Lecture 41: Design Patterns

CS 62

Spring 2018

Alexandra Papoutsaki & William Devanny

Patterns in Architecture

A Pattern Language: Towns, Buildings, Construction (1977) - Christopher Alexander, Sara Ishikawa, and Murray Silverstein

> “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"

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

Software Design Patterns

- Experimentation with applying patterns to programming during the late 80s

- Popularized by the *Gang of Four (GoF)* book:
  - Gamma, Helm, Johnson, Vlissides (1995).
    Design Patterns: Elements of Reusable Object-Oriented Software.

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

Design Pattern structure

- 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

- https://sourcemaking.com/design_patterns

Classification

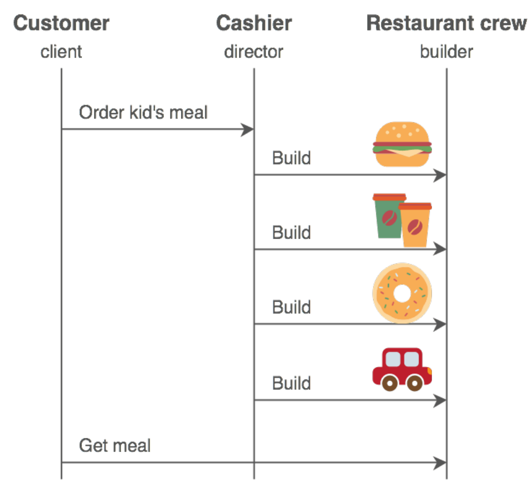
1. Creational Design Patterns
    - concern the process of object creation
2. Structural Design Patterns
    - deal with the composition of classes or objects
3. Behavioral Design Patterns
    - characterize the ways in which classes or objects interact and distribute responsibility

Creational Patterns

- **Abstract Factory/Method**
  Creates an instance of several derived/families of classes
- **Builder**
  Separates object construction from its representation
- **Prototype**
  A fully initialized instance to be copied or cloned
- **Singleton**
  A class of which only a single instance can exist

Builder

- **Intent**
  Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Problem**
  An application needs to create the elements of a complex aggregate.
- **Example**
  Ordering meals

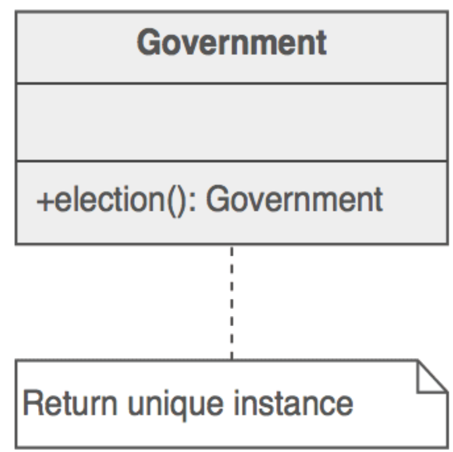


Prototype

- **Intent**
  Avoid the inherent cost of creating objects with new
- **Problem**
  Application "hard wires" the class of object to create in each "new" expression.
- **Examples**
  Chess initialization

Singleton

- **Intent**
  Ensure a class has only one instance, and provide a global point of access to it.
- **Problem**
  Application needs one, and only one, instance of an object.
- **Example**
  - US President
  - Java.lang.System



Structural Patterns

- **Adapter**
  Match interfaces of different classes
- **Bridge**
  Separates an object's interface from its implementation
- **Composite**
  A tree structure of simple and composite objects
- **Decorator**
  Add responsibilities to objects dynamically
- **Facade**
  A single class that represents an entire subsystem
- **Flyweight**
  A fine-grained instance used for efficient sharing
- **Proxy**
  An object representing another object

Decorator

- **Intent**
  Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **Problem**
  You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.
- **Solution**
  Enclose the component in another object that adds the responsibility/capability
  The enclosing object is called a decorator.

Decorator

- 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 examples

- Pizza topings
- Java I/O
- ```
  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

- **Chain of responsibility**
  A way of passing a request between a chain of objects
- **Command**
  Encapsulate a command request as an object
- **Interpreter**
  A way to include language elements in a program
- **Iterator**
  Sequentially access the elements of a collection
- **Mediator**
  Defines simplified communication between classes
- **Memento**
  Capture and restore an object's internal state
- **Null Object**
  Designed to act as a default value of an object
- **Observer**
  A way of notifying change to a number of classes
- **State**
  Alter an object's behavior when its state changes
- **Strategy**
  Encapsulates an algorithm inside a class
- **Template method**
  Defer the exact steps of an algorithm to a subclass
- **Visitor**
  Defines a new operation to a class without change

Observer

- **Intent**
  Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Problem**
  Objects that depend on a certain subject must be made aware of when that subject changes.
- **Example**
  - 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

- 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

Iterator

- **Intent**
  Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Problem**
  Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.
- **Solution**
  Aggregate returns an instance of an implementation of Iterator interface to control iteration.

Iterator

- 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 its creation

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 (now called *agile*)
    - Design for your immediate needs
    - When needs change, redesign your code to match
    - Use extensive testing to validate frequent changes