Object Oriented Design

• Prime Benefit
  – Reusable software

• Difficulty
  – Designing classes is hard
  – Designing reusable classes is even harder

Object Oriented Design

• Designing Object Oriented System
  – Identify the participants in the system
  – Factor them into classes
    • At the right level of granularity
  – Define the public interface of each class
  – Establish inheritance hierarchy
  – Define the relationships between classes

Design Pattern

• Again
  – Designing OO SW is hard
  – Designing reusable OO SW is even harder

• Design Pattern
  – Proven solutions to recurring design problems
  – Purpose
    • To make SW reusable
    • To make SW extensible
  – Take abstraction one step further

Design Pattern

• Pattern Description
  – Independent of PLs or implementation details
    • Name and classification
    • Intent
    • Also known as
    • Motivation
    • Applicability
    • Structure
    • Participants
  – Collaboration
  – Consequences
  – Implementation
  – Sample code
  – Known as
  – Related patterns

Design Pattern

• Design patterns are
  – Solutions to recurring problems
  – Descriptions of communicating objects/classes
  – To capture the static/dynamic structure and collaboration among key participants in SW design

• Design patterns are not
  – Designs that can be encoded in classes, reused as is
    • Ex) linked lists, hash tables, etc
  – Complex domain-specific designs
    • Ex) for an entire application or subsystem
Design Pattern

- Why Use Design Pattern
  - Provide concrete, well-tested solutions
    - Help analyze the more abstract areas of a program
  - Provide clearer picture of how to implement the design
    - Help write code faster
  - Supply well-tested mechanism for reuse techniques
    - Encourage code reuse and accommodate change
  - Encourage more legible and maintainable code
  - Provide common language / jargon for programmers

- Types
  - Creational patterns
    - Deal with the process of object creation
  - Structural patterns
    - Deal with the composition of classes or objects
  - Behavioral patterns
    - Characterize the ways classes or objects interact
    - Characterize the ways they distribute responsibility

- Scope
  - Class Patterns
    - Relationship between classes and their subclasses
      - Static relationship fixed at compile time
  - Object Patterns
    - Relationship between objects
      - More dynamic: can be changed at run time

- 23 GoF Design Patterns

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Factory Method</td>
<td>Adapter(Object)</td>
<td>Chain of Responsibility</td>
</tr>
<tr>
<td>Abstract</td>
<td>Builder</td>
<td>Bridge</td>
<td>Command</td>
</tr>
<tr>
<td>Factory</td>
<td>Prototype</td>
<td>Composite</td>
<td>Iterator</td>
</tr>
<tr>
<td>Object</td>
<td>Singleton</td>
<td>Decorator</td>
<td>Mediator</td>
</tr>
<tr>
<td>Scope</td>
<td>Adapter(Class)</td>
<td>Facade</td>
<td>Observer</td>
</tr>
<tr>
<td>Class</td>
<td>Chain of Responsibility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scope</td>
<td>Command</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Interpreter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scope</td>
<td>Template Method</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How to become a Game Master

- First, learn the rules
  - Weapons, treasures, enemies, etc
- Then, learn the principles
  - Strategic value of certain treasures,
  - Power of weapons to different enemies,
  - etc
- Study the games of other masters
  - Masters know the patterns for the game
  - Understand, memorize, and apply them repeatedly

How to be a SW Design Master

- First, learn the rules
  - Algorithms, data structures, languages of SW
- Then, learn the principles
  - Structural, modular, OO, generic, etc
- Study the designs of other masters
  - Masters know the patterns for the design
  - Understand, memorize, and apply them repeatedly
How to become a Game Addict

• You cannot appreciate the game at first
• As you play them, you learn the differences between different types of weapons, enemies, or treasures
• As you become a connoisseur, you experience the various hidden surprises you didn’t notice before

How to become a Design Addict

Warning
Once you are hooked, you will no longer be satisfied with plonk!

Design Patterns for Today

• Singleton
  – Creational object pattern
• Facade
  – Structural object pattern
• Mediator
  – Behavioral object pattern

Singleton

• Intent
  – Ensure that a class has only one instance
  – Provide a global point of access to the instance
• Motivation
  – Singularity: To ensure only one instance exists
    • Only one printer spooler
    • Only one window manager
    • Only one audio output

Singleton

• Applicability
  – The class must have exactly one instance
  – The instance must be accessible from a well-known access point
  – When extending the sole instance by subclassing, client codes should not have to be modified

Singleton

• Structure

```
Singleton
static Instance()
SingletonOperation()
GetSingletonData()

static uniqueInstance
singletonData
```

return uniqueInstance
Singleton

• Participants
  – Singleton
    • Create the unique instance
    • Provide an access point to the instance
      – Defines operation `Instance()`
  • Collaboration
    – Clients access the instance solely through Singleton’s `Instance()` operation

• Consequences
  – Controlled access
  – Permits subclassing
  – Enhances flexibility
  – Reduces name space
  – Permits a variable number of instances

Singleton

• Implementation

Singleton

• Known uses
  • Related patterns
    – Abstract factory
    – Builder
    – Prototype

Facade

• Intent
  – Provide a unified interface to a set of interfaces in a subsystem
  – Make a single object represent an entire system

• Motivation
  – Structure a system into subsystems
    • Minimizes the communication and dependencies
  – Uses Facade as an entry point to each subsystem.
  – Example
    • Compiler subsystem in IDE

Facade

• Motivation

Facade
Facade

- **Applicability**
  - Want to provide a simple interface to a complex subsystem
  - Want to decouple a subsystem from clients and other systems
  - Want to layer your subsystems

- **Structure**

- **Participants**
  - Facade
    - Delegates client requests to subsystem objects
  - Subsystem classes
    - Implement subsystem functionality
    - Handle work assigned by the Facade
    - No knowledge about Facade
  - Collaboration
    - Clients, Facade, Subsystems
      - Clients sending requests to Facade
      - Facade delegating requests to subsystem objects

- **Consequences**
  - Provides a higher level interface
    - Shields clients from subsystem components
  - Promotes weak coupling
    - Helps layer a system
    - Eliminates complex or circular dependencies
    - Reduces compilation dependencies
  - Provides choice
    - Ease-of-use vs Generality

Mediator

- **Intent**
  - Define an object that encapsulates how a set of objects interact.
  - Promote loose coupling
    - Keep objects from referring to each other explicitly
    - Let you vary their interaction independently

- **Motivation**
  - The "know-all" class
    - Decides how the objects interact
    - Enables object reusability
  - Different system behavior by replacing mediator
Mediator

• Applicability
  – A set of objects communicate in well-defined but complex way
    • Unstructured interdependency
    • Difficult to understand
  – Reusing an object is difficult because it refers to and communicates with many other objects
  – A behavior that is distributed between several classes should be customizable

Mediator

• Structure

Mediator

• Participants
  – Mediator
  – Concrete mediator
  – Colleague classes
• Collaboration
  – Colleagues send/receive requests from a mediator
  – Mediator implements the cooperative behavior by routing request between the appropriate colleagues

Mediator

• Consequences
  – Centralized control
  – Decoupled colleagues
  – Simplified object protocols
  – Limited subclassing
  – Abstracts how objects cooperate

Mediator

• Implementation