Design Patterns

Spring 2021

(Inspiration from Ben Liblit and Mike Ernst)
Software Design Patterns

- Reusable solutions to common programming problems
  - Gamma, Helm, Johnson, Vlissides (“Gang of Four”, “GoF”), *Design Patterns*, 1995

- Patterns provide
  - Vocabulary for common programming problems
  - Good design ideas for solving those problems
  - Tradeoffs between different design choices

- Patterns are not
  - Classes, libraries, or frameworks
  - Fully fleshed out designs
  - Very well defined (what is and what is not a pattern?)
Iteration

- Problem: Loop through all objects in a collection

```java
public class LinkedList { // from last lecture
    public int size() {
        int i = 0; Cell c = head;
        while (c != null) { i++; c = c.next; }
        return i;
    }

    public int get(int pos) {
        Cell c = head;
        for (int i = 0; i < pos; i++) {
            if (c == null) {
                throw new IndexOutOfBoundsException();
            }
            c = c.next;
        }
        return c.elt;
    }
}
```
Iteration: Commonalities

• Problem: Loop through all objects in a collection

```java
public class LinkedList { // from last lecture
    public int size() {
        int i = 0; Cell c = head;
        while (c != null) { i++; c = c.next; }
        return i;
    }

    public int get(int pos) {
        Cell c = head;
        for (int i = 0; i < pos; i++) {
            if (c == null) {
                throw new IndexOutOfBoundsException();
            }
            c = c.next;
        }
        return c.elt;
    }
}
```
Iteration as Design Pattern?

• Examples are similar but not exactly the same
  ■ Seems fine for instance methods
  ■ Sensible to optimize those for implementation details

• But what if a client wants to iterate through a list?
  ■ Probably shouldn’t expose Cell to them
  ■ Probably shouldn’t expose other implementation details
  ■ Need to abstract the concept of iteration

• Tradeoffs of abstraction
  ■ Pros: ease-of-use, strong separation between client/library
  ■ Cons: increased overhead, limited iteration strategies
Iteration in Java.Util

• Create an object to maintain iteration state

```java
public interface Iterator<E> {
    boolean hasNext();
    E next();
    // also, forEachRemaining and remove
}
```

• Example client usage

```java
LinkedList l = …;
Iterator i = l.iterator();
while (i.hasNext()) {
    Integer x = i.next();
    // do something with x
}
```
public class LinkedList {
    public class LinkedListIterator implements Iterator<Integer> {
        Cell cur;
        LinkedListIterator(Cell head) { cur = head; }
        public boolean hasNext() { return cur != null; }
        public Integer next() {
            Integer temp = cur.elt;
            cut = cur.next;
        }
    }
    public LinkedListIterator iterator() {
        return new LinkedListIterator(head);
    }
}
Cool Java Syntactic Sugar

- If we add the following:
  
  ```java
  public class LinkedList implements Iterable<Integer>
  
  - Iterable interface just means we have an iterator method
  
  - (The Iterable interface also includes a couple of default methods, which mean the interface provides code for them)
  
  - Then the following code is the same!
  ```

```java
LinkedList l = ...;
Iterator i = l.iterator();
while (i.hasNext()) {
    Integer x = i.next();
    // do something with x
}
```

```java
LinkedList l = ...;
for (Integer x : l) {
    // do something with x
}
```
## Some Tradeoffs

// suppose code in an instance method
LinkedList l = ...;
Cell c = l.head;
while (c != null) {
    Integer x = c.elt;
    c = c.next;
    // do something with x
}

LinkedList l = ...;
Iterator i = l.iterator();
while (i.hasNext()) {
    Integer x = i.next();
    // do something with x
}

<table>
<thead>
<tr>
<th>Direct code</th>
<th>Iterator code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Longer</td>
<td>Shorter</td>
</tr>
<tr>
<td>Stores iterator state on stack</td>
<td>(Heap) object for iterator state</td>
</tr>
<tr>
<td>Iteration code mixed in</td>
<td>Iteration code separate</td>
</tr>
</tbody>
</table>
Other Iteration Concerns

- Iterator should not modify collection
  - That’s why `LinkedListIterator` are separate objects
  - Design goal: allow multiple iterators at once

- Client should not modify list during iteration!
  - If client adds an element, should element be seen by iterator or not?
    - Might depend on implementation details
  - `java.util` classes will throw a `ConcurrentModificationException` if client tries this

- **Iterables** can choose whether to support removal during iteration
  - See optional `remove` method in `Iterator` interface
  - Discussion: Is supporting `remove` a good idea?
Iterators are a Design Pattern

• Problem: Each data structure needs its own traversal code
  ▪ Putting that code in the client introduces coupling between client and data structure

• Solution: Data structure provides a traversal method

• Consequences:
  ▪ Library does not need to expose internal representation
  ▪ Iteration order fixed by library
  ▪ Possible performance overhead
Boxes and Arrows

- Imagine you were at the whiteboard, trying to explain iterators to another student
  - What would you draw?
  - Answer always seems to be: Boxes and arrows

- GoF book proposes *object modeling technique*
  - Class diagrams: static relationship between classes
  - Object diagrams: state of a program’s objects
  - Interaction diagram: sequencing of method calls

- Became *Unified Modeling Language (UML)*
  - Standardized in 1997
  - Many people take UML very seriously
    - Please don’t do so; UML is a means, not an end
    - And it’s never sufficient in practice
UML Class Diagram for Iterators

- Most boxes indicate classes or interfaces
  - Arrows to indicate inheritance/implements (open triangles) or pointer relationship (closed triangles)
- For this course, don’t worry about different arrows etc.

```
Iterator<E>

hasNext()
next()

LinkedListIterator

Iterable<E>

iterator()

LinkedList

Cell head
iterator()

return new LinkedListIterator(head);
```
Internal Iterators

• Alternative design: Client passes in *callback* to iterator method; library calls client once per element

```java
interface Processor {
    void process(Integer x);
}
class LinkedList {
    void iterate(Processor p) {
        Cell cur = head;
        while (cur != null) {
            p.process(cur.elt);
            cur = cur.next;
        }
    }
}
class LengthProcessor {
    int size = 0;
    void process(Integer x) {
        size++;
    }
}
LinkedList l = ...;
LengthProcessor p = new LengthProcessor();
l.iterate(p);
// p.size is list len
```
Anonymous Inner Classes

• Could also use an *anonymous inner class*
  
  - `new C() { fields and methods }` creates a subclass of C with the given fields and methods, and creates one instance of it
  
  - Works with class or interface

```java
LinkedList l = ...;
LengthProcessor p = new Processor() {
    int size = 0;
    void process(Integer x) {
        size++;
    }
} 

l.iterate(p);
// p.size is list len
```
UML Sequence Chart Example

- Shows pattern of method calls across objects

```plaintext
Client

new LengthProcessor

iterate

LinkedList

process

process

...
Exercise

• In a small group, implement internal and/or external iterators for ArrayList
Coupling

• Design patterns often reduce coupling
  ▪ Coupling is the amount of interdependence among modules
  ▪ Low coupling helps make software easier to understand and change

• Iterator pattern reduces coupling
  ▪ Hides implementation details from client
  ▪ Helps separate iteration code from other concerns

• But it’s not perfect
  ▪ Performance details are not hidden
  ▪ Whether elts can be removed during iteration not hidden

• ADTs also reduce coupling!
Cohesion

- **Cohesion** is the degree to which a module’s internal elements are related
  - `LinkedList`, `ArrayList` have high cohesion because all the methods are concerned with the data structures
  - But, `java.lang.Math` has only moderate cohesion, because the methods are not that related
    - E.g., `sin` and `cos` (sine and cosine) should be in same class, but does `sqrt` need to be in the same class?

- High cohesion is good because
  - Code that may need to be modified together is grouped together
  - Code that has dependencies on each other is grouped inside a module

- Design patterns say little to nothing about cohesion!
When Not to Use Design Patterns

• Key rule: Avoid premature complication!
  ▪ Don’t add a design pattern because it’s cool
  ▪ First get something working, then generalize it

• Design patterns can cause bloat
  ▪ Adds indirection, increases code size, adds complexity
  ▪ Could wind up making code *harder* to understand!

• Important: Design patterns are not fixed and rigid
  ▪ They *must* be modified to suit the circumstances
  ▪ Focus on solving your problem well, not on using a particular pattern
Design Patterns Across Languages

- Most design patterns don’t generalize that well across different programming paradigms
  - And most design patterns are for OO languages
  - Functional programming has design pattern-like stuff, but it’s not usually called design patterns

- Design patterns often compensate for language weaknesses
  - E.g., internal iterators are really common in functional programming, like `map` and `fold` (see COMP 105)
Creational Patterns
Singleton Objects

• Some classes should have only one instance
  ▪ Logger, DB, ThreadPool, Config, …

• Problem: No way to intercept `new`
  ▪ Each call to `new` allocates a fresh object and initializes it
  ▪ But we want to somehow return the same object

• Solution: don’t expose `new`
  ▪ Make constructor `private`
  ▪ Create a single instance and manage it through a method

• Benefits
  ▪ Can create instance lazily without client worrying about it

• But mostly, starting place for other creational patterns
  ▪ Singletons often become non-singleton as software evolves…
Singleton Example

- `theLogger` only created once
  - Notice: we can guarantee that without looking at other code!
  - Lazy allocation, on first use

```java
class Logger {
    private static theLogger;

    private Logger() {
        ...}

    public static getLogger() {
        if (theLogger == null) {
            theLogger = new Logger();
        }
        return theLogger;
    }
}
```
Singleton Example (Alternative)

```java
class Logger {
    private Logger() { … }

    final private static theLogger = new Logger();

    public static getLogger() {
        return theLogger;
    }
}
```

- A **final** field cannot be overwritten
- **theLogger** guaranteed created before use
  - Eager allocation, when Logger class loaded
Generalizing Singletons: Enums

• What if we need several, related unique objects rather than one?
  ▪ Common scenario: an enumeration, i.e., a finite set of objects representing a finite set of abstract things
  ▪ E.g., days of week: MONDAY, TUESDAY, WEDNESDAY, …
  ▪ E.g., card suits: CLUBS, DIAMONDS, HEARTS, SPADES

• C solution: enumeration
  ▪ enum suit { clubs, diamonds, hearts, spades }
  ▪ Problem: not type safe!
  ▪ Freely interchangeable with ints

• Java solution: multiple instances of a class
Enum Example

private 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");

• Enum members cannot be mixed with ints
• Set of members is immutable
• Members can be compared using physical equality
• Enum members can carry useful methods!
Java Enumerations

• This design pattern is actually built in to Java!

```java
public enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
```

- Exercise: Use `javap -c` to figure out implementation!

• Enums have some other useful methods
  - `values()` — enumerator elements
  - `valueOf(String name)` — get corresponding element
Why is the constructor public?!

- “The Boolean type should not have had public constructors…I've seen programs that produce millions of trues and millions of falses, creating needless work for the garbage collector.” —Josh Bloch, JavaWorld, Jan 4, 2004
Factories

• Making constructor `private` is generally useful
  - Gives us a “hook” so classes can control object creation

• Two additional design patterns that use this idea
  - *Factory methods* — A method called to create objects
    - Key: Might not return a fresh object each time
  - *Factory object* — An object with a creator method
    - The object can be passed around, i.e., object creation becomes “higher order”
Integer.valueOf Factory Method

- Complex logic to reduce number of allocations
  - “Small” integers are preallocated in cache and reused
  - Other integers are allocated on the fly and *not* reused
- Client can call it without worry about details

```java
public class Integer {
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }
}
```

https://hg.openjdk.java.net/jdk/jdk11/file/1ddf9a99e4ad/src/java.base/share/classes/java/lang/Integer.java
Calendar.getInstance Factory Meth

public static Calendar getInstance() {
    Locale aLocale = Locale.getDefault(Locale.Category.FORMAT);
    return createCalendar(defaultTimeZone(aLocale), aLocale);
}

• Uses default time zone and locale to create and return an appropriate Calendar object

https://hg.openjdk.java.net/jdk/jdk11/file/1ddf9a99e4ad/src/java.base/share/classes/java/util/Calendar.java
**Factory Objects for Themes**

interface Theme {
    Button newButton(String text);
    DatePicker newDatePicker();
}

class LightFactory implements Theme {
    Button newButton(String text) { return new LightButton(text); }
    DatePicker newDatePicker() { return new LightDatePicker(); }
}

class DarkFactory implements Theme {
    Button newButton(String text) { return new DarkButton(text); }
    DatePicker newDatePicker() { return new DarkDatePicker(); }
}

void drawWindow(Theme t) {
    ... t.newButton("Open") ... t.newDatePicker() ...
Factory Object Discussion

• Two parallel collections of objects, each of which have the same interface
  - \{Light,Dark\}Button, \{Light,Dark\}DatePicker, ...

• Factory object has methods for creating the objects in the collection
  - One factory for each collection of objects
  - Provides the capability to create objects
  - Can pass an instance of the factory around the program

• At use (e.g., \texttt{drawWindow}), dynamic dispatch decides which objects are created
  - Can be flexibly expanded with many more themes \textit{without changing} \texttt{drawWindow}!
Structural Patterns
Wrappers

- Wrappers are a thin layer around an existing class
  - Adapter — same functionality, different interface
  - Proxy — same interface, additional logic
    - Usually, access control or condition checking
  - Decorator — same interface, change functionality
Adapter Pattern

- Problem: Client needs functionality of another class (adaptée) but is written to a different interface
- Solution: Introduce an adapter

```
Client <- DesiredInterface
    someMeth()
    otherMeth()

Adapter
    private adaptee someMeth()
    otherMeth()

    ...adaptee.meth1()...adaptee.meth2()...

Adaptee
    meth1()
    meth2()
```
An Example You’ve Seen?

```java
interface Graph {
    boolean addNode(String n);
    boolean addEdge(String n1, String n2);
}

public interface EdgeGraph {
    boolean addEdge(Edge e);
}

public class EdgeGraphAdapter implements EdgeGraph {
    private Graph g;
    EdgeGraphAdapter(Graph g) { this.g = g; }
    // methods of EdgeGraph call methods of g
}
```
interface XYVec {
    double getX();
    double getY();
}

interface PolarVec {
    double getAngle();
    double getLen();
}

class XYAdapter implements XYVec {
    private PolarVec v;
    XYVec(double x, double y) { ... }
    double getX() {
        return v.getLen()*Math.cos(v.getAngle());
    }
    double getY() {
        return v.getLen()*Math.sin(v.getAngle());
    }
}
Discussion

• Why not just change the adaptee to have the new interface?
  ▪ There might be other code that relies on the current adaptee interface
  ▪ The adaptee might be code someone else “owns”
    - Either externally, e.g., some open source code from GitHub
    - Or internally, e.g., another group in your company

• Why not duplicate the adaptee and change its interface?
  ▪ Okay temporary, but what happens as the adaptee evolves
  ▪ Need to continually maintain your “shadow” copy of the adaptee and apply your changes to it
  ▪ Likely more painful than maintaining adapter because adapter is written to the public interface
Discussion (cont’d)

• Why not create the adapter by subclassing?
  ▪ Subclass can easily add a few more methods
  ▪ Particularly attractive if adapter has lots of overlapping methods; no need to rewrite them as delegators
  ▪ But, creates tight coupling between adapter and its superclass
Proxy Pattern

• Prevent object from being accessed directly
  ▪ Introduce proxy object to mediate requests
  ▪ Most likely, proxy object should own proxied object
    - No way to get to proxied object except through proxy
  ▪ Guarantees complete mediation, i.e., all accesses go through proxy

• Use cases
  ▪ Access control: check client has permission to call methods
  ▪ Virtual proxy: don’t create proxied object until used
    - Useful if object creation is expensive
  ▪ Communication proxy: object conceptually lives on a remote system, hide that fact from client
    - It’s a bad idea to hide it completely, since clients must worry about network failure
Proxy Pattern Example

class NetworkConnection {
    String getPage(URL u) {
        ...
    }
}

class SafeNetworkConnection {
    private NetworkConnection c;
    String getPage(URL u) {
        if (Safe.check(u)) {
            return c.getPage(u);
        } else {
            return new SuspiciousPageWarning();
        }
    }
}
Proxy Pattern Class Diagram

- Like adapter, but interface doesn’t change
  - **Proxy** and **RealSubject** both implement **Subject**
Discussion

• Both proxy and adapter are a bit of a hack
  ▪ Might be hard to sustain long-term
  ▪ If the adaptee/proxied class is not intended for the adapted/proxied use, it might change in ways incompatible with it

• Ideal: these are temporary solutions that will eventually be eliminated through long-term changes
  ▪ Convince the adaptee/proxied class to change
  ▪ If functionality diverges significantly, implement your own version of adaptee/proxied class with features you want

• Line between adapter/proxy unclear
  ▪ What if we both adapt and add proxy features? Then maybe it’s just a “wrapper”
Decorator Pattern

• Problem:
  ▪ Want to add several different pieces of functionality to object
  ▪ Want to combine these pieces \textit{without} making classes for all possible combinations
  ▪ Want to decide \textit{at run time} what the combinations are

• Solution: The decorator pattern
  ▪ Like a proxy—implements the same interface
    - That way, multiple decorators can be combined
  ▪ But, adds additional functionality
package java.io;
class Reader { ... }  
class BufferedReader { ... }

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

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

- **LineNumberReader** is a decorator for **Reader**
  - It wraps an instance of **Reader**
  - Implements the same interface
    - Can use it wherever a **Reader** is expected
  - It adds functionality (**getLineNumber()**)  
    - Can access the functionality either through **LineNumberReader** type or by downcasting to that type
- Wrapping happens at runtime
  - When we create a **Reader**, we don’t need to allocate it as a **LineNumberReader**
  - We can wrap it some time later
interface Window { void draw(); }

class BorderedWindow implements Window {
    Window inner;
    BorderedWindow(Window inner) { this.inner = inner; }
    void draw() { inner.draw(); /* and draw border */ }
}

class ScrollingWindow implements Window {
    Window inner;
    ScrollingWindow(Window inner) { this.inner = inner; }
    void draw() { inner.draw(); /* and draw scrollbar */ }
}

• Can mix and match borders and scrolling
  - Even without using multiple inheritance
Removing Functionality

- Can’t add, remove, or replace list elements
  - Removing functionality via decoration
  - (But can mutate list elements themselves if they have mutable fields)
- It’s slightly awkward that we now have a `List` that’s behaviorally not a list in that some methods can’t actually be called

```java
interface List {
    static List<E> copyOf(Collection<E> coll);
    // returns unmodifiable List containing els of coll
}
```
Decorator Pattern Discussion

• Advantages
  ■ Fewer classes than with static inheritance
    - Don’t need to define classes for combinations of decorators
  ■ Dynamic addition/removal of decorators
  ■ Keeps root classes simple

• Disadvantages
  ■ Requires lots of layers of objects
    - Adds overhead through extra method calls, extra object allocations
  ■ Still need to have a common interface for all decorators

• Overall, unclear if decorator pattern is best choice
  ■ Might be better in practice to make a single class with all functionality, and use a field to keep track of which functionality is enabled
Behavioral Patterns
Observer Pattern

- Problem: One object must be consistent with another’s state

- Solution:
  - One object is the subject, it holds the state
  - Another object is the observer, it wants to know when the subject’s state changes
  - Whenever the subject changes, notify the observer
Observer Pattern Example: GUIs

When the button’s state changes (via a click), the Button will call the registered handler

This pattern is very common in GUIs

```java
// From Java Swing
class AbstractButton {
    void addActionListener(ActionEvent l) {
        ...
    }
}
class JButton extends AbstractButton {
    ...
}
interface ActionListener {
    void actionPerformed(ActionEvent e);
}

class MyListener {
    void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
}

JButton b = new JButton("Click me!");
b.addActionListener(new MyListener());
```
Observer Class Diagram

Client

Subject
private observers
attach(Observer)
detach(Observer)
notify()

for (Observer o:observers) {
  o.update(new Event(state_change))
}

Observer
update(Event)

observers.add(o)
observers.remove(o)
Example Observers in Android

• Android LifeCycle: three methods called at various points of app startup
  - Depending on whether launched (onCreate), on screen (onStart), or in the foreground (onResume)

```java
class MyActivity extends Activity {
    // an app screen
    void onCreate(Bundle b) { ... }
    void onStart() { ... }
    void onResume() { ... }
}
```

• Receive notifications of location changes

```java
interface LocationListener {
    void onLocationChanged(Location loc); ...
}
```
Observer Design Choices

• Where is list of observers stored?
  ▪ Typically in subject

• How much is communicated to observer?
  ▪ Easiest: an observer only observes a single kind of event
  ▪ For multiple events, pass an object (e.g., `ActionEvent`)
    - Or use multiple observer methods, e.g., `onCreate`, `onStart`, `onResume`
  ▪ Or, observer inspects subject to figure out what changed

• Who triggers the update?
  ▪ State-setting operations of the subject
  ▪ Does every state change trigger an event?
    - E.g., `onLocationChange` is not called instantly on a location change

• Granularity of events that can be observed
  ▪ Notified on any state change? Only certain state changes?
Abstract Syntax Trees (ASTs)

- An *abstract syntax tree* is a data structure representing some program code
  - Example: \((3+4)*5\)

```
Parse Tree

Abstract Syntax Tree
```

```
( E + E )
  E      E
  |      |
  3      4
```

```
E * E
  |  |
  5 +
```

```
+ 5
  |
  |
3 4
```
interface Expr {
}

class IntExpr implements Expr {
    int val;
    IntExpr(int val) { this.val = val; }
}

class AddExpr implements Expr {
    Expr left, right;
    AddExpr(Expr left, Expr right) { this.left=left;
      this.right=right; }
}

class MultExpr implements Expr {
    /* Similar to AddExpr */
}

Expr e = new MultExpr(new AddExpr(new IntExpr(3),
                            new IntExpr(4)),
                        new IntExpr(5));

// e = (3+4)*5
Traversal Patterns

• In general, we could have many more expressions
  ▪ More operators, e.g., subtraction, division, etc
  ▪ Conditionals
  ▪ Variables
  ▪ Assignments
  ▪ Method calls
  ▪ etc.

• We also might want to implement several computations over ASTs
  ▪ Evaluate
  ▪ toString()
  ▪ Typecheck
  ▪ …
int eval(Expr e) {
    if (e instanceof IntExpr) {
        IntExpr ie = (IntExpr) e;
        return ie.val;
    } else if (e instanceof AddExpr) {
        AddExpr ae = (AddExpr) e;
        return eval(ae.left) + eval(ae.right);
    } else if (e instanceof MultExpr) {
        MultExpr me = (MultExpr) e;
        return eval(me.left) * eval(me.right);
    }
}

- Variation: put each case in a method
  - ...if (e instanceof IntExpr) { return eval((IntExpr) e); }...
  - int eval(IntExpr e) { return e.val; }
// could also use overloading
int eval(IntExpr e) { return e.val; }
int eval(AddExpr e) {
    return eval(e.left) + eval(e.right);
}
int eval(MultExpr e) {
    return eval(e.left)*eval(e.right);
}
int eval(Expr e) {
    if (e instanceof IntExpr) {
        return eval((IntExpr) e);
    } else if (e instanceof AddExpr) {
        return eval((AddExpr) e);
    } else if (e instanceof MultExpr) {
        return eval((MultExpr) e);
    }
}
interface Expr {
    int eval();
}

class IntExpr implements Expr {
    int eval() { return val; }
}

class AddExpr implements Expr {
    int eval() { return left.eval() + right.eval(); }
}

class MultExpr implements Expr {
    int eval() { return left.eval() + right.eval(); }
}
Tradeoffs

• Functional-style traversal
  ▪ Code for single operation grouped together
  ▪ Code for different operations separated
  ▪ Easy to add operations
  ▪ Hard to add classes, need to modify every operation
  ▪ Need to duplicate conditional tests for every operation
    - And cascaded `if-then-else`s might not be that efficient

• OO-style traversal
  ▪ Code for single operation spread across classes
  ▪ All operations for single class grouped together
  ▪ Hard to add operations, need to modify every class
  ▪ Easy to add classes, just go through and implement all ops
Implementing OO Traversal Once

- What if we want to
  - Use the OO-style traversal
  - Implement multiple operations (eval, toString, etc)
- Only write the traversal code once

```java
interface Expr {
}
class IntExpr implements Expr {
    ...
}
class AddExpr implements Expr {
    ...
}
class MultExpr implements Expr {
    ...
}

interface Visitor {
    ...
}
class Eval implements Visitor {
    ...
}
class ToString implements Visitor {
    ...
}
```
The Problem: Single Dispatch

Here’s what we want to do:

\[
\text{Expr } \text{ex} = \text{new MultExpr}(\ldots); \\
\text{Eval } \text{ev} = \text{new Eval}(); \\
// \text{Use } \text{ev} \text{ to evaluate } \text{ex}
\]

Which method should we start running?
- Clearly, \text{Eval’s} method for \text{MultExpr}

So, the method we want to call depends on both
- The run-time type of \text{ex}
- The run-time type of \text{ev}

Standard use of dynamic dispatch can’t handle this
- Calling \text{ev.m(ex)} can only choose which \text{m} based on \text{ev}, not based on \text{ex}
Double Dispatch Problem

interface I
class A implements I { }
class B implements I { }

interface Z
class X implements Z { }
class Y implements Z { }

• Suppose
  ▪ We have an I and a Z
  ▪ We want to invoke method depending on those objects’ run-time types (classes)
  ▪ So we are choosing among four methods
    - (A, X), (A, Y), (B, X), (B, Y)
interface I {
}
class A implements I {
    void accept(Z z) { z.visitA(this); }
}
class B implements I {
    void accept(Z z) { z.visitB(this); }
}

interface Z
class X implements Z {
    void visitA(I i) { /* this is X, i is A */ }
    void visitB(I i) { /* this is X, i is B */ }
}
class Y implements Z {
    void visitA(I i) { /* this is Y, i is A */ }
    void visitB(I i) { /* this is Y, i is B */ }
}
Double Dispatch, Pictorally

\[ i \in \{A,B\} \quad z \in \{X,Y\} \]

- Use dynamic dispatch on one value, then flip args and use dynamic dispatch on the other value
Visitor Pattern

• Combine two things
  - External iteration, usually over a tree structure
    - We have two objects: the tree and the visitor
  - Double dispatch
    - So that we can call a method depending on the run-time type of a tree node and which visitor object is doing the visiting

```java
class SomeExpr implements Expr {
    void accept(Visitor v) {
        // postorder traversal
        for each child of this node { child.accept(v); }
        v.visitSomeExpr(this);
    }
}
class SomeVisitor implements Visitor {
    void visitSomeExpr(SomeExpr e) { ... }
    void visitOtherExpr(OtherExpr e) { ... }
}
```
interface Expr {
    void accept(Visitor v);
}
class IntExpr implements Expr{
    void accept(Visitor v) {
        v.visitIntExpr(this);
    }
}
class AddExpr implements Expr{
    void accept(Visitor v) {
        left.accept(v);
        right.accept(v);
        v.visitAddExpr(this);
    }
}
class MultExpr implements Expr{
    void accept(Visitor v) {
        left.accept(v);
        right.accept(v);
        v.visitMultExpr(this);
    }
}

interface Visitor {
    // ...
}
class Eval implements Visitor {
    void visitIntExpr(IntExpr e) {
        e.evald = e.val;
    }
    void visitAddExpr(AddExpr e) {
        e.evald = e.left.evald +
                   e.right.evald;
    }
    void visitMultExpr(AddExpr e) {
        e.evald = e.left.evald *
                e.right.evald;
    }
}
Expr e = new MultExpr(new AddExpr(new IntExpr(3),
    new IntExpr(4)),
    new IntExpr(5));

Visitor v = new Eval();
e.accept(v); // calls MultExpr’s accept
e.left.accept(e); // calls AddExpr’s accept
    e.left.left.accept(e); // call IntExpr(3)’s accept
    e.left.left.evald = 3;
    e.left.right.accept(e);
    e.left.right.evald = 4;
    e.visitAddExpr(e);
    e.left.evald = 7; // 3+4
    e.right.accept(e); // call IntExpr(5)’s accept
    e.right.evald = 5;
v.visitMultExpr(e);
e.evald = 12 // 7+5
interface Expr {
    void accept(Visitor v);
}

class IntExpr implements Expr{
    void accept(Visitor v) {
        v.visit(this);
    }
}

class AddExpr implements Expr{
    void accept(Visitor v) {
        left.accept(v);
        right.accept(v);
        v.visit(this);
    }
}

class MultExpr implements Expr{
    void accept(Visitor v) {
        left.accept(v);
        right.accept(v);
        v.visit(this);
    }
}

// Just have a single method name, visit, and rely on overloading to resolve which visit method is called

interface Visitor {
    ... 
}

class Eval implements Visitor {
    void visit(IntExpr e) {
        e.evald = e.val;
    }
    void visit(AddExpr e) {
        e.evald = e.left.evald + e.right.evald;
    }
    void visit(MultExpr e) {
        e.evald = e.left.evald * e.right.evald;
    }
}
Challenges with Visitors

• Visit order is fixed by accept method
  ■ What if we want to visit in preorder? inorder?
  ■ Could do the following, but then visitors are big

```java
void accept(Visitor v) {
    v.visitPre(this);
    left.accept(v);
    v.visitIn(this);
    right.accept(v);
    v.visitPost(this);
}
```

• visit methods needs to store results elsewhere
  ■ In this, in custom data structure or in the data structure

• Visitors are popular but are pretty clunky
  ■ Pattern matching is a much better solution
More Patterns?
The following aren’t usually called “design patterns,” but they kind of are...
OO Programming in C

• C is not object-oriented
  ▪ Should that stop us from using objects in C? No!

```c
enum clazz {A, B};
typedef struct PrintI { // an interface
    enum clazz id;
    void (*print)(void);
} *PrintI;

void printA(void) { printf("I'm an A\n"); }  
PrintI newA(void) {
    PrintI o = malloc(sizeof(struct PrintI));
    o->id = A; o->print = printA;
    return o;
}

PrintI a = newA();
a->print(); // dynamic dispatch!
```
Imperative Programming in Haskell

- Haskell is a *pure* functional programming language
  - Does not allow changing value of a variable or of heap cell
- *Monads*: program imperatively in pure func. setting
  - Idea: pass state around to all functions

```java
class State {
    public final int x, y;
    State(int x, String y) { this.x = x; this.y = y; }
}
State theWorld;
theWorld = new State(0, "");
theWorld = newState(theWorld.x + 1, theWorld.y);
theWorld = m(42, theWorld);

State m(int z, State theWorld) {
    return new State(theWorld.x + z, theWorld.y);
}
```

- Monads include syntactic sugar to avoid the boilerplate
Convention over Configuration

• **A framework** is a code base that supports the development of a certain class of applications
  - E.g., Ruby on Rails is a framework for building web apps
  - Unlike a library, which is called by an app, the framework runs on the “outside” and executes the app code

• Frameworks tend to be broad and shallow
  - Supports many different bits and pieces of functionality
    - E.g., Rails includes support for: accessing database, rendering web pages, running different web servers, sending email, storing persistent objects, testing apps, securing apps, supporting JavaScript, etc, etc

• How can anyone program something that complex?
  - **Convention over configuration** = developer only needs to specify non-standard parts of the app
Conv. over Config. w/Rails Routing

- Above specifies standard behavior:
  - Requesting URL / will invoke `TalksController#index`
  - When `TalksController#index` finished, it will send `views/talks/index.html.erb` back to the user
  - (Same for `show`, `edit`, etc)

```ruby
# config/routes.rb
Talks::Application.routes.draw do
  resources :talks
end

# app/controllers/talks_controller.rb
class TalksController < ApplicationController
def index ... end
end

# app/views/talks/index.html.erb
```