Patterns

What are Patterns?
Software Patterns
Design Patterns
Architectural Patterns
J2EE
Patterns: Still Evolving ...
What are Patterns?

"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."

Christopher Alexander
A Dictionary Definition

A discernible coherent system based on the intended interrelationships of component parts [Webster Collegiate]

Kinds of Software Patterns

- Systems Engineering → Systems Spec → Context
- (Sw) Reqs. Analysis → SRS → Problem
- (Sw) Arch. Design → SADS → SAD
- (Sw) Detailed Design → SDDS → (SD) Design
- Implementation → Program → Idioms
A key source

Elements of Reusable Object-Oriented Software
[Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides] (GoF)

A catalogue of 23 design patterns

Four essential elements

Name, Problem, Solution, Consequences

Categories of design patterns

By purpose (what does a pattern do?)

- Creational: concerns the process of object creation
- Structural: concerns the composition of classes or objects
- Behavioral: concerns the ways in which classes or objects interact and distribute responsibility (e.g., algorithms and flow of control)

By scope (Is the pattern for classes or instances?)
**Design Patterns: An Example**

✈ **MVC (Model-View-Controller) classes for UI (in Smalltalk-80)**

- Model: application object
- View: screen presentation
- Controller: the way the UI reacts to user input

**Advantages**

- Decoupling of views from models
- Applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others = Observer
OBSERVER  Object Behavioral

Intent
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Also Known As
Dependents, Publish-Subscribe

Motivation
... You don’t want to achieve consistency by making the classes tightly coupled, because that reduces their reusability ...

Applicability
Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other .... lets you vary and reuse them independently
- When a change to one object requires changing others, and you don’t know how many objects need to be changed
- When an object should be able to notify other objects without making assumptions about who these objects are
for all o in observers{
    o->Update()
}

Lawrence Chung

Participants

- Subject: knows its observers. Any number of Observer objects may observe a subject and provides an interface for attaching and detaching Observer objects.

- Observer: defines an updating interface for objects to be notified of changes in a subject.

- ConcreteSubject: stores state of interest to ConcreteObserver objects; sends a notification to its observers when its state changes.

- ConcreteObserver: maintains a reference to a ConcreteSubject object; stores state that should stay consistent with the subject’s state and implements Observer updating interface to keep its state consistent with the subject’s state.
Collaborations

... The following interaction diagram illustrates the collaborations between a subject and two observers:

- **aConcreteSubject**
  - `SetState()`
  - `Notify()`
  - `Update()`
  - `GetState()`
- **aConcreteObserver**
  - `Notify()`
  - `Update()`
  - `GetState()`
- **anotherConcreteObserver**
  - `GetState()`

Consequences

- Abstract coupling between Subject & Observer: All a subject knows is that it has a list of observers.
- Support for broadcast communication
- Unexpected updates: one subject operation may cause a cascade of updates to observers and their dependent objects

Implementation

- Mapping subjects to their observers; Observing more than one subject;
- Who triggers the update?; Dangling references to deleted subjects; ...
Design Patterns: An Example

✈ MVC (Model-View-Controller) classes for UI (in Smalltalk-80)

* Model: application object
* View: screen presentation
* Controller: the way the UI reacts to user input

Advantages

✜ decoupling of views from models
applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others = Observer

✜ nesting of views (control panel of buttons of buttons of ...)
applicable to a more general problem: group objects & treat the group like an individual object = Composite

✜ Controller for changing the way a view responds to user input without changing the visual rep.
applicable to a more general problem: replace an algorithm statically or dynamically = Strategy

✜ specify a default class (for controller) = Factory Method
add scrolling to a view = Decorator
Composite

Object Structural

Intent
Compose objects into tree structures to represent part-whole hierarchies.
Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation
... Graphic applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components ...

Applicability
Use the Composite pattern when

✦ you want to represent part-whole hierarchies of objects

✦ you want to clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
Composite

Structure

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

A typical Composite object structure might look like this:

```
forall g in children
  g.Operation();
```
Motivation (continued)

**Composite**

- **Graphic**
  - Draw()
  - Add(Graphic)
  - Remove(Graphic)
  - GetChild(int)

- **Line**
  - Draw()

- **Rectangle**
  - Draw()

- **Text**
  - Draw()

- **Composite**
  - Operation()
  - Add(Component)
  - Remove(Component)
  - GetChild(int)

forall g in graphics
  g.Draw();

add g to list of graphics
Intent
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Also Known As
Policy

Motivation
Many algorithms exist for breaking a stream of text into lines. Hard-writing all such into the classes that require them isn’t desirable for several reasons:

- Inclusion of the linebreaking code in the client makes it more complex bigger and harder to maintain, esp. if to support multiple linebreaking algorithms
- Different algorithms will be appropriate at different times
- Difficult to add new algorithms and vary existing ones when linebreaking is an integral part of the client

Applicability
Use the Strategy pattern when

- many related classes differ only in their behavior
- need different variants of an algorithm, implemented as a class hierarchy
- an algorithm uses data that the client shouldn’t know about
Strategy

Object Behavioral

Participants

ConcreteStrategy

ConcreteStrategyA

ConcreteStrategyB

Strategy

Context

(design pattern)

ConcreteStrategy (composition)

Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy implements the algorithm using the Strategy interface.

ConcreteStrategy may define an interface that lets Strategy access its data.

Strategy (composition)

Context (composition)

SimpleCompositor, TeXCompositor, ArrayCompositor

Composition

Context

Strategy

AlgorithmicInterface

Strategy

AlgorithmicInterface
# Design Pattern Space

<table>
<thead>
<tr>
<th>Scope</th>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Creational</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factory Method</td>
<td></td>
<td>Interpreter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Template Method</td>
</tr>
<tr>
<td><strong>Object</strong></td>
<td>Abstract Factory</td>
<td>Adapter (object)</td>
</tr>
<tr>
<td>Builder</td>
<td>Bridge</td>
<td>Chain of Responsibility</td>
</tr>
<tr>
<td>Prototype</td>
<td>Composite</td>
<td>Command</td>
</tr>
<tr>
<td>Singleton</td>
<td>Decorator</td>
<td>Iterator</td>
</tr>
<tr>
<td></td>
<td>Facade</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td>Flyweight</td>
<td>Memento</td>
</tr>
<tr>
<td></td>
<td>Proxy</td>
<td>Observer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>State</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visitor</td>
</tr>
</tbody>
</table>
Recall:

\[ A = \langle \text{Com}, I, \text{Con}, P, S, R \rangle \]

Pattern Matching in IQ Test

- choose the closest match to

\[
\begin{align*}
A. & \quad B. & \quad C. & \quad D. \\
\text{Diagram}
\end{align*}
\]

Topological View of Architectural Patterns

\[ P(A) = \langle \text{Com}, I \rangle \]

Modeling View of Architectural Patterns

\[ P(A) = \langle \text{Com}, I, \text{Con}, S, R \rangle \]
J2EE
Java 2 Platform Enterprise Edition

✈ Vision
✦ an open standard
✦ anything Java-related
✦ for implementing and deploying component-based enterprise applications

✈ Commercial platforms
✦ WebLogic (BEA Systems)
✦ WebSphere (IBM)
✦ iPlanet (Sun & NetScape)
✦ JBoss (open source)

✈ Reference application
✦ Pet Store

✈ Services
✦ EJB components
✦ JDBC API
✦ CORBA technology
✦ Java Servlets API
✦ XML technology
J2EE Distributed Multitiered Applications

**EJB:** server-side component containing the business logic of an application. The application clients invoke their methods.
J2EE 5-Tier Model & Design Patterns

Client Tier

Resource Tier

Integration Tier

Business Tier

JMS/JDBC/
DB/external-resources

Presentation Tier

Business Tier

Database Server Machine

Resource Tier

DB/external-resources

J2EE Design Patterns

* JMS = Java Message Service API;  * JDBC = Java Database Connectivity API;

http://java.sun.com/features/2001/03/patterns.html

Lawrence Chung
The problem: Supporting Multiple Enterprise Clients

A Solution: Model-View-Controller Architecture

Design Patterns Catalog

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Access Object</td>
<td></td>
</tr>
<tr>
<td>Fast-Lane Reader</td>
<td>Accelerate read-only data by not using enterprise beans</td>
</tr>
<tr>
<td>Front Controller</td>
<td></td>
</tr>
<tr>
<td>Page-by-Page Iterator</td>
<td></td>
</tr>
<tr>
<td>Session Facade</td>
<td>Provide a unified, workflow-oriented interface to a set of enterprise beans</td>
</tr>
<tr>
<td>Value Object</td>
<td></td>
</tr>
</tbody>
</table>

http://java.sun.com/j2ee/blueprints/design_patterns/catalog.html
Patterns: Still Evolving ...

✈ Architectural Patterns:
   ✦ Not about detailed design:
       cf. the Strategy design pattern for algorithms

✈ Anti-Patterns:
   ✦ Those that describe a bad solution to a problem which resulted in a bad situation.
   ✦ Those that describe how to get out of a bad situation and how to proceed from there to a good solution

✈ Problem Patterns:
   organizational/enterprise, business rules/logic, agent interaction, goal interaction, task, resource usage
   structural (entity, activity, constraints), behavioral, nfr

✈ OO Software Framework [GoF]:
   a set of cooperating classes that make up a reusable design for a specific class of software

✈ Pattern Mining:

✈ QWAN (Quality Without A Name) [Alexander]:
   universally recognizable aesthetic beauty and order;
   recursively nested centers of symmetry and balance
   life and wholeness; resilience, adaptability, and durability
   human comfort and satisfaction; emotional and cognitive resonance