Object-Oriented Programming
Design patterns
Composite Pattern

CSIE Department, NTUT
Woei-Kae Chen
Catalog of Design patterns

- Creational patterns
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton

- Structural patterns (composition)
  - Adaptor, Bridge, Composite, Decorator, Facade, Flyweight, Proxy

- Behavioral patterns (interaction)
  - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor
### Design patterns space

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

*Relationship*
Creational patterns

- **Abstract Factory**
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

- **Builder**
  - Separate the construction of a complex object from its representation so that same construction process can create different representations.

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

- **Singleton**
  - Ensure a class only has one instance, and provide a global point of access to it.
Structural patterns (1)

- **Adaptor**
  - 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.

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

- **Decorator**
  - Attach additional responsibilities to an object dynamically.
Structural patterns (2)

● Facade
  – Provide a unified interface to a set of interfaces in subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

● 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 (1)

- 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, thereby letting you parameterize clients with different requests.

- Interpreter
  - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

- Iterator
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Behavioral patterns (2)

- **Mediator**
  - Define an object that encapsulates how a set of objects interact.

- **Memento**
  - Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.

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

- **State**
  - Allow an object to alter its behavior when its internal state changes. The object all appear to change its class.
Behavioral patterns (3)

- **Strategy**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable.

- **Template Method**
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclass. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

- **Visitor**
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
Composite pattern
Composite pattern: Intent

- Compose objects into a tree structure to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
We want to build a complex component out of some simple components.
Composite pattern: Motivation

A typical Composite object structure

```
  aComposite
  ├── aLeaf
  │    ├── aLeaf
  │    │    ├── aLeaf
  │    │    │    └── aLeaf
  │    └── aLeaf
  └── aLeaf
```
Composite pattern: Applicability

- You want to represent part-whole hierarchies of objects.
- You want clients to be able to ignore the difference between composite structure uniformly.
Composite pattern: Structure

Component

Operation()
Add(Component)
Remove(Component)
GetChild(int)

Leaf
Operation()

Composite

Operation()
Add(Component)
Remove(Component)
GetChild(int)

for all g in children
  g.Operation() ;
Composite pattern: Participants

- **Component**
  - Declare interface (leaf, composite, parent)
  - Implements default behavior
- **Leaf**
  - Leaf objects (no children)
  - Defines behavior for primitive objects
- **Composite**
  - Defines behavior for components having children
  - store child components
  - Implement child-related operations
- **Client**
  - manipulate objects with component interface
Composite pattern: Collaboration

- Clients use the Component class interface to interact with objects in the composite structure.
  - If the recipient is a leaf, then the request is handled directly.
  - If the recipient is a composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.
Composite pattern: Consequence

- Defines class hierarchies consisting of primitive objects and composite objects. (Recursive composition)
- Makes the client simple. Clients usually do not care whether they are dealing with a leaf or a composite component.
- Makes it easier to add new kinds of components.
- Can make your design overly general. You can’t rely on the type system to enforce constraints for components of a composite.
Composite pattern: Implementation (1)

- Explicit parent references
- Sharing components
  - when a component have only one parent
- Maximizing the Component interface
  - conflict with the principle that a class (component) should only defines operations that are meaningful to its subclass.
- Declaring the child management operations.
  - trade-off between safety and transparency
- Should Component implement a list of Components?
  - only if there are relatively few children in the structure
Composite pattern: Implementation (2)

- Child ordering
  - example: front-to-back ordering for Graphics

- Caching to improve performance
  - Composite class can cache traversal or search information
  - best for components to know their parents

- Who should delete component?
  - make a Composite responsible for deleting its children

- What’s the best data structure for storing component?
  - efficiency (array, vector, list, hash table)