What is a pattern?

- Patterns = problem/solution pairs in context
- Patterns facilitate reuse of successful software architectures and design
- Not code reuse
  - Instead, solution/strategy reuse
  - Sometimes, interface reuse

Gang of Four

- The book that started it all
- Community refers to authors as the “Gang of Four”
- Figures and some text in these slides come from book

Some design patterns you already know

- Iterator
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Observer
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Proxy
  - Provide a surrogate or placeholder for another object to control access to it.

Iterator pattern

Uses of Iterator Pattern
You’ve already seen this in Java

- Iterators for Collections
  - Also, Enumerators for Hashtables and Vectors

Observer pattern

Use of an Observer pattern

Implementation details

- Observing more than one subject.
  - It might make sense in some situations for an observer to depend on more than one subject. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.
  - Making sure Subject state is self-consistent before notification.

More Implementation Issues

- Implementations of the Observer pattern often have the subject broadcast additional information about the change.
  - At one extreme, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.

- You can extend the subject’s registration interface to allow registering observers only for specific events of interest.

You’ve seen this before

- The standard Java and JavaBean event model is an example of an observer pattern.
Known uses (from DP book)

- McCullough [McC87] discusses using proxies in Smalltalk to access remote objects. Pascoe [Pas86] describes how to provide side-effects on method calls and access control with "Encapsulators."
- NEXTSTEP [Add94] uses proxies (instances of class NXProxy) as local representatives for objects that may be distributed.
  - On receiving a message, the proxy encodes it along with its arguments and then forwards the encoded message to the remote subject. Similarly, the subject encodes any return results and sends them back to the NXProxy object.

Creation patterns

- Singleton
  - Ensure a class only has one instance, and provide a global point of access to it.
- Abstract Factory
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Structural patterns

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

Behavioral patterns

- State
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Visitor
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
Singleton pattern

• It's important for some classes to have exactly one instance
  – Many printers, but only one print spooler
  – One file system
  – One window manager
• Such designs can be limiting

The Singleton solution

• Make the class itself responsible for keeping track of its sole instance.
• Make constructor private
• Provide static method/field to allow access to the only instance of the class

Implementing the Singleton method

• In Java, just define a final static field
  – Class Singleton {
    private Singleton() {...}
    final static Singleton instance = new Singleton();
    ...
  }
• Java semantics guarantee object is created immediately before first use

Implementing the Singleton Method in C++

• C++ static initialization is very limited and ill-defined
• Use instance() method
• Note: not thread safe
  – Java method is thread safe

Implementing the Singleton Method in C++

• Class Singleton {
  private:
    Singleton() {...}
    static Singleton _instance = 0;
  public:
    static Singleton instance() {
      if (_instance == 0)
        _instance = new Singleton();
    return _instance;
    }
  ...
}

Abstract Factory

• Different look-and-feels define different appearances and behaviors for user interface “widgets” like scroll bars, windows, and buttons.
Using an abstract factory

- Get a reference to a type WidgetFactory
  - To an object of the appropriate subtype of WidgetFactory
  - Ask the WidgetFactory for a scrollbar, or for a window

Adapter pattern

- Clients needs a target that implements one interface

JDK 1.3 Proxy class

- Can be used for both Proxy and Adapter pattern
- Use java.lang.reflect.Proxy
- Create object that implements a set of interfaces
  - Specified dynamically
  - Invocation handler handles calls

State pattern

- Suppose an object is always in one of several known states
- The state an object is in determines the behavior of several methods
- Could use if/case statements in each method
- Better solution: state pattern

State pattern

- Have a reference to a state object
  -Normally, state object doesn’t contain any fields
  -Change state: change state object
  -Methods delegate to state object

Structure of State pattern
Instance of State Pattern

State pattern notes

- Can use singletons for instances of each state class
  - State objects don’t encapsulate state, so can be shared
- Easy to add new states
  - New states can extend other states
  - Override only selected functions

Example – Finite State Machine

```
class FSM {
    State state;
    public FSM(State s) { state = s; }
    public void move(char c) { state = state.move(c); }
    public boolean accept() { return state.accept(); }
}
```

```
public interface State {
    State move(char c);
    boolean accept();
}
```

```
class State1 implements State {
    static State1 instance = new State1();
    private State1() {}
    public State move (char c) {
        switch (c) {
            case 'a': return State2.instance;
            case 'b': return State1.instance;
            default: throw new IllegalArgumentException();
        }
    }
    public boolean accept() { return false; }
}
```

```
class State2 implements State {
    static State2 instance = new State2();
    private State2() {}
    public State move (char c) {
        switch (c) {
            case 'a': return State1.instance;
            case 'b': return State1.instance;
            default: throw new IllegalArgumentException();
        }
    }
    public boolean accept() { return true; }
}
```

Visitor pattern

- A visitor encapsulates the operations to be performed on an entire structure
  - E.g., all elements of a parse tree
- Allows the operations to be specified separately from the structure
  - But doesn’t require putting all of the structure traversal code into each visitor/operation

Not a visitor
Double-dispatch

- Accept code is always trivial
  - Just dynamic dispatch on argument, with runtime type of structure node taking into account in method name
- A way of doing double-dispatch
  - Dynamic dispatching on the run-time types of two arguments
  - Visitor code invoked depends on run-time type of both visitor and node

Using overloading in a visitor

- You can name all of the visitXXX(XXX x) methods just visit(XXX x)
  - Calls to Visit(AssignmentNode n) and Visit(VariableRefNode n) distinguished by compile-time overload resolution

Easy to provide default behavior

- Default
  - visit(BinaryPlusOperatorNode)
  - can just forward call to visit(BinaryOperatorNode)
- Visitor can just provide implementation of visit(BinaryOperatorNode) if it doesn’t care what type of binary operator node it is at

State in a visitor pattern

- A visitor can contain state
  - E.g., the results of parsing the program so far
- Or use stateless visitors and pass around a separate state object

Traversals

- In the standard visitor pattern, the visitor at a node is responsible for visiting the components (i.e., children) of that node
  - if that is what is desired
    - Visitors can be applied to flat object structures
- Several other solutions
  - acceptAndTraverse methods
  - Visit/process methods
  - traversal visitors applying an operational visitor
Reactor Pattern

Intent

• The Reactor architectural pattern allows event-driven applications to
  – demultiplex and dispatch service requests
  – delivered to an application from one or more clients
• Also known as
  – Dispatcher, Notifier

Forces

• A good solution must resolve the following forces:
  – To enhance scalability and minimize latency, an application must not block indefinitely waiting on any single source
  – To maximize throughput, unnecessary utilization of the CPU(s) should be avoided
  – Minimal modifications and maintenance effort should be required to integrate new or enhanced services

Solution

• Synchronously await indication events on one or more event sources
• Integrate the mechanisms that demultiplex events and dispatch them to processing elements
• Decouple these from application-specific processing elements

Applicability

• Use Reactor Pattern when a process is event-driven and:
  • Threading leads to poor performance
  • Threading requires complex concurrency control
  • OS does not support multiple threading within a process
**Consequences**

- **Benefits:**
  - Separation of concerns
  - Reusability
  - Modularity, reusability, and configurability
  - Portability
  - Coarse-grained concurrency control

- **Liabilities:**
  - Restricted applicability
  - Non-preemptive
  - Hard to debug

**Why patterns?**

- Sometimes, patterns are just a cool idea you might not have come up with on your own
  - Learn from others
- If you work with code written by someone else
  - If they use a pattern you know, it will be easier for you to understand their code
- Some frameworks can automatically build/manipulate certain design patterns
Pattern hype

- Patterns get a lot of hype and fanatical believers
  - We are going to have a design pattern reading group, and this week we are going to discuss the Singleton Pattern!
- Patterns are sometimes wrong (e.g., double-checked locking) or inappropriate for a particular language or environment
  - Patterns developed for C++ can have very different solutions in Smalltalk or Java

The End

More on visitor traversals

- accept method could be responsible for traversing children
  - Assumes all visitors have same traversal pattern
    - E.g., visit all nodes in pre-order traversal
  - Could provide previst and postvisit methods to allow for more complicated traversal patterns
    - Still visit every node
    - Can’t do out of order traversal
    - In-order traversal requires inVisit method

acceptAndTraverse methods

- Can have two parallel sets of methods in visitors
  - Visit methods
  - Process methods
- Visit method on a node:
  - Calls process method of visitor, passing node as an argument
  - Calls accept on all children of the node (passing the visitor as an argument)

Accept and traverse

- Class BinaryPlusOperatorNode {
  void accept(Visitor v) {
    v->visit(this);
    lhs->accept(v);
    rhs->accept(v);
  }
  ...
}

Visitor/process methods
Preorder visitor

• Class PreorderVisitor {
  void visit(BinaryPlusOperatorNode n) {
    process(n);
    n->lhs->accept(this);
    n->rhs->accept(this);
  }
  ...

Visit/process, continued

• Can define a PreorderVisitor
  – Extend it, and just redefine process method
  – Except for the few cases where something other than preorder traversal is required
• Can define other traversal visitors as well
  – E.g., PostOrderVisitor

Traversal visitors applying an operational visitor

• Define a Preorder traversal visitor
  – Takes an operational visitor as an argument when created
• Perform preorder traversal of structure
  – At each node
    • Have node accept operational visitor
    • Have each child accept traversal visitor

PreorderVisitor with payload

• Class PreorderVisitor {
  Visitor payload;
  void visit(BinaryPlusOperatorNode n) {
    payload->visit(n);
    n->lhs->accept(this);
    n->rhs->accept(this);
  }
  ...
}