Design Patterns

With a little help from slides by Bill Pugh et al at University of Maryland

What are design patterns?

- Design pattern is a problem & solution in context
- Design patterns capture software architectures and designs
  - Not code reuse
  - Instead solution/strategy reuse
  - Sometimes interface reuse

Elements of Design Patterns

- Pattern Name
- Problem statement - context where it might be applied
- Solution - elements of the design, their relations, responsibilities, and collaborations.
  - Template of solution
- Consequences: Results and trade-offs

Example: Iterator Pattern

- Name: Iterator or Cursor
- Problem statement
  - How to process elements of an aggregate in an implementation independent manner
- Solution
  - Aggregate returns an instance of an implementation of Iterator interface to control 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

Goals of Patterns

- To support reuse, of
  - Successful designs
  - Existing code *(though less important)*
- To facilitate software evolution
  - Add new features easily, without breaking existing ones
- Design for change!
- Reduce implementation dependencies between elements of software system.

Taxonomy of Patterns

- 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.

Creational Patterns

- Singleton
  - Ensure a class only has one instance, and provide a global point of access to it.
  - *We used with BinaryTree by not having public constructor for EmptyBinaryTree*
- Abstract Factory
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - We used something like this in Garden assignment with newPlant() method.

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

Creational Patterns

Abstract Factory

- Context:
  - System should be independent of how pieces created and represented
  - Different families of components
  - Must be used in mutually exclusive and consistent way
  - Hide existence of different families from clients

Abstract Factory *(cont.)*

- Solution:
  - Create interface w/ operations to create new products of different kinds
  - Multiple concrete classes implement operations to create concrete product objects.
  - Products also specified w/interface
  - Concrete classes for each interface and family of products.
  - Client uses only interfaces

Abstract Factory *(cont.)*

- Examples:
  - GUI Interfaces:
    - Mac
    - Windows XP
    - Unix
  - Garden:
    - Text version
    - Graphical version

Abstract Factory Consequences

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

Structural Patterns

Proxy Pattern

- 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 and/or location of protected object

Uses of Proxy Pattern

- Virtual proxy: impose a lazy creation semantics, to avoid expensive object creations when strictly unnecessary. *(Getting image from disk.)*
- Monitor proxy: impose security constraints on the original object, say by making some public fields inaccessible.
- Remote proxy: hide the fact that an object resides on a remote location.

Decorator Pattern

- Motivation
  - Want to add responsibilities/capabilities to individual objects, not to an 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.

Decorator Pattern

- A decorator forwards requests to its encapsulated component and may perform additional actions before or after forwarding.
- Can nest decorators recursively, allowing unlimited added responsibilities.
- Can add/remove responsibilities dynamically

Decorator Pattern Consequences

- 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

Decorator Example

```
FileReader frdr= new FileReader(filename);

LineNumberReader lrdr =
                     new LineNumberReader(frdr);

String line;

line = lrdr.readLine()
while (line != null){
   System.out.print(lrdr.getLineNumber() +
                     ":\t" + line);
   line = lrdr.readLine()
}
```

Behavioral Patterns

Template 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
  - 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

Observer Pattern

- Problem
  - Objects that depend on a certain subject must be made aware of when that subject changes
- E.g. receives an event, changes its local state, etc.
  - These objects should not depend on the implementation details of the subject
- They just care about how it changes, not how it's implemented.

Observer Pattern

- Solution structure
  - Subject is aware of its observers (dependents)
  - Observers are notified by the subject when something changes, and respond as necessary
  - *Examples: Java event-driven programming*
- Subject
  - Maintains list of observers; defines a means for notifying them when something happens
- Observer
  - Defines the means for notification (update)

Observer Pattern

```
class Subject {
  private Observer[] observers;

  public void addObserver(Observer newObs){... }

  public void notifyAll(Event evt){
    forall obs in observers do
      obs.process(this,evt)}
}

class Observer {
  public void process(Subject sub, Event evt) {
        ... code to respond to event ...
  }
}
```

Observer Pattern Consequences

- Low coupling between subject and observers
  - Subject indifferent to its dependents; can add or remove them at runtime
- Support for broadcasting
- Updates may be costly
  - Subject not tied to computations by observers

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: use dynamic dispatch

State Pattern

- Encode different states as objects with the same interface.
- To change state, change the state object
- Methods delegate to state object

State Pattern Example

```
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();
}
```

State Pattern Example

```
class State1 implements State {
  public 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 {
  public 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 booleanaccept() {return true;}
}
```

State Pattern

- 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 implement the State interface, or
  - New states can extend other states
- Override only selected functions

Visitor Pattern

- Problem: 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
- Flawed solution: implement each analysis as a method in each object
  - Follows idea objects are responsible for themselves
  - But many analyses will occlude the objects' main code
  - Result is classes hard to maintain

Visitor Pattern

- We define each analysis as a separate Visitor class
  - Defines operations for each element of a structure
- A separate algorithm traverses the structure, applying a given visitor
  - But, like iterators, objects must reveal their implementation to the visitor object
- Separates structure traversal code from operations on the structure
  - Observation: object structure rarely changes, but often want to design new algorithms for processing

Visitor Pattern

- One class hierarchy for object structure
  - AST in compiler
- One class hierarchy for each operation family, called visitors
  - One for typechecking, code generation, pretty printing in compiler

Visitor Pattern Consequences

- Gathers related operations into one class
- Adding new analyses is easy
  - New visitor for each one
  - Easier than modifying the object structure
- Adding new concrete elements is difficult
  - must add a new method to each concrete Visitor subclass

Visitor Traversal Choices

- Traversal in object structure (typical)
  - Define operation that performs traversal while applying visitor object to each component
- Traversal implemented in visitor itself
  - E.g., perform processing at this node, then pass visitor to children nodes.
- Traversal code replicated in each concrete visitor
  - External Iterator

Designing with Patterns

- How do you know which patterns to use?
- What if you choose the wrong pattern?
  - I.e. your code doesn't evolve the way you thought it would.
- What if all your work to make things extensible via patterns never pays off?
  - I.e. your code doesn't change in the way you thought it would.
- Choosing the right pattern implies prognostication

Designing with Patterns

- Some design patterns are immediately useful
  - Observer, Decorator
- Some are not immediately useful, but you think they might be
  - You anticipate changing things later -- prognostication
- Recently popular philosophy: XP
  - Design for your immediate needs
  - When needs change, redesign your code to match
  - Use extensive testing to validate frequent changes