# Design Patterns

15-413: Introduction to Software Engineering

Jonathan Aldrich

## Design 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



## History

- Christopher Alexander, *The Timeless Way of Building* (and other books
  - Proposes patterns as a way of capturing design knowledge in architecture
  - Each pattern represents a tried-and-true solution to a design problem
  - Typically an engineering compromise that resolves conflicting forces in an advantageous way



## Patterns in Physical Architecture

- When a room has a window with a view, the window becomes a focal point: people are attracted to the window and want to look through it. The furniture in the room creates a second focal point: everyone is attracted toward whatever point the furniture aims them at (usually the center of the room or a TV). This makes people feel uncomfortable. They want to look out the window, and toward the other focus at the same time. If you rearrange the furniture, so that its focal point becomes the window, then everyone will suddenly notice that the room is much more "comfortable".
  – Leonard Budney, Amazon.com review of *The Timeless Way of Building*



## Benefits of Patterns

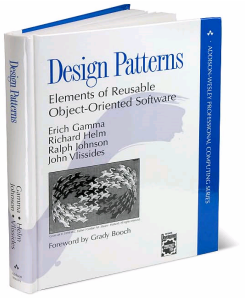
- Shared language of design
  - Increases communication bandwidth
  - Decreases misunderstandings
- Learn from experience
  - Becoming a good designer is hard
    - Understanding good designs is a first step
  - Tested solutions to common problems
    - Where is the solution applicable?
    - What are the tradeoffs?



## Elements of a Pattern

- Name
  - Important because it becomes part of a design vocabulary
  - Raises level of communication
- Problem
  - When the pattern is applicable
- Solution
  - Design elements and their relationships
  - Abstract: must be specialized
- Consequences
  - Tradeoffs of applying the pattern
    - Each pattern has costs as well as benefits
    - Issues include flexibility, extensibility, etc.
    - There may be variations in the pattern with different consequences

## Worth Buying (and no, I don't get a kickback ;-)




- Brought Design Patterns into the mainstream
- Authors known as the Gang of Four (GoF)
- Focuses on *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*



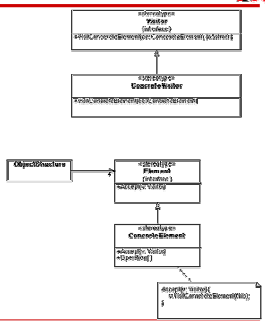
## Let's look at some patterns

- Creational Patterns
- Structural Patterns
- Behavioral Patterns



## Behavioral: Visitor

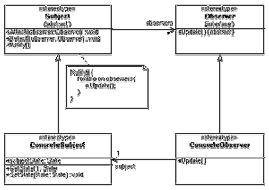
- Applicability
  - Structure with many classes
  - Want to perform operations that depend on classes
  - Set of classes is stable
  - Want to define new operations
- Consequences
  - Easy to add new operations
  - Groups related behavior in Visitor
  - Adding new elements is hard
  - Visitor can store state
  - Elements must expose interface





## Behavioral: Observer

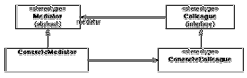
- Applicability
  - When an abstraction has two aspects, one dependent on the other, and you want to reuse each
  - When change to one object requires changing others, and you don't know how many objects need to be changed
  - When an object should be able to notify others without knowing who they are
- Consequences
  - Loose coupling between subject and observer, enhancing reuse
  - Support for broadcast communication
  - Notification can lead to further updates, causing a cascade effect





## Behavioral: Mediator

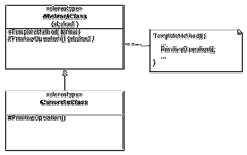
- Applicability
  - A set of objects that communicate in well-defined but complex ways
  - Reusing an object is difficult because it communicates with others
  - A behavior distributed between several classes should be customizable without a lot of subclassing
- Consequences
  - Avoids excessive subclassing to customize behavior
  - Decouples colleagues, enhancing reuse
  - Simplifies object protocols: many-to-many to one-to-many
  - Abstracts how objects cooperate into the mediator
  - Centralizes control
    - Danger of mediator monolith





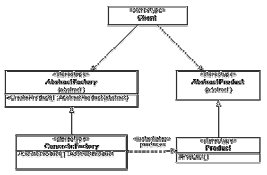
## Behavioral: Template Method

- Applicability
  - When an algorithm consists of varying and invariant parts that must be customized
  - When common behavior in subclasses should be factored and localized to avoid code duplication
  - To control subclass extensions to specific operations
- Consequences
  - Code reuse
  - Inverted "Hollywood" control: don't call us, we'll call you
  - Ensures the invariant parts of the algorithm are not changed by subclasses



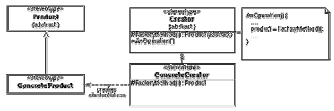
## Creational: Abstract factory

- Applicability
  - System should be independent of product creation
  - Want to configure with multiple families of products
  - Want to ensure that a product family is used together
- Consequences
  - Isolates concrete classes
  - Makes it easy to change product families
  - Helps ensure consistent use of family
  - Hard to support new kinds of products





## Creational: Factory Method

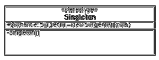


- Applicability
  - A class can't anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates
- Consequences
  - Provides hooks for subclasses to customize creation behavior
  - Connects parallel class hierarchies



## Creational: Singleton

- Applicability
  - There must be exactly one instance of a class
  - When it must be accessible to clients from a well-known place
  - When the sole instance should be extensible by subclassing, with unmodified clients using the subclass
- Consequences
  - Controlled access to sole instance
  - Reduced name space (vs. global variables)
  - Can be refined in subclass or changed to allow multiple instances
  - More flexible than class operations
    - Can change later if you need to



- Implementation
  - Constructor is protected
  - Instance variable is private
  - Public operation returns singleton
    - May lazily create singleton
- Subclassing
  - Instance() method can look up subclass to create in environment



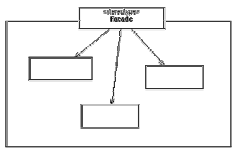
## Announcements

- Iteration 2 plan due today
  - Next homework out this afternoon; due next Friday
- Course workload
  - 6-7 hours: 6 students
  - 8-9 hours: 10 students
  - 12-14 hours: 2 students



## Structural: Facade

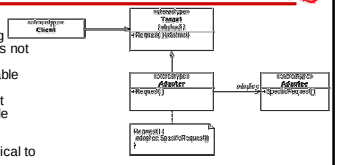
- Applicability
  - You want to provide a simple interface to a complex subsystem
  - You want to decouple clients from the implementation of a subsystem
  - You want to layer your subsystems
- Consequences
  - It shields clients from the complexity of the subsystem, making it easier to use
  - Decouples the subsystem and its clients, making each easier to change
  - Clients that need to can still access subsystem classes





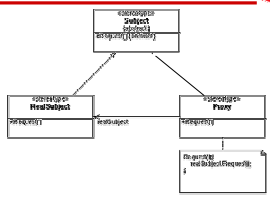
## Structural: Adapter

- Applicability
  - You want to use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated classes that don't necessarily have compatible interfaces
  - You need to use several subclasses, but it's impractical to adapt their interface by subclassing each one
- Consequences
  - Exposes the functionality of an object in another form
  - Unifies the interfaces of multiple incompatible adaptee objects
  - Lets a single adapter work with multiple adaptees in a hierarchy



## Structural: Proxy

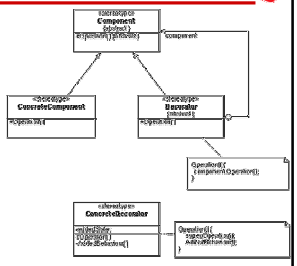
- Applicability
  - Whenever you need a more sophisticated object reference than a simple pointer
    - Local representative for a remote object
    - Create or load expensive object on demand
    - Control access to an object
    - Reference count an object
- Consequences
  - Introduces a level of indirection
    - Hides distribution from client
    - Hides optimizations from client
    - Adds housekeeping tasks





## Structural: Decorator

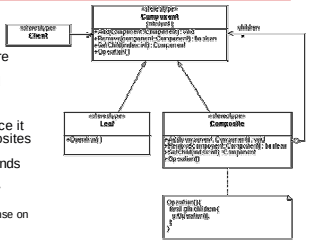
- Applicability
  - To add responsibilities to individual objects dynamically and transparently
  - For responsibilities that can be withdrawn
  - When extension by subclassing is impractical
- Consequences
  - More flexible than static inheritance
  - Avoids monolithic classes
  - Breaks object identity
  - Lots of little objects





## Structural: Composite

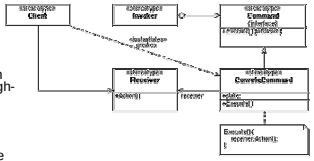
- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components





## Behavioral: Command

- Applicability
  - Parameterize objects by an action to perform
  - Specify, queue and execute requests at different times
  - Support undo
  - Support logging changes that can be reapplied after a crash
  - Structure a system around high-level operations built out of primitives
- Consequences
  - Decouples the object that invokes the operation from the one that performs it
  - Since commands are objects they can be explicitly manipulated
  - Can group commands into composite commands
  - Easy to add new commands without changing existing code





## Other GoF Patterns

- Creational
  - Builder – separate creation from representation
  - Prototype – create objects by copying
- Structural
  - Bridge – decouple abstraction from implementation
  - Flyweight – use sharing for fine-grained objects
- Behavioral
  - Chain of Responsibility – sequence of objects can respond to a request
  - Interpreter – canonical implementation technique
  - Memento – externalize/restore an object's state
  - State – allow object to alter behavior when state changes
  - Strategy – encapsulate algorithm as object