## Design Patterns

- Designing good and reusable OO software is hard.
  - Mix of specific + general
  - Impossible to get it right the first time
- Experienced designers will use solutions that have worked for them in the past.
- Design patterns
  - Systematically
    - names,
    - explains,
    - and evaluates

    important, recurring designs in OO systems.



## Genesis

- Christopher Alexander, *et. al.*
  - *A Pattern Language*
  - Oxford University Press, 1977

  *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of a 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."*

  - Talking about buildings, bridges and towns.
- During the last decade, a "pattern community" has developed in the field of software design.



## Design Patterns in General

- Pattern name
  - A word or two that increases our design vocabulary
- Problem
  - Describes when to apply the pattern.
- Solution
  - Describes the elements that make up the design:
    - Responsibilities, relationships, collaborations
    - A general arrangement of classes
      - Must be adapted for each use
- Consequences
  - Results and trade-offs of applying the pattern
    - Space & time
    - Implementation issues
    - Impact on flexibility, extensibility, portability



## Design Patterns Specifically

- **Pattern name and classification**
- **Intent**
  - What does it do? What's its rationale
- **Also knows as**
- **Motivation**
  - A use scenario
- **Applicability**
  - In what situations can you apply it? How can you recognize these situations.
- **Structure**
  - UML
- **Participants**
- **Collaborations**
- **Consequences**
  - Trade-offs in applying this pattern
- **Implementation**
  - Any implementation tips when applying the pattern
- **Sample code**
- **Known uses**
- **Related patterns**



## Design Pattern Coverage

- In this course, we will cover a limited number of very basic design patterns.
- This is only a fraction of what a real expert might know.



## Design Pattern Space

| | | Purpose | | | | |
|---|---|---|---|---|---|---|
| | | Creational | Structural | Behavioral | Storage | Distributed |
| Scope | Class | Factory method | Adapter<br>Template Base | Interpreter<br>Template Method | Object File<br>RDB Direct | |
| | Object | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter<br>Bridge<br>Composite<br>Decorator<br>Façade<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Flyweight<br>Observer<br>State<br>Strategy<br>Visitor | OODB Proxy | Attribute Factory |

## Scope

- Class
  - Relationships between classes and their subclasses
  - No need to execute any code to set them up
  - Static, fixed at compile-time
- Object
  - Relies on object pointers.
  - Can be changed at run-time, are more dynamic.



## Purpose

- Creational
  - Concerns the process of object creation
- Structural
  - Concerns the relationships between classes and objects
- Behavioral
  - Concerns the ways objects and classes distribute responsibility for performing some task.
- Storage
  - Concerns the ways objects can be made persistent.
- Distributed
  - Concerns the ways server objects are represented on a client.



## Creational Patterns

Class
- Factory Method
  - Define an interface for creating an object, but let subclasses decide which class to instantiate.

Object
- Abstract Factory
  - Provide an interface for creating families of related objects without specifying their concrete classes.
- Builder
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Prototype
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Singleton
  - Ensure a class only has one instance, and provide a global point of access to it.



## Structural Patterns

Class
- Adapter
  - Convert the interface of a class into another interface clients expect.
- Template Base
  - Use templated base classes to specify associations.

Object
- Adapter
  - Convert the interface of a class into another interface clients expect.
- Bridge
  - Decouple an abstraction from its implementation so that the two can vary independently (run-time inheritance)
- Composite
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



## Structural Patterns (cont'd)

- Object (cont'd)
  - Decorator
    - Attach additional responsibilities to an object dynamically.
  - Façade
    - Provide a unified interface to a set of interfaces in a subsystem.
  - Flyweight
    - Use sharing to support large numbers of fine-grained objects efficiently.
  - Proxy
    - Provide a surrogate or placeholder for another object to control access to it.



## Behavioral Patterns

- Class
  - Interpreter
    - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
  - Template Method
    - Let subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Object
  - Chain of Responsibility
    - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
  - Command
    - Encapsulate a request as an object.
  - Iterator
    - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
  - Mediator
    - Define an object that encapsulates how a set of objects interact.

## Behavioral Patterns (cont'd)

Object (cont'd)
- Memento
  - Capture and externalize an object's internal state so that the object can be restored to this state later.
- Observer
  - When one object changes state, all its dependents are notified and updated automatically.
- State
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Strategy
  - Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Visitor
  - Represent an operation to be performed on the elements of an object structure.



## Storage Patterns

Class
- Object File
  - Store and retrieve a network of objects to a sequential file.
- RDB Direct
  - Store and retrieve a network of objects to a relational database.

Object
- OODB Proxy
  - Store and retrieve objects from an object-oriented database.

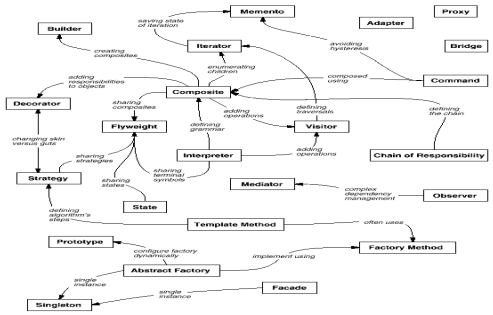


## Distributed Patterns

Object
- Attribute Factory
  - Generate a lightweight object graph on the client-side of a client-server system.



## Relationships Between Patterns






## Finding Appropriate Objects

- Hard part about OOD is decomposing a system into objects.
- Many objects come directly from the analysis model or from the implementation space.
- OO designs often wind up with classes that have no such counterparts.
  - E.g., Composite, Strategy, Sate



## Determining Object Granularity

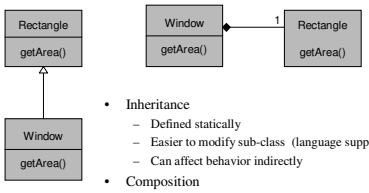
- Too large
  - Hard to change.
  - Procedural program inside an object.
  - Large, shared data structure.
  - Hard to understand
- Too small
  - Inefficiencies
    - Copied data
    - Method invocation overhead
  - Hard to understand
- Whatever the choice, negative consequences can be mitigated by judicious use of certain patterns:
  - Flyweight, Façade, Builder, Visitor, Command, …

## Using Object Interfaces

- Never refer to a class by name. Always use interfaces.
  - Callers remain unaware of the specific types they use.
    - can extend the type structure
  - Callers remain unaware of the classes that implement the interfaces.
    - can dynamically load new implementations
- Sometimes difficult to put into practice.
  - Creational patterns help a great deal.



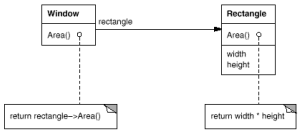
## Inheritance v.s. Composition




- Inheritance
  - Defined statically
  - Easier to modify sub-class (language supported)
  - Can affect behavior indirectly
- Composition
  - Can change implementations at run-time
  - Does not break encapsulation
  - Less "uselessly" general



## Delegation




- Can implement inheritance using delegation.
- Makes it easier to compose behaviours at run-time (e.g., Window can become circular at run-time)
- Many design patterns rely on delegation.