Intro to Software
“Design Patterns”
Design Patterns

- Focus of a great deal of current attention in software development research and practice
What started the idea

Two books on *architecture* (not software) by Christopher Alexander, et al.
and then came ...

(aka “Gang of four”)

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Uses earlier Booch Notation, rather than UML
Definition of a Pattern

- a solution to a recurrent problem
- not a "concrete" solution, but an abstract version of it
- four essential elements:
  - Name of the Pattern
  - The Problem
  - The Solution
  - Consequences, tradeoffs
Uses of Patterns

- Conversational “handle” on which to hang ideas and concepts
- To direct the developer to a known solution for a kind of problem
- To help focus a design
- As a vehicle for refining solution techniques
Patterns help to:

- solve specific design problems
- reduce the need for redesign
- provide reusable solutions
- act as templates
- pass on knowledge from experts to novices
Patterns are not ...

- Classes
- Libraries
- Packages
- Macros
- Higher-order functions
- Template classes

However, some of these could conceivably capture some design patterns.
Examples of Patterns

- “Design Patterns” book lists 23 patterns, in several categories. These are sometimes annotated GoF (“Gang of Four”).

- GRASP Patterns in Larman book.

- Others have contributed many additional patterns.
Example: Composite Pattern

- **Name:** Composite
- **The Problem:** Construct a class of objects wherein
  - Objects can be indivisible *or* have multiple components.
  - A collection of components can be treated as a *single* object by a client.
- **The Solution:** The one shown in the diagram
- **Consequences, tradeoffs**
UML for Composite
Non-Recursive Composite Pattern UML
Examples of Composite Patterns

- Drawing program shape: rectangle, oval, **group**: set of shapes

- File systems, files, directories, links

- S expressions
Tradeoffs in Composite Pattern

- Whether *recursive* structure is needed, or will “flat” structure suffice
- Whether *ordering* of components is significant
- Whether components refer to *parents*
- Whether components can be *shared*
- Who should *delete* the children
- Whether children are represented as a *list* or simply enumerated
- Data structure issues (whether to use struct, array, linked list, etc.)
Patterns to be Discussed (alphabetized)

- Adapter
- Cache
- Command
- Composite
- Decorator
- Delegation
- Façade
- Interface
- Iterator
- Memento
- Model-View-Controller
- Observer
- Proxy
- Singleton
- State
- Stream
- Visitor
 Iterator Pattern

- (GoF, p 257) aka Cursor Pattern
- provides a way to enumerate the elements in a container without exposing the internal structure of the implementation.
- There can be multiple Iterators on a given container.
- Examples:
  - Java: Enumeration and Iterator interfaces
  - C++: Standard Library, there are for many iterator template classes.
C++ Iterator Concepts

- Trivial Iterator
- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator

See www.sgi.com/tech/stl/Iterators.html
C++ Iterator Types

- `istream_iterator`
- `ostream_iterator`
- `reverse_iterator`
- `reverse_bidirectional_iterator`
- `insert_iterator`
- `front_insert_iterator`
- `back_insert_iterator`

- `input_iterator`
- `output_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`
What Every Iterator Needs

- A way to establish a specific element, such as the “first” element
- A way to access the current element
- A way to move on to the “next” element
- A way to indicate that there are no more elements
Iterator Pattern UML, with Interfaces

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>first()</td>
<td>createlIterator()</td>
</tr>
<tr>
<td>next()</td>
<td></td>
</tr>
<tr>
<td>current()</td>
<td></td>
</tr>
<tr>
<td>done()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>anIterator</th>
<th>aContainer</th>
</tr>
</thead>
<tbody>
<tr>
<td>first()</td>
<td>createlIterator()</td>
</tr>
<tr>
<td>next()</td>
<td></td>
</tr>
<tr>
<td>current()</td>
<td></td>
</tr>
<tr>
<td>done()</td>
<td></td>
</tr>
</tbody>
</table>

Interfaces

Implementation
Iterator Patterns in Java: Enumeration Interface

- Informal Methods:
  - creator
  - first()
  - next()
  - !done()
  - currentItem()

- Java realization:
  - Enumeration e = container.elements();
    (implied in initialization)
  - nextElement()
  - hasMoreElements()
  - none- use result of nextElement()
Iterator Patterns in Java:

**Iterator Interface**

- **Informal Methods:**
  - creator
  - first()
  - next()
  - !done()
  - currentItem()

- **Java realization:**
  - Iterator e = linkedList.iterator();
  - (none)
  - next()
  - hasNext()
  - (none - save result of next ())
Iterator Pattern in C++ STL

- **General Methods:**
  - first()
  - next()
  - done()
  - current()

- **C++:**
  ```
  Container<Type>::Iterator myIterator;
  ```

  ```
  myIterator = container.begin();
  ```

  ```
  myIterator++;
  ```

  ```
  myIterator == container.end();
  ```

  ```
  *myIterator
  ```

- **Syntax is that of a pointer**
Exercise

- Consider a composite that structures its elements as directed, ordered, tree.

- What kinds of iterators would you propose?

- What are the methods on each iterator?
Visitor Pattern

- (GoF, p 331)

- Similar to the Iterator pattern, except that rather than passing objects outside during the enumeration, a Visitor object is passed into the Container.

- The Visitor works on the objects while inside the Container.

- **Purpose**: Can add new operations on a structure without knowing structure details.
Similar to **map** in Functional Programming

- `map(F, L)` maps a function over a list.

- There is no explicit extraction of the list elements outside of the map.

- The function does not need to know how the list is structured.
Visitor Pattern UML

Visitor
- next()
- visit()
- done()

Container
- acceptVisitor()

aVisitor<T>
- next()
- visit(T)
- done()

aContainer<T>
- acceptVisitor()

Interface

Implementation
Visitor in C++

- class Container<Type>
  ```
  {
    acceptVisitor(Visitor<Type>);
    ....
  }
  ```

- class Visitor<Type>
  ```
  {
    visit(<Type> Thing); // Do something to Thing
    next(); // Advance to the next.
    boolean done(); // Tell if no more things.
  }
  ```
Façade Pattern

- (GoF, p 185)
- An entire sub-system or set of classes, etc. is given a **single simple interface** in order to
  - shield the user against the internal complexity of how the classes are used together
  - bundle together less-coupled components
Façade Pattern example

VisualTool

drawShape()
addText()

Shape

<table>
<thead>
<tr>
<th>Position</th>
<th>Size</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>setSize</td>
<td>getSize</td>
<td>setColor</td>
</tr>
<tr>
<td>draw</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TextEditor

<table>
<thead>
<tr>
<th>Position</th>
<th>Size</th>
<th>Font</th>
</tr>
</thead>
<tbody>
<tr>
<td>setSize</td>
<td>getSize</td>
<td>setFont</td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

VisualTool

- drawShape()
- addText()
In building using a façade pattern, it is important that the individual components not depend on the façade itself.

This would introduce cross-coupling, which is undesirable.

When the façade is “removed”, the components should “fall apart” as their original, uncoupled or loosely-coupled, entities.
Rather than Make the Façade a Single Class, A Package can also be used

- A package is a group of related classes.
- Packages can have sub-packages.
- Visibility can be controlled
Brief Digression on Packages
Packages/Dependency notation in UML
Coupling / Cohesion Terminology

- Two packages (or classes, for that matter) between which there is a high-degree of interplay are said to be **strongly-coupled**.

- Strong coupling is considered undesirable; **loosely-coupled** is better for packages and classes.
Coupling / Cohesion Terminology

- A set of methods for a class, or classes in a package, are said to be **cohesive** (or “coherent”) if they provide aspects of a uniform model for dealing with the objects / classes.

- Cohesiveness is desirable; it is the mark of a careful design.
Coupling / Cohesion Summary

- Coupling is “bad”.
- Cohesion (coherence) is “good”.

See also: GRASP (Larman):
  - Low-coupling pattern
  - High-cohesion pattern
About Packages

- **Innermost** packages contain classes
- **Cyclic dependencies** among packages should be avoided: break them, or combine into a single package.
- Packages can help delineate work-breakdown among teams, and can thus serve as a management device
- Consider Java *package* construct.
- Equivalent in C++ is *namespaces*
A Portion of a Java Package Hierarchy

java.net
java.rmi
  java.rmi.activation
  java.rmi.dgc
  java.rmi.registry
java.rmi.server
java.security
  java.security.acl
  java.security.cert
  java.security.interfaces
  java.security.spec
Namespaces in C++

- Without Namespaces, there is one big flat name space.
- The danger is conflicting names in different modules and name space “pollution” by modules with lots of names.
- Namespace construct allows structuring into an arbitrary number of hierarchical levels.
C++ Namespace Example

RT

DB

two parallel namespaces

OODB
object-oriented databases

RDB
relational databases

realtime databases

sub-namespaces
C++ Namespace Example

Short-cut rules can lead to confusion if used.
Returning to Patterns
Decorator Pattern

- (GoF, p 175)
- aka Wrapper (one version)
- Enclose an object of one class in another class that “decorates” the original objects (e.g. scrollbars or a border around a window).
- Examples: streams of various types (OutputStream, FileOutputStream, PrintStream, ...)
At least three possibilities:

- The decorating class inherits from the decorated class ("direct" decoration).
- The decorating class aggregates or composes a member of the decorated class ("decoration by delegation").
- A third class aggregates or composes both the decorated class and the decoration.
Adapter Pattern

- (GoF, p 139)
- aka Binding, Wrapper
- Adapts one or more existing APIs to fit another API specification (one that clients expect).
- (API = Application Programming Interface)
An interface defines a **Stack**.

A dynamic array implementation defines an **Array**.

An adapter is an **ArrayStack**, i.e. a class satisfying the stack interface, implemented using an array.

The roles of the Stack and Array classes are not symmetric.
Adapter Pattern based on Association

In C++, multiple inheritance would be used.
public class ServicesEnumeration {
    implements Enumeration {
        public boolean hasMoreElements() {
            return this.currentServiceIdx <= adaptee.numServices();
        }
        public Object nextElement() {
            if (!this.hasMoreElements()) {
                throw new NoSuchElementException();
            }
            return adaptee.getService(this.currentServiceIdx++);
        }
    }
}
Bridge Pattern

- (GoF, p 151)
- aka “Handle/Body” or “Interface” pattern
- Abstract an API by providing an interface class, with the intention by design of providing different implementations for the interface.
- Example: Java awt (abstract window toolkit) vs. M/S Windows or MacOS Windows (called “peer” classes)
Bridge Pattern UML

Client

Abstraction

Operation()

Optional

Reflined Abstraction 1

Reflined Abstraction 2

Concrete Implementor A

Concrete Implementor B

optional

Implementor

OperationImpl()
Using a Bridge

- Example: Interface to a component that is incomplete, not yet known or unavailable during testing

- JAMES Project (CMU): if seat (for vehicle) data is required to be read, but the seat is not yet implemented, not yet known or only available by a simulation, provide a bridge:
public interface SeatImplementation {
    public int GetPosition();
    public void SetPosition(int newPosition);
}

public class AimSeat implements SeatImplementation {
    public int GetPosition() {
        // actual call to the AIM simulation system
    }
    ...
}

public class SARTSeat implements SeatImplementation {
    public int GetPosition() {
        // actual call to the SART seat simulator
    }
    ...
}
Adapter vs. Bridge

- Adapter and Bridge are similar:
  - **Adapter**: Adapts existing classes to an expected interface. The interface and classes exist; the new thing is the adapter, a go-between class.
  - **Bridge**: Creates an abstract Interface to be implemented by multiple classes; keeps the implementations separate from the interface. The new thing is the interface.
Adapter vs. Bridge

- The adapter pattern is geared towards making unrelated components work together
  - Applied to systems after they’re designed (reengineering, interface engineering).

- A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.
  - Ab initio engineering of an “extensible system”
  - New “beasts” can be added to the “object zoo”, even if these are not known at analysis or system design time.
C++ Bridge Pattern

- class Stack<Item>
  {
    virtual void push(Item item) = 0;
    virtual Item pop() = 0;
  }

class ArrayStack<Item> : public Stack<Item>
{
    Item array[ ];
}

Interface definition

An implementation of the interface
Another implementation of the interface

- class ListStack<Item> : public Stack<Item>
  {
    list<Item> List;
  }
Singleton Pattern

- (GoF, p 127)
- A class with exactly one instance (or a class used only for its static members).
- Implements a global access point.
- Still use a class
  - Globals are still not a good idea
  - May wish to use multiple instances in the future.