Software Engineering

Design Patterns -- Introduction

Software Engineering 2012-2013
Sources

1. **Classical text:** *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson, Vlissides, (GoF), 1995


4. *Patterns home page*:  
   [http://hillside.net/patterns/](http://hillside.net/patterns/)
What is a Pattern?

- The term *design pattern* was coined in the mid-1970’s by Christopher Alexander, an architect, who abstracted common *design patterns* in architecture and formalized a way of describing the patterns in a “pattern language.”

- A *design pattern* addresses both a problem that occurs *repeatedly* in our environment and the core of the solution to that problem—but at a level of generality that the solution can be applied many times without ever being the same in its particulars.
What is a Pattern?

- "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

  *A Pattern Language*, Christopher Alexander, 1977
Example: A Place To Wait (Alexander 1977)

**Fundamental problem:**
- How to spend time wholeheartedly and
- Still be on hand when doctor, airplane etc arrive

**The solution:**
- Fuse the waiting with other activity that keeps them in earshot
  - Playground beside Pediatrics Clinic
  - Horseshoe pit next to terrace where people waited
- Allow the person to become still meditative
  - A window seat that looks down on a street
  - A protected seat in a garden
  - A dark place and a glass of beer
  - A private seat by a fish tank
Example: A Place To Wait  (Alexander 1977)

The Solution:

"In places where people end up waiting create a situation which makes the waiting positive. Fuse the waiting with some other activity - newspaper, coffee, pool tables, horseshoes; something which draws people in who are not simple waiting. And also the opposite: make a place which can draw a person waiting into a reverie; quiet; a positive silence"

The active part might have a window on the street - STREET WINDOWS (164), WINDOW PLACE (180), a cafe - STREET CAFE (88), games, positive engagements with the people passing by - OPENING TO THE STREET (165). The quiet part might have a quiet garden seat - GARDEN SEAT (176), a place for people to doze SLEEPING IN PUBLIC (94), perhaps a pond with fish in it - STILL WATER (71). To the extent that this waiting space is a room, or a group of rooms, it gets its detailed shape from LIGHT ON TWO SIDES OF EVERY ROOM (159) and THE SHAPE OF INDOOR SPACE (191).
What are design patterns? (J.W. Cooper)

- **Recurring solutions to design problems** you see over and over. (Alpert et al, 1998)

- **A set of rules describing how to accomplish certain tasks** in the realm of software development. (Pree, 1994)

- **Focus on reuse of recurring architectural design themes** (Coplien and Schmidt, 1995)

- **Address a recurring design problem** that arises in a specific context and presents a solution to it (Buschmann. et al, 1996)

- **Identify and specify abstractions that are above the level of single classes or instances, or of components.** (GoF)
More Paradigms of design patterns

- Problem Analysis (Fowler)
- Enterprise systems (Fowler)
- Responsibility assignment in system design (Larman)
- User interfaces.
- Web site construction.
- Software design (GoF, Coplien, and others)
What is a Software design pattern?
(F. Tip)

- Related question: **what is the difference between experienced and inexperienced software designers?**
  - experienced designers:
    - know from experience what works and what doesn’t
    - able to recognize “standard” design problems and apply “proven” solutions to them

- Definition of a software design pattern:
  
  *a description of communicating classes and objects that are customized to solve a general design problem in a particular context*
Stages of Design Pattern awareness
(J. Vlissides)

Ignorance  consternation  initiation  understanding  familiarity  benefit
Learn to apply design patterns to the design process (J. Vlissides)

- find the right patterns
- understand (un)applicability
- see when and how to bend a pattern
- evaluate design trade-offs effectively

- Learn by (counter) example
Software Design Patterns - Motivation

- OOD methods emphasize **design notations**
  - Fine for specification, documentation
- **Good OO designers** rely on lots of experience
- **Most powerful reuse is design reuse**
  - Match problem to design experience
Recurring Design Structures

- OO systems exhibit recurring structures that promote
  - abstraction
  - Flexibility
  - Modularity
  - elegance

- Therein lies valuable design knowledge

- **Problem:**
  - capturing, communicating, & applying this knowledge
Design Patterns

• A design Pattern
  • abstracts a recurring design structure
  • comprises class and/or object
  • dependencies
  • structures
  • interactions
  • Conventions

• names & specifies the design structure explicitly purify design experience
Elements of Design Patterns (F. Tip)

- A design pattern has 4 elements:
  - a name (e.g., “Abstract Factory” or “Visitor”)
  - the problem that the pattern addresses
  - the solution: the program constructs that are part of the pattern
  - the consequences: the results and tradeoffs of applying the pattern

- other factors:
  - problem & solution have been observed in practice
  - choice of implementation language important
Goals

- Codify good design
  - distill & generalize experience
  - aid to novices & experts alike
- Give design structures explicit names
  - common vocabulary
  - reduced complexity
  - greater expressiveness
- Capture & preserve design information
  - articulate design decisions succinctly
  - improve documentation
- Facilitate restructuring/refactoring
  - patterns are interrelated
  - additional flexibility
Classifying Design Patterns (F. Tip)

- **purpose**: what a pattern does
  - **creational**: concerned with creation of objects
  - **structural**: related to composition of classes or objects
  - **behavioral**: related to interaction and distribution of responsibility

- **scope**
  - **class-level**: concerned with relationship between classes and their subclasses
  - **object-level**: concerned with object relationship (more dynamic, may be changed at run-time)
## GoF Design Patterns Classified (F. Tip)

<table>
<thead>
<tr>
<th>Scope</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td>Factory Method</td>
<td>Adapter (class)</td>
<td>Interpreter Template Method</td>
</tr>
<tr>
<td><strong>Object</strong></td>
<td>Abstract Factory Builder Prototype Singleton</td>
<td>Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy</td>
<td>Chain of Resp. Command Iterator Mediator Memento Observer State Strategy Visitor</td>
</tr>
</tbody>
</table>
Principles of Object-Oriented Design (F. Tip)

*Program to an interface, not an implementation.*

*Favor object composition over class inheritance.*
Class vs. Interface Inheritance (F. Tip)

- **Class inheritance** defines an object’s implementation in terms of another object’s implementation
  - mechanism for code & representation sharing
- **Interface inheritance** describes when an object can be used in place of another (subtyping)
- many languages (e.g., C++) don’t make this distinction, but Java does

An object's class defines how the object is **implemented**. The class defines the object's internal state and the implementation of its operations.

An object's type only refers to its **interface**—the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type.
Class vs. Interface Inheritance (2) (F. Tip)

• benefits of class inheritance
  • extend an application’s functionality by reusing functionality in parent classes
  • lets you get new implementations almost for free, inheriting most of what you need from existing classes

• benefits of interface inheritance
  • clients remain unaware of specific types of objects they use, and of the classes that implement these objects

• using interface inheritance greatly reduces dependencies between subsystems
  • reduces the impact of changes
Mechanisms for Reusing Functionality (F. Tip)

Inheritance versus Composition

- **class inheritance**: define implementation of one class in terms of another
  - often referred to as *white-box reuse*: internals of parent class visible to extending class
    - *class inheritance breaks encapsulation*

- **object composition**: compose objects to get new, more complex functionality
  - implemented by giving objects references to other objects; access these objects via interfaces
  - requires that objects have well-defined interfaces
  - often called *black-box reuse*: no internal details of objects are visible to the class that uses them
    - *composition does not break encapsulation*
Pros & Cons of Class Inheritance (F. Tip)

• **Advantages:**
  • directly supported by the programming language, hence easy to use
  • makes it easy to modify the reused implementation (by simply overriding a few methods)

• **Disadvantages:**
  • cannot change inherited functionality at run-time, because inheritance is fixed at compile-time
  • parent classes define at least part of their subclasses’ physical representation, and subclasses are exposed to details of their parent’s implementation
    • implementation of subclass becomes very tightly coupled with implementation of parent
    • change in parent is likely to require changes in subclass
Delegation (F. Tip)

- Delegation is an alternative to inheritance:
  - two objects are involved: a receiving object delegates an operation to its delegate
  - analogy: a subclass that defers a request to its parent class
- suppose an object of type C wants to delegate a method f() to an object of type D:
  - class D defines method f()
  - class C needs to contain a reference field d to a D-object, which needs to be initialized
  - C needs a forwarding method f() that calls f() on d
Delegation: Example (F. Tip)

```java
public class Rectangle {
    private int width;
    private int height;
    public int area(){ return width * height; }
}

public class Window {
    private Rectangle rect;
    public int area(){ return rect.area(); }
}

public class WindowClient {
    void someOperation(Window w){ ... w.area() ... }
}
```

class Window delegates its area() operation to a Rectangle instance
(Ab)use inheritance for the same purpose (F. Tip)

```java
public class Rectangle {
    private int width;
    private int height;
    public int area(){ return width * height; } } 
```

```java
public class Window extends Rectangle {
    private Rectangle rect;
    // method area() inherited from superclass
}
```

```java
public class WindowClient {
    void someOperation(Window w){ ... w.area() ... } 
}
```
Why use delegation? (F. Tip)

• inheritance can be more convenient:
  • only define method f() once
  • no need to forward calls
  • somewhat more efficient

• however, it is less flexible:
  • cannot change the implementation of f() after creating the object
  • in languages with single inheritance, you can only inherit methods from one superclass
True delegation (F. Tip)

• with inheritance, the method in the superclass can use dynamic dispatch to invoke methods in a subclass
• with delegation, this requires some extra work:
  • pass receiving object’s this pointer as an argument to the delegate
  • delegate invokes methods on this reference when it needs to invoke methods on receiving object
• this form of delegation is called true delegation
• for an example of true delegation, see the “State” design pattern
When to use inheritance? (F. Tip)

- generally speaking, use inheritance for:
  - is-a relationships that don’t change over time
  - situations where the class containing the actual operation is abstract

- generally speaking, use delegation for:
  - has-a, part-of relationships
  - is-a-role-played-by relationships
  - relationships that change over time
  - situations where multiple inheritance would be needed (if language doesn’t allow MI)
Designing for change (F. Tip)

- many design patterns introduce flexibility to avoid common causes of redesign such as:
  - creating an object by specifying a class explicitly
  - dependence on specific operations
  - dependence on hardware/software platform
  - dependence on object representations or implementations
  - algorithmic dependencies
  - tight coupling
  - extending functionality by subclassing
  - inability to alter classes conveniently
Designing for change (R. Whitney)

Some common design problems that GoF patterns address:

- Creating an object by specifying a class explicitly
  Abstract factory, Factory Method, Prototype

- Dependence on specific operations
  Chain of Responsibility, Command

- Dependence on hardware and software platforms
  Abstract factory, Bridge

- Dependence on object representations or implementations
  Abstract factory, Bridge, Memento, Proxy
Designing for change (R. Whitney)

- Algorithmic dependencies
  
  Builder, Iterator, Strategy, Template Method, Visitor

- Tight Coupling
  
  Abstract factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

- Extending functionality by subclassing
  
  Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

- Inability to alter classes conveniently
  
  Adapter, Decorator, Visitor
GOF: Describing Design Patterns

- **Pattern Name and Classification**
  The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

- **Intent**
  A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

- **Also Known As**
  Other well-known names for the pattern, if any.

- **Motivation**
  A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

- **Applicability**
  What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

- **Structure**
  A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique, interaction diagrams to illustrate sequences of requests and collaborations between objects.
GOF: Describing Design Patterns

- **Participants**
  The classes and/or objects participating in the design pattern and their responsibilities.

- **Collaborations**
  How the participants collaborate to carry out their responsibilities.

- **Consequences**
  How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

- **Implementation**
  What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

- **Sample Code**
  Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

- **Known Uses**
  Examples of the pattern found in real systems. We include at least two examples from different domains.

- **Related Patterns**
  What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?