Lecture 40: Design Patterns

CS 62
Fall 2017
Kim Bruce & Alexandra Papoutsaki

What is a Pattern?

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution"

Christopher Alexander on architecture patterns

"Patterns are not a complete design method; they capture important practices of existing methods and practices uncodified by conventional methods"

James Coplien

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.
  - *Often used in recursively defined classes (e.g., lists & trees) where don't have public constructor, just public constant defined using private constructor*
- Abstract Factory
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - *Allows hiding actual constructor call in method definition*

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
    - *Abstract superclass*
- 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

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

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

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

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