Administrivia

- Make sure you keep up on the reading. See the lecture webpage for more info
- Project 2 due Sept. 29th
Inner Classes

• Classes can be nested inside other classes
  – These are called inner classes

• Within a class that contains an inner class, you can use the inner class just like any other class

Example: The Queue Class

class Queue<Element> {
    class Entry { // Java inner class
        Element elt; Entry next;
        Entry(Element i) { elt = i; next = null; }
    }

    Entry theQueue; // Front of the Queue

    void enqueue(Element e) {
        if (theQueue == null) theQueue = new Entry(e);
        else {
            Entry last = theQueue;
            while (last.next != null) last = last.next;
            last.next = new Entry(e);
        }
    }
    ...
}
Example: The Queue Class (cont’d)

class Queue<Element> {
  ...
  Element dequeue() throws EmptyQueueException {
    if (theQueue == null)
      throw new EmptyQueueException();
    Element e = theQueue.elt;
    theQueue = theQueue.next;
    return e;
  }
}

Referring to Outer Class

class Queue<Element> {
  ...
  int numEntries;
  class Entry {
    Element elt; Entry next;
    Entry(Element i) { elt = i; next = null;
      numEntries++; }
  }
}

• Each inner “object” has an implicit reference to
  the outer “object” whose method created it
  – Can refer to fields directly, or use outer class name.
Anonymous Inner Classes

```java
(new Thread() {
    public void run() {
        try {
            Thread.sleep(1000*60*20);
            System.out.println("...");
            System.exit(1);
        } catch (Exception e) {}
    }
}).start();
```

- Create anonymous subclass of thread, and invoke method on it

Other Features of Inner Classes

- Outside of the outer class, use `outer.inner` notation to refer to type of inner class
  - E.g., `Queue.Entry`
- An inner class marked `static` does not have a reference to outer class
  - Can’t refer to instance variables of outer class
  - Must also use `outer.inner` notation to refer to inner class
Compiling Inner Classes

- The JVM doesn’t know about inner classes
  - Compiled away, similar to generics
  - Inner class Foo of outer class A produces A$Foo.class
  - Anonymous inner class of outer class A produces A$1.class

- Why are inner classes useful?

Iteration

- Goal: Loop through all objects in an aggregate

```java
class Node { Element elt; Node next; }
Node n = ...;
while (n != null) { ...; n = n.next; }
```

- Problems:
  - Depends on implementation details
  - Varies from one aggregate to another
Iterators in Java

```java
public interface Iterator {
    // returns true if the iteration has more elts
    public boolean hasNext();

    // returns the next element in the iteration
    public Object next() throws NoSuchElementException;
}
```

- Implementation of aggregate not exposed
- Generic for wide variety of aggregates
- Supports multiple traversal strategies

Generic Iterators in Java 1.5

```java
public interface Iterator<A> {
    // returns true if the iteration has more elts
    public boolean hasNext();

    // returns the next element in the iteration
    public A next() throws NoSuchElementException;
}
```
Using Iterators

```java
Iterator<Element> i = c.iterator();
while (i.hasNext()) {
    Element e = i.next();
    // do stuff with e
}
// or use for loop
for (Iterator<Element> i = c.iterate(); i.hasNext(); ) {
    Element e = i.next();
    // do stuff with e
}
// or use for statement
for (Element e : c) {
    // do stuff with e
}
```

Iterators and Queues

- Recall queue example from beginning of lecture
- We’ll explore options for adding iterators
next() Shouldn’t Mutate Aggregate

class Queue<Element> {
  ...
  class QueueIterator implements Iterator<Element> {
    Entry rest;
    QueueIterator(Entry q) { rest = q; }
    boolean hasNext() { return rest != null; }
    Element next() throws NoSuchElementException {
      if (rest == null)
        throw new NoSuchElementException();
      Entry e = rest.elt;
      rest = rest.next; // queue data intact
      return e;
    }
  }  
}

Evil Mutating Clients

• But a client could mutate the data structure …

    HashMap h = ...;
    ...
    Iterator i = h.entrySet().iterator();
    System.out.println(i.next());
    System.out.println(i.next());
    h.put("Foo", "Bar"); // hash table resize!
    System.out.println(i.next()); // prints ???
Dealing with Mutable Aggregates

• Two example solutions
  – Defensive copying
  – Timestamping
• Neither solution tracks mutations to container elts

Defensive (Proactive) Copying

• Solution 1: Iterator copies data structure

```java
class QueueIterator implements Iterator<Element> {
    Entry rest;

    QueueIterator(Queue q) {
        // store(deep?)copy q.theQueue in rest
    }
}
```

• Pro: Works even if queue is mutated
• Con: Expensive to construct iterator
Solution 2: Track Mutations

```java
class Queue<Element> {
    int modCount = 0;
    void enqueue(Element e) { ... modCount++; }
    Element dequeue() { ... modCount++; }
    ...
}
```

- Pro: Iteration construction cheap
- Con: Doesn’t allow any mutation
What if Mutation is Allowed?

- Allowed mutation must be part of iterator spec
  
  ```java
  public void remove()
  throws IllegalStateException;
  ```

- Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to next.

- The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

Iterators

- Key ideas
  - Separate aggregate structure from traversal protocol
  - Allows additional kinds of traversals
    - E.g., smallest to largest, largest to smallest, unordered
  - Can support multiple simultaneous traversals
    - Though many Java Collections do not provide this

- Structure
  - Iterator interface defines traversal protocol
  - Concrete Iterator implementations for each aggregate
    - And for each traversal strategy
  - Aggregate instances create Iterator object instances
Design Patterns

• Iterators are an example of a design pattern:
  – Design pattern = problem + solution in context
  – Iterators: solution for providing generic traversals

• Design patterns capture software architectures and designs
  – Not direct 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
Object Modeling Technique (OMT)

• Used to describe patterns in GO4 book
• Graphical representation of OO relationships
  – **Class diagrams** show the static relationship between classes
  – **Object diagrams** represent the state of a program as a series of related objects
  – **Interaction diagrams** illustrate execution of the program as an interaction among related objects

Classes

<table>
<thead>
<tr>
<th>ClassName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation1()</td>
</tr>
<tr>
<td>Type Operation2()</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

| instanceVariable1 |
| Type instanceVariable2 |
| ... |
Object instantiation

Subclassing and Abstract Classes

Note the italics
Pseudo-code and Containment

Window
Area() Q

Rectangle
Area() Q
width
height

return rectangle->Area()

return width * height

Object diagrams

aDrawing
shape[0]
shape[1]

aLineShape

aCircleShape
Components of a Pattern

- **Name(s)**
- **Problem**
  - Context
  - Real-world example
- **Solution**
  - Design/structure
  - Implementation
- **Consequences**
- **Variations, known uses**

Iterator Pattern, Again

- **Name**: Iterator (*aka* Cursor)
- **Problem**:  
  - How to process the elements of an aggregate in an implementation-independent manner?
- **Solution**:  
  - Define an Iterator interface
    - `next()`, `hasNext()`, etc.
  - Aggregate returns an instance of an implementation of Iterator interface to control the iteration
**Iterator Pattern**

- **Consequences:**
  - Support different and simultaneous traversals
    - Multiple implementations of Iterator interface
    - One traversal per Iterator instance
  - Requires coherent policy on aggregate updates
    - Invalidate Iterator by throwing an exception, or
    - Iterator only considers elements present at the time of its creation

- **Variations:**
  - Internal vs. external iteration
    - Java Iterator is external

**Internal Iterators**

```java
public interface InternalIterator<Element> {
    void iterate(Processor<Element> p);
}
public interface Processor<Element> {
    public void process(Element e);
}
```

- The internal iterator applies the processor instance to each element of the aggregate
  - Thus, entire traversal happens “at once”
  - Less control for client, but easy to use
Design Patterns: Goals

• To support **reuse** of successful designs

• To facilitate **software evolution**
  – Add new features easily, without breaking existing ones

• In short, we want to **design for change**

Underlying Principles

• Recurring theme: Reduce implementation dependencies between elements of a software system

• Principles:
  – **Encapsulate what varies**
  – Program to an interface, not an implementation
  – Favor composition over inheritance
    • Use delegation
Encapsulate What Varies

- Some parts of your system are fixed
- Some parts will vary
- Encapsulate the parts that vary in separate classes
- Later you can change these classes without affecting other parts of the system

Program to Interface, Not Implementation

- Read “Interface” as Supertype
- Rely on abstract classes and interfaces to hide differences between subclasses from clients
  - Interface defines an object’s use (protocol)
  - Class defines a particular implementation

- Example: **Iterator** interface, compared to its implementation for a **LinkedList**
Rationale

• Decouples clients from the implementations of the applications they use.
• When clients use an interface, they remain unaware of the specific object types being used.
• Therefore: clients are less dependent on an implementation, so it can be easily changed later.

Favor Composition over Class Inheritance

• White box reuse: Inheritance
  – AClass
    amethod
  →
  BClass
    amethod
• Black box reuse: Composition
  – AClass
    amethod
    aVar
  →
  BClass
    amethod
Rationale

- White-box reuse results in implementation dependencies on the parent class
  - Reusing a subclass may require rewriting the parent
  - But inheritance easy to specify
- Black-box reuse often preferred
  - Eliminates implementation dependencies, hides information, object relationships non-static for better run-time flexibility
  - But adds run-time overhead (additional instance allocation, communication by dynamic dispatch)

Delegation

- Forward messages (delegate) to different instances at run-time; a form of composition
  - May pass invoking object’s this pointer to simulate inheritance

```
Window
area()

rectangle

Rectangle
area()
width
height

Return width * height

return rectangle.area()
```
Rationale

• Object relationships dynamic
  – Composes or re-composes behavior at run-time
• But:
  – Sometimes code harder to read and understand
  – Efficiency (because of black-box reuse)

Design Patterns Taxonomy

• Creational patterns
  – Concern the process of object creation
• Structural patterns
  – Deal with the composition of classes or objects
• Behavioral patterns
  – Characterize the ways in which classes or objects interact and distribute responsibility
Catalogue of Patterns: Creation Patterns

- **Singleton**
  - Ensure a class only has one instance, and provide a global point of access to it.

- **Typesafe Enum**
  - Generalizes Singleton: ensures a class has a fixed number of unique instances.

- **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.

- **Proxy**
  - Provide a surrogate or placeholder for another object to control access to it.

- **Decorator**
  - Attach additional responsibilities to an object dynamically.
Behavioral Patterns

• Template
  – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses

• State
  – Allow an object to alter its behavior when its internal state changes. The object will appear to change its class

• Observer
  – Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Singleton Objects

• Problem:
  – Some classes have conceptually one instance
    • Many printers, but only one print spooler
    • One file system
    • One window manager
  – Creating many objects that represent the same conceptual instance adds complexity and overhead

• Solution: only create one object and reuse it
  – Encapsulate the code that manages the reuse
The Singleton Solution

- Class is responsible for tracking its sole instance
  - Make constructor private
  - Provide static method/field to allow access to the only instance of the class
- Benefit:
  - Reuse implies better performance
  - Class encapsulates code to ensure reuse of the object; no need to burden client

Singleton pattern
Implementing the Singleton method

• In Java, just define a final static field

```java
public class Singleton {
    private Singleton() {...}

    final private static Singleton instance = new Singleton();

    public static Singleton getInstance()
    {
        return instance;
    }
}
```

• Java semantics guarantee object is created immediately before first use

Marshalling

• Marshalling is the process of transforming internal data into a form that can be
  – Written to disk
  – Sent over the network
  – Etc.

• Unmarshalling is the inverse process
Java provides support for marshalling objects

- Classes implement the `Serializable` interface
- The JVM implements standard marshalling and unmarshalling automatically
  - E.g., enables you to create persistent objects, stored on disk
  - This can be useful for building a light-weight database
  - Also useful for distributed object systems

Often, generic implementation works fine
- But let’s consider singletons...

What happens when we unmarshall a singleton?

```
// Singleton.instance // Singleton.instance
```

Problem: JVM doesn’t know about singletons
- It will create two instances of `Singleton.instance`!
- Oops!
Marhsalling and Singletons (cont’d)

- **Solution**: Implement
  - `Object readResolve() throws ObjectStreamException;`
    - This method will be called after standard unmarshilling
    - Returned result is substituted for standard unmarshalled result
- **E.g., add to Singleton class the following method**
  - `Object readResolve() { return Singleton.instance; }`
- **Notes**: Serialization is wacky!
  - For example, objects can only be nested 1001 deep????

Generalizing Singleton: Typesafe Enum

- **Problem**:
  - Need a number of unique objects, not just one
  - Basically want a C-style enumerated type, but safe
- **Solution**:
  - Generalize the Singleton Pattern to keep track of multiple, unique objects (rather than just one)
Typesafe Enum Pattern

```
Enum
  static Enum inst1
  static Enum inst2
  EnumOp ()
  data

Enum
  EnumOp ()
data

Enum
  EnumOp ()
data

Note: constructor is private
```

Typesafe Enum: Example

```
public class Suit {
  private final String name;

  private Suit(String name) { this.name = name; }

  public String toString() { return name; }

  public static final Suit CLUBS    = new Suit("clubs");
  public static final Suit DIAMONDS = new Suit("diamonds");
  public static final Suit HEARTS   = new Suit("hearts");
  public static final Suit SPADES   = new Suit("spades");
}
```
Enumerations in Java 1.5

• New version of Java has type safe enums
  – Built-in: Don’t need to use the design pattern
• public enum Suit {
  CLUBS("clubs"), DIAMONDS("diamonds"), HEARTS("hearts"), SPADES("spades")
}
  – Type checked at compile time
  – Can be used in switch statements
  – Two extra class methods:
    • public static <this enum class>[] values() -- the enumeration elts
    • public static <this enum class> valueOf(String name) -- get an elt

Adapter (aka Wrapper) Pattern

• Problem:
  – You have some code you want to use for a program
  – You can’t incorporate the code directly (e.g., you just have the .class file, say as part of a library)
  – The code does not have the interface you want
    • Different method names
    • More or fewer methods than you need
• To use this code, you must adapt it to your situation
• Here’s what we have:

- Client is already written, and it uses the Target interface
- Adaptee has a method that works, but has the wrong name/interface

• How do we enable the Client to use the Adaptee?

• Solution: adapter class to implement client’s expected interface, forwarding methods
Proxy Pattern Motivation

• Goal:
  – Prevent an object from being accessed directly by its clients

• Solution:
  – Use an additional object, called a proxy
  – Clients access protected object only through proxy
  – Proxy keeps track of status, location, etc. of protected object

Uses of the Proxy Pattern

• *Virtual proxy*: impose a lazy creation semantics, to postpone expensive object creations until strictly necessary.

• *Monitor proxy*: impose security constraints on the original object, say by intercepting some method calls to proxied object.

• *Remote proxy*: hide the fact that an object resides on a remote location.
Proxy vs. Adapter

- Proxies implement *the same* interface as the objects they adapt
  - But may restrict some operations
  - E.g., refuse to perform a sensitive operation

- Adapters implement *a different* interface than the objects they adapt

Example Usage Class Diagram
More OMT Notation

- Arrow ending in filled circle
  - More than one

Decorator Pattern

- Motivation
  - Want to add responsibilities/capabilities to individual objects, not to entire class
  - Inheritance requires a compile-time choice of parent class

- Solution
  - Enclose the component in another object that adds the responsibility/capability
    - The enclosing object is called a decorator.
Example: Java I/O

class LineNumberReader extends BufferedReader {
    private int lineNumber;
    public LineNumberReader(Reader in) { super(in); }
    public int getLineNumber() { return lineNumber; }

    public int read() {
        int c = super.read();
        if (c == '\n') lineNumber++;
        return c;
    }
}

Interaction Diagram
Decorator Pattern: Another Example

More OMT Notation

- Arrow beginning with diamond
  - “Part-of” or aggregation
  - Only accessed by object pointing to it
Decorator Pattern: Features

- Decorator conforms to interface of decorated component
  - Its presence is transparent to the component's clients.
- Decorator forwards requests to encapsulated component
  - May perform additional actions before or after
- Can nest decorators recursively
  - Allows unlimited added responsibilities
- Can add/remove responsibilities dynamically
Decorator Pattern Analysis

- Advantages
  - Fewer classes than with static inheritance
  - Dynamic addition/removal of decorators
  - Keeps root classes simple

- Disadvantages
  - Proliferation of run-time instances
  - Abstract Decorator must provide common interface

- Tradeoffs:
  - Useful when components are lightweight
  - Otherwise use Strategy

Decorator vs. Adapter

- A decorator *adds* to the responsibilities of an object
  - Usually has the same interface plus more features

- An adapter *changes* the interface
  - But usually not the responsibilities
Template Method Pattern

• Problem
  – You’re building a reusable class
  – You have a general approach to solving a problem,
  – But each subclass will do things differently

• Solution
  – Put invariant parts of an algorithm in parent class
  – Encapsulate variant parts in template methods
  – Subclasses override template methods
  – At runtime template method invokes subclass ops

Structure
Example: JUnit 3

- JUnit 3 uses template methods pattern for `run()`

```java
package junit.framework;
public class TestCase {
  ...
  public void run() {
    setUp(); runTest(); tearDown()
  }
}
```

- Users can subclass `TestCase` and override `setUp()`, `runTest()` and `tearDown()`

Observer Pattern

- Problem
  - Dependent must be consistent with master’s state

- Solution structure: Four kinds of objects
  - **Abstract subject (master)**
    - Maintains list of dependents; notifies them when master changes
  - **Abstract observer (dependents)**
    - Defines protocol for updating dependents
  - **Concrete subject**
    - Manages data for dependents; notifies them when master changes
  - **Concrete observers**
    - Gets new subject state upon receiving update message
Observer Pattern

Use of Observer Pattern
Observer Pattern (cont’d)

• Consequences
  – Low coupling between subject and observers
    • Subject unaware of dependents
  – Support for broadcasting
    • Dynamic addition and removal of observers
  – Unexpected updates
    • No control by the subject on computations by observers

Observer Pattern (cont’d)

• Implementation issues
  – Storing list of observers
    • Typically in subject
  – Observing multiple subjects
    • Typically add parameters to update()
  – Who triggers update?
    • State-setting operations of subject
      – Possibly too many updates
    • Client
      – Error-prone if an observer forgets to send notification message
Observer Pattern (cont’d)

• Implementation issues (cont’d)
  – Possibility of dangling references when subject is deleted
    • Easier in garbage-collected languages
    • Subject notifies observers before dying
  – Possibility of premature notifications
    • Typically, method in Subject subclass calls inherited method which does notification
    • Solve by using Template method pattern
      – Method in abstract class calls deferred methods, which is defined by concrete subclasses

Observer Pattern (cont’d)

• Implementation issues (cont’d)
  – How much information should subject send with update() messages?
    • Push model: Subject sends all information that observers may require
      – May couple subject with observers (by forcing a given observer interface)
    • Pull model: Subject sends no information
      – Can be inefficient
  – Registering observers for certain events only
    • Use notion of an aspect in subject
    • Observer registers for one or more aspects
Observer Pattern (cont’d)

• Implementation issues (cont’d)
  – Complex updates
    • Use change managers
    • Change manager keeps track of complex relations among (possibly) many subjects and their observers and encapsulates complex updates to observers

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.

• Make 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.

State Pattern

• Problem
  – An object is always in one of several known states
  – The state an object is in determines the behavior of several methods

• Solution
  – Could use if/case statements in each method
  – Better solution: use dynamic dispatch
State Pattern Approach

• Encode different states as objects with same superclass

• To change state, change the state object

• Methods delegate to state object

Example – Finite State Machine

```java
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();
}
```
FSM Example – cont.

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;}
}

Structure of State Pattern
State Pattern Notes

• Can use singletons for instances of each state class
  – State objects don’t encapsulate (mutable) state, so can be shared

• Easy to add new states
  – New states can extend the base class, or
  – New states can extend other states
    • Override only selected functions
Lexi: Simple GUI-Based Editor

- Lexi is a WYSIWYG editor
  - Supports documents with textual and graphical objects
  - Scroll bars to select portions of the document
  - Be easy to port to another platform
  - Support multiple look-and-feel interfaces
- Highlights several OO design issues
- Case study of design patterns in the design of Lexi

Lexi User Interface
Design Issues

- Representation and manipulation of document
- Formatting a document
- Adding scroll bars and borders to Lexi windows
- Support multiple look-and-feel standards
  - Motif and Presentation Manager (!)
- Handle multiple windowing systems
- Support user operations
- Advanced features
  - spell-checking and hyphenation

Structure of a Lexi Document

- Goals:
  - Store text and graphics in document
  - Generate visual display
  - Maintain info about location of display elements
- Caveats:
  - Treat different objects uniformly
    - E.g., text, pictures, graphics
  - Treat individual objects and groups of objects uniformly
    - E.g., characters and lines of text
Structure of a Lexi Document

- Use recursive composition for defining and handling complex objects
  - Abstract class Glyph for all displayed objects
  - Glyph responsibilities:
    - Know how to draw itself
    - Knows what space it occupies
    - Knows its children and parent
  - Glyph instances can recursively compose other Glyph instances
The Composite Pattern

• Motivation:
  – Support recursive composition in such a way that a client need not know the difference between a single and a composite object (as with Glyphs)

• Applicability:
  – When dealing with hierarchically-organized objects (e.g., columns containing rows containing words …)
Composite Pattern Structure

Composite Pattern Consequences

- Class hierarchy has both **simple** and **composite** objects
- Simplifies clients
- Aids extensibility
  - Clients do not have to be modified
- Too general a pattern?
  - Difficult to to restrict functionality of concrete leaf subclasses
• We know that documents are represented as Glyphs, but not how documents are constructed.

• Formatting:
  – Document structure will be determined based on rules for justification, margins, line breaking, etc.
  – Many good algorithms exist;
    • different tradeoffs between quality and speed

• Design decision: implement different algorithms, decide at run-time which algorithm to use
  – define root class that supports many algorithms
  – each algorithm implemented in a subclass

---

Strategy Pattern

• Name
  – Strategy (aka Policy)

• Applicability
  – Many related classes differ only in their behavior
  – Many different variants of an algorithm
  – Need to encapsulate algorithmic information
Strategy Pattern: Structure

- **Context**
  - ContextInterface()

- **Strategy**
  - AlgorithmInterface()

- **ConcreteStrategy**
  - ConcreteStrategyA: AlgorithmInterface()
  - ConcreteStrategyB: AlgorithmInterface()
  - ConcreteStrategyC: AlgorithmInterface()

Strategy Pattern: Consequences

- Clear separation of algorithm definition and use
  - Glyphs and formatting algorithms are independent
  - Alternative (many subclasses) is unappealing
    - Proliferation of classes
    - Algorithms cannot be changed dynamically

- Elimination of conditional statements
  - Like State, Template, ...
  - Typical in OO programming
Strategy Pattern Consequences (cont’d)

- Clients must be aware of different strategies
  - When initializing objects
- Proliferation of instances at run-time
  - Each Glyph has a strategy object with formatting information
  - If strategy is stateless, share strategy objects

Lexi: Using Strategy

- Compositor and Composition classes
  - **Compositor**: class encapsulating formatting algorithm
    - Pass Composition objects to be formatted as parameters to Compositor methods
  - **Composition**: things being formatted
    - Glyph subclass
    - Each Composition object refers to its Compositor object
    - When a Composition needs to format itself, it sends a message to its Compositor instance
Spell-Checking and Hyphenation

- Must do textual analysis
  - Multiple operations and implementations
- Must add new functions and operations easily
- Must efficiently handle scattered information and varied implementations
  - Different traversal strategies for stored information
- Should separate actions from traversal

Structure of Iterator Pattern
Visitor: Implementing Analyses

- Often want to implement multiple analyses on the same kind of object data
  - Spellchecking and Hyphenating Glyphs
  - Generating code for and analyzing an Abstract Syntax Tree (AST) in a compiler
- One solution: implement each analysis as a method in each object
  - Follows idea “objects are responsible for themselves”
  - But many analyses will occlude the object’s main code
  - Result is classes hard to maintain

Abstract Syntax Trees

```java
public interface Node { }

public class Number extends Node {
    public int n;
}

public class Plus extends Node{
    public Node left;
    public Node right;
}
```
Traversing Abstract Syntax Trees

```java
public interface Node {
    public int sum();
}

public class Number extends Node {
    public int n;
    public int sum() { return n; }
}

public class Plus {
    public Node left;
    public Node right;
    public int sum() { return left.sum() + right.sum(); }
}
```

Naïve approach (not a visitor)

One method for each analysis

```
Node
  TypeCheck()
  GenerateCode()
  PrettyPrint()

VariableRefNode
  TypeCheck()
  GenerateCode()
  PrettyPrint()

AssignmentNode
  TypeCheck()
  GenerateCode()
  PrettyPrint()
```
Use a Visitor

• Alternatively, can define a separate **visitor** class
  – A visitor encapsulates the operations to be performed on an entire structure, e.g., all elements of a parse tree

• Allows operations to be separate from structure
  – But doesn’t necessarily require putting all of the structure traversal code into each visitor/operation

Sample Visitor class

```
NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)

TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)

CodeGenGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
```
How to perform traversal?

- Now that we have a visitor class, how do we apply its analysis to the objects of interest?
  - Add `accept(visitor)` method to each structure class, that will invoke the given visitor on `this`.
  - Builds on Java’s dynamic dispatch.
  - Use an iteration algorithm (like an Iterator) to call `accept()` on each relevant object.

Sample visited objects
Visitor Interaction

Visitor pattern

• Name
  – Visitor or double dispatching

• Applicability
  – Related objects must support different operations and actual op depends on both the class and the op type
  – Distinct and unrelated operations pollute classdefs
  – **Key**: object structure rarely changes, but ops changed often
Visitor Pattern Structure

- Define two class hierarchies
  - One for object structure
    - AST in compiler, Glyphs in Lexi
  - One for each operation family, called visitors
    - One for typechecking, code generation, pretty printing in compiler
    - One for spellchecking or hyphenation in Lexi
Use of Visitor Pattern in Lexi

Visitor Pattern Consequences

- Adding new operations is easy
  - Add new op subclass with method for each concrete elt class
  - Easier than modifying every element class
- Gathers related operations and separates unrelated ones
- Adding new concrete elements is difficult
  - Must add a new method to each concrete Visitor subclass
- Allows visiting across class hierarchies
  - Iterator needs a common superclass (i.e., composite pattern)
- Visitor can accumulate state rather than pass it as parameters
Implementing Traversal

• Who is responsible for traversing object structure?
• Plausible answers:
  – Visitor
    • But, must replicate traversal code in each concrete visitor
  – Object structure
    • Define operation that performs traversal while applying visitor object to each component
  – Iterator
    • Iterator sends message to visitor with current element as arg

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
  – Traversal routine takes two arguments, the visitor and the object to traverse
    • o.accept(aVisitor) will dispatch on the actual identity of o (the object being considered)
    • ...and accept will internally dispatch on the identity of aVisitor (the object visiting it).
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

Visitors Can Forward Common Behavior

• Useful for composites
  – If subclasses of a particular object all treated the same
  – Can have visit(SubClass) call visit(SuperClass)

• For example
  – visit(BinaryPlusOperatorNode) can just forward call to superclass visit(BinaryOperatorNode)
State in a Visitor Pattern

• A visitor can contain state
  – E.g., the results of typechecking the program so far

```java
class TypeCheckingVisitor extends Visitor {
    private TypeMap map;
    void visit(VariableRefNode n) { …
        map.add(n,t)
    … }
}
```

• Or visitors pass around a separate state object
  – Impacts the type of the Visitor superclass

Traversals

• It’s sometimes preferable to try to keep traversal separate from the Visitor
  – E.g., use an Iterator
  – Thus traversal and analysis can evolve independently

• But can also do it within node or visitor class. Several solutions here:
  – `acceptAndTraverse` methods
    • `traverse from within accept()`
  – Separating processing from traversal
    • `Visit/process methods`
  – Traversal visitors applying an operational visitor
acceptAndTraverse Methods

- 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 previsit 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

Accept and Traverse

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

- Can have two parallel sets of methods in visitors
  - Visit() methods
  - Process() methods
- Allows finer-grained subtyping of Visitor classes that include traversal
  - Subclass a visitor, and just change the process method
- How it works: the 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)

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);
  }
  ...
}
Adding Scroll Bars and Borders: Decorator

- How to define classes for scrollbars and borders?
- Define as subclasses of Glyph
  - Scrollbars and borders are displayable objects
  - Will use notion of **transparent enclosure**
    - Clients don’t need to know whether they are dealing with a component or with an enclosure
- Inheritance increases number of classes
  - Use composition instead ("has a")

Transparent Enclosure

- Two features:
  - Single-child composition
    - Calls its child, then adds its own behavior
  - Compatible interfaces
    - Can use the enclosing object in place of the one it encloses
- Implemented by the Decorator pattern
  - Saw this earlier
Monoglyph class: a Decorator

Class Monoglyph {
    void Draw (Window w) {
        component.Draw(w);
    } ...
}

Class Border extends Monoglyph {
    void Draw (Window ){
        super.Draw(w);
        DrawBorder(w);
    } ...
}

Changing Look-and-Feel: Abstract Factory

- Goal: easily change Lexi’s look-and-feel
  - When new libraries are available (future variability)
  - At run-time by switching between them (present variability)
- Thoughtless implementation technique:
  - Use distinct class for each widget and standard
  - Let clients handle different instances for each standard
    - Button pb = new MotifButton(); // bad
Abstracting Creation

• Concrete Creation problems:
  – Class of object is fixed at compile-time
    • Can’t change standard at run-time
  – Changing the class means making changes all over the code
• Instead:
  – Use a class to create abstract classes:
    • Button pb = guiFactory.createButton(); // better

Solution: Use Abstract Factory

• Define abstract class GUIFactory with creation methods for widgets
  – Concrete subclasses of GUIFactory actually define creation methods for each look-and-feel standard
    • MotifFactory, MacFactory, etc.
  – Specialize each widget into subclasses for each look-and-feel standard
• Thus, can easily change the kind of factory without changes all over the place
Class Diagram for GUIFactory

Diagram for Product Classes
Abstract Factory Pattern

- **Name**
  - Abstract Factory or Kit

- **Applicability**
  - Different families of components (products)
  - Must be used in mutually exclusive and consistent way
  - Hide existence of multiple families from clients

Structure of Abstract Factory
Abstract Factory: Consequences

- Isolate instance creation and handling from clients
- Can easily change look-and-feel standard
  - Reassign a global variable;
  - Recompute and redisplay the interface
- Enforce consistency among products in each family
- Adding to family of products is difficult
  - Have to update factory abstract class and all concrete classes

Multiple Window Systems

- Want portability to different window systems
  - Similar to multiple look-and-feel problem, but different vendors will build widgets differently
- Solution:
  - Define abstract class Window, with basic window functionality (e.g., draw, iconify, move, resize, etc.)
  - Define concrete subclasses for specific types of windows (e.g., dialog, application, icon, etc.)
  - Define WindowImp hierarchy to handle window implementation by a vendor
Bridge Pattern

- **Name**
  - Bridge or Handle or Body

- **Applicability**
  - Handles abstract concept with different implementations
  - Implementation may be switched at run-time
  - Implementation changes should not affect clients
  - Hide a class’s interface from clients

- **Structure**: use two hierarchies
  - Logical one for clients,
  - Physical one for different implementations
Bridge Pattern

- Consequences:
  - Decouple interface from implementation and representation
  - Change implementation at run-time
  - Improve extensibility
    - Logical classes and physical classes change independently
    - Hides implementation details from clients
      - Sharing implementation objects and associated reference counts
Supporting User Commands

- Support execution of Lexi commands
  - GUI doesn’t know
    - Who command is sent to
    - Command interface
- Complications
  - Different commands have different interfaces
  - Same command can be invoked in different ways
  - Undo and Redo for some, but not all, commands (print)

An improved solution
- Create abstract “command” class
- Create action-performing glyph subclass
- Delegate action to command

Key ideas
- Pass an object, not a function
- Pass context to the command function
- Store command history
Command Objects

Command Pattern

- **Name**
  - Command or Action or Transaction

- **Applicability**
  - Parameterize objects by actions they perform
  - Specify, queue, and execute requests at different times
  - Support undo by storing context information
  - Support change log for recovery purposes
  - Support high-level operations
    - Macros
Structure of Command Pattern

Command Pattern

- Consequences:
  - Decouple receiver and executor of requests
    - Lexi example: Different icons can be associated with the same command
  - Commands are first class objects
  - Easy to support undo and redo
    - Command must have method to check whether it’s reversible
    - Must add state information
  - Can create composite commands
    - Editor macros
  - Can extend commands more easily
Command Pattern

• Implementation notes
  – How much should command do itself?
  – Support undo and redo functionality
    • Operations must be reversible
    • May need to copy command objects
    • Don’t record commands that don’t change state
  – Avoid error accumulation in undo process

Comparing Objects

• Java has two designs for objects that can be (totally) ordered
  – These are things for which sorting makes sense
  – E.g., strings, integers, etc.
Comparable

```java
public interface Comparable {
    // Returns negative integer, zero, or a positive integer if this
    // object is less than, equal to, or greater than o.
    public int compareTo(Object o);
}
```

- Advantages and disadvantages?
  - Can only implement one `compareTo` operation
  - No extra levels of indirection; objects know how to compare themselves

Comparator

```java
public interface Comparator {
    int compare(Object o1, Object o2);
}
```

- Advantages and disadvantages?
  - Can have multiple comparison operations
  - An example of delegation
  - Comparator needs to know innards of your objects
    - Can make the Comparator implementer an inner class
  - Extra indirection; more objects floating around
Pattern Hype

• Patterns get a lot of hype and fanatical believers
  – Your mileage may vary (YMMV)
• Patterns are sometimes wrong or inappropriate for a particular language or environment
  – Patterns developed for C++ can have very different solutions in Smalltalk or Java