Design Patterns

...live and in action!

Arranged by Krysta Yousoufian for CSE 331, 3/8/2012
With material from Hal Perkins, David Notkin, Michael Ernst, Marty Stepp, and Joshua Bloch (Effective Java)
MVC

- One of the most well-known patterns
- Review it before interviews (especially at web companies)
More Common Patterns

- Recall from lecture...
- Creational
  - Create objects without calling constructor directly
  - Singleton: allow only one instance
  - Factory: hide constructors
  - Prototype: “cloneable” objects
- Structural (wrappers)
  - Interact with the “important” class through a wrapper class
  - Adapter: different interface, same functionality
  - Decorator: same interface, different functionality
  - Proxy: same interface, same functionality
- Behavioral
  - Interface for communication between objects
  - Visitor: traverse a data structure
Singleton

• One shared instance of a class

• When useful
  o Maintaining global state; coordinating among objects or threads
  o Often lower-level tasks (e.g. hardware interaction)

• When not useful
  o Need to store state/data specific to each use (instance fields)

• Controversial
  o Global \(\rightarrow\) hides dependencies, hard to test
  o Overused
  o Good tool to have, but only use if it’s the right tool (get a second opinion!)

• Examples: logger, window manager
Implementing Singleton

- Private constructor
- Several options (*Effective Java* pp. 18+)
  - One private static instance, accessed with getInstance()
    - Pros: flexibility – could reimplement getInstance() to no longer be Singleton
  - One publicly accessible static instance
    - Pros: clarity – obvious that you’re using a shared copy
  - Enum
    - Pros: safer (harder to break Singleton), provides serialization
    - But not how Enum is meant to be used
Singleton Demo

FileServer / Logger
Factory

• Get new object by calling non-constructor (getInstance(), valueOf(), …)
  o May create a new object or may reuse an old one

• Advantages (Effective Java, pg. 5)
  o Can reuse objects
  o Can return objects of subclasses
  o More descriptive naming than constructors
Factory Demo

GameFactory / GameRoom
Adapter

- Different interface, same functionality
- Use: translate interface to be compatible with a different object
Demo: TicTacToe / GameRoom
Strategy

- **Problem**: We want to generalize behavior of one part of our app.
  - Example: Layout of components within containers.
  - Example: Ways of sorting to arrange data.
  - Example: Computer game player AI algorithms.
Poor Solutions

• Boolean flags or many set methods to enable various algorithms.
  myContainer.useFlow(); game.playerDifficulty(3);

• Lots of if statements in our app to choose between algorithms.
  if (abc) { mergeSort(data); } else if (xyz) { bubbleSort(data); }

• Rewriting entire model classes just to change the algorithm.
  FlowContainer, BorderContainer, ..., EasyPlayer, HardPlayer
Strategy Pattern

- **strategy**: An algorithm separated from the object that uses it, and encapsulated as its own object.
  - Each strategy implements one specific behavior; one implementation of how to solve the same problem.
  - Separates algorithm for behavior from object that wants to act.
  - Allows changing an object's behavior dynamically without extending or changing the object itself.
Strategy Demo

- RockPaperScissors / GameRoom