

- How can the Client's code() method be simplified using a "*Null-Object*"?

Null Object (cont.)

```
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: *Stubbed-out Subsystem*

***Intent*:** Provide a trivial (or "null") 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 *Stub* might provide a fake, hard-coded, or trivial version of the service in question.
- Sometimes the term *Stub* is also used to describe a local *Proxy* object that trivially delegates to a remote system.

Very common example: A stubbed-out database…

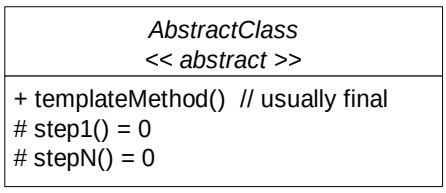
This is useful at the beginning of a project when the real database is still under construction. If the persistence layer design uses a DbProxy as the desired interface for the remote DB, then there can be a DbProxyStub class that returns hard-coded results for all queries. For example, a high-level method on a Customer class might be to get the customer's current account balance. The Customer delegates this query to the data/persistence layer, which simply always returns $1000 (or whatever).

This kind of Stub is 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.

- Very common in code which uses polymorphism.
- The templateMethod() is designed to have subclasses which implement certain steps of the process.
- Example: The **`BinaryOperator.evaluate()`** method in the Composite Expression example.



Template Method Example

```
abstract class ProcessManager extends java.util.Observable {
  protected final void process( ) {
    try {
      initProcess( );
      doProcess( );
      setChanged( );
      notifyObservers( ); // Tell my Observers my state changed.
    }
    catch( Throwable t ) {
      Log.log( "ProcessManager.process():", Severity.ERROR, t );
    }
  }
  abstract protected void initProcess( );
  abstract protected void doProcess( );
}
```


Design Pattern: *Command*

Intent: Allows requests to be modeled as *Command* objects, so that they can be treated uniformly, queued, etc... Also, we wish for each concrete *Command* to be responsible for creating its own undo *Command*.

- The Command pattern is also very useful in distributed applications, where the client sends a command to the server.

Example: Cut / Paste ...

- 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.

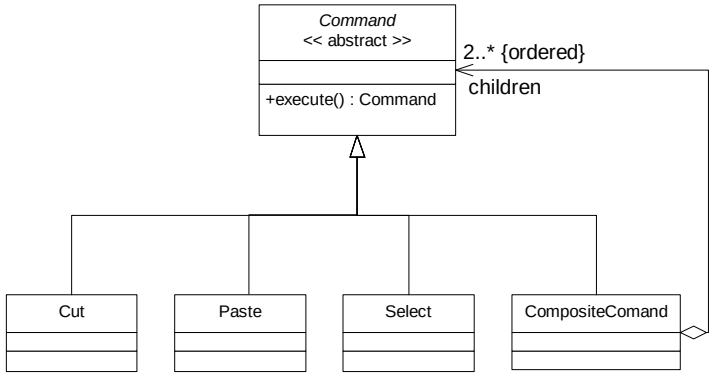
The abstract *Command* class has an execute() method that returns the *Command's* UNDO *Command*. Thus, CutCommand's execute() method must return a new *Command* which has the following functionality:

1) PASTE the previously CUT text.

2) SELECT the previously CUT text.

Note: Cut's Undo *Command* is a *Composite* of 2 other *Commands*, paste & select.

Command (cont.)

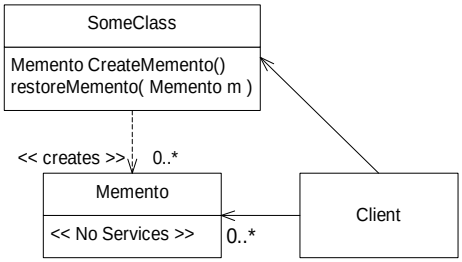


Design Pattern: *Memento*

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

***Applicability*:** The state of an object must be saved (by a client) so that it can be restored later. The *Memento* object contains all 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, in Java & C++, for methods & data in the class *Memento* to be available to *SomeClass*, but not to *Clients*?



More and More Patterns ...

Domain Analysis, Architecture, Design, Project Organization, Development Process, Implementation (language idioms, memory management, algorithms), Anti-Patterns, ...

Patterns Home Page: **http://hillside.net/patterns/patterns.html**
WikiWikiWeb Patterns Project: **http://c2.com/cgi/wiki?WelcomeVisitors**

Design Patterns: Elements of Reusable Object-Oriented Software, by Gamma et al., Addison-Wesley, 1994 <ISBN 0-201-63361-2>

Pattern-Oriented Software Architecture - A System of Patterns by Buschman, et al., Wiley, 1996 <ISBN 0-471-95869-7>

Advanced C++ - Programming Styles and Idioms, by Coplien, Addison-Wesley, 1992

- Seminal work on design patterns and C++ idioms <ISBN 0-201-54855-0>

Pattern Language of Program Design. Various authors, Addison-Wesley

- An evolving series of edited volumes containing patterns in many different domains.

Vocabulary

- Gang of Four (GoF)
- Pattern Language
- Intent
- Context
- Forces
- QWAN (Quality Without a Name)
- *GoF: Adapter (Wrapper), Bridge, Composite, Decorator, Façade, Flyweight, Proxy, Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor, Abstract Factory, Builder, Factory Method, Prototype, Singleton*
- "Virtual Constructor" (factory), Null Object, Stub, Model-View-Controller
- JNI (Java Native Interface)