Design Patterns

Sommerville, Chapter 18

Instructor: Peter Baumann

email: pbaumann@constructor.university

tel: -3178

office: room 88, Research 1

Credits:
Xiaochuan Yi, U of Georgia
Nenad Medvidović
dofactory.com

CONGRESS.SYS Corrupted:
Re-boot Washington D.C. (Y/n)?

Introduction to Design Patterns

- Be a good programmer
  - …and efficient – *learn from others!*
- Similar patterns occur over and over
  - Not reinventing the wheel
  - Sharing knowledge of problem solving
  - communication between programmers
  - Write elegant and graceful code
- Computer programming as art [Donald Knuth]
  - Recognize conceptual beauty

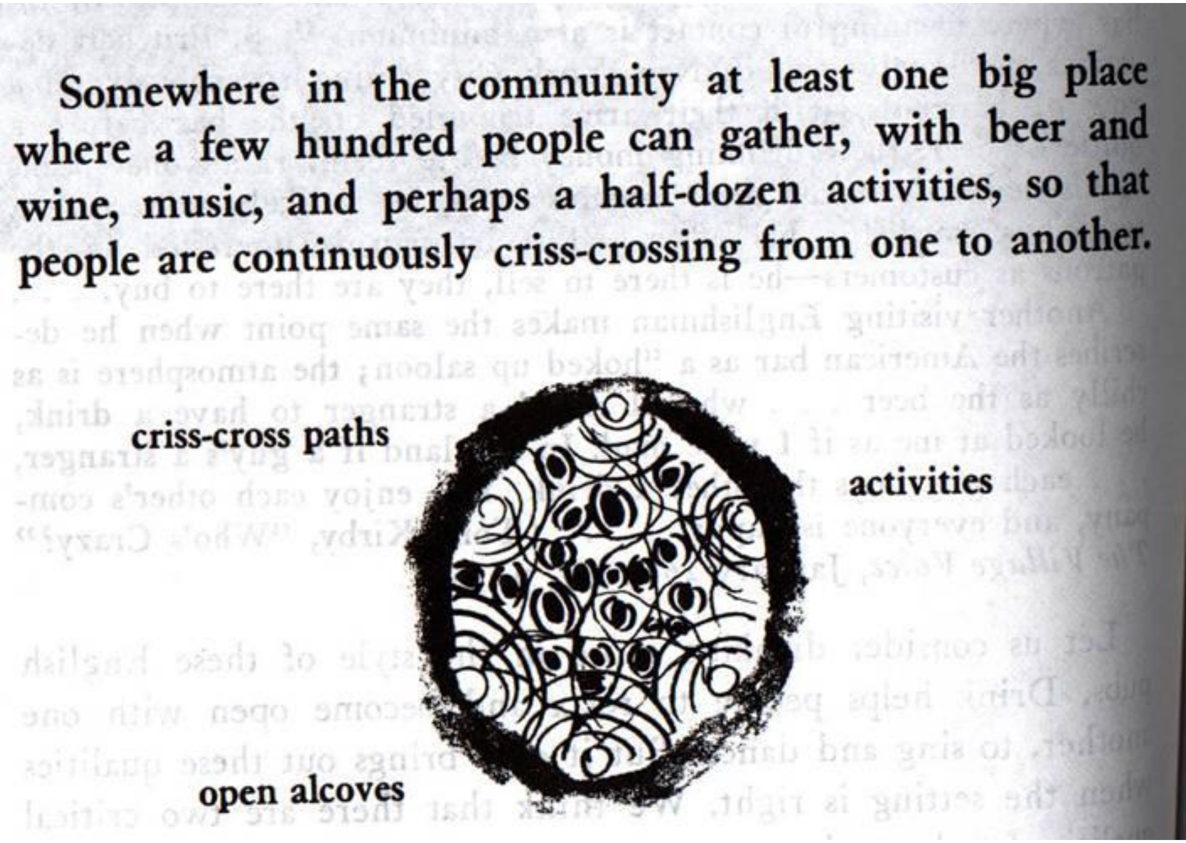
Design Patterns

- pattern =
  description of the problem and the essence of its solution
  - should be sufficiently abstract to be reused in different settings
  - often rely on object characteristics such as inheritance and polymorphism
- design pattern =
  way of re-using abstract knowledge about a (sw) design problem and its solution

C›ONSTRUCTOR
UNIVERSITY

History of Design Patterns

- First used in architecture
  - Christopher Alexander, 1977
  - Ex. How to create a beer hall where people socialize?



Somewhere in the community at least one big place where a few hundred people can gather, with beer and wine, music, and perhaps a half-dozen activities, so that people are continuously criss-crossing from one to another.

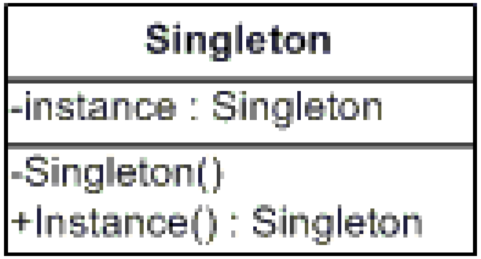
criss-cross paths
activities
open alcoves

- Design Patterns: Elements of Reusable Object-Oriented Software (1995)
  - “Gang of four”: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

A Pattern Template

- Name
  - meaningful identifier
- Description
  - What’s the essence?
- Problem / applicability description
  - When advantageous to use?
- Solution description
  - Not concrete design, but template → can be instantiated in different ways
- Consequences
  - results & trade-offs

Patterns by Example: Singleton



Singleton

-instance : Singleton

-Singleton()

+Instance() : Singleton

- Name
  - Singleton
- Description
  - Ensure a class has only one instance and provide a global point of access to it
- Problem / Applicability
  - Used when only one object of a kind may exist in the system
- Solution
  - defines an Instance operation that lets clients access its unique instance
  - Instance is a class operation
  - responsible for creating and maintaining its own unique instance

Singleton Code

```
// Singleton pattern -- Structural example

class Singleton
{
public:
    static Singleton* Instance()
    {
        static Singleton instance;
        return &instance;
    }
private:
  Singleton() {}
}
```

```
int main()
{
    // Constructor is protected, cannot use new
    Singleton *s1 = Singleton::Instance();
    Singleton *s2 = Singleton::Instance();
    Singleton *s3 = s1->Instance();
    Singleton &s4 = *Singleton::Instance();

    if( s1 == s2 )
        cout << "same instance"  << endl;
}
```

Singleton Application

```
class LoadBalancer
{
private:
    LoadBalancer()
    {
        add_all_servers;
    }
public:
    static LoadBalancer *GetLoadBalancer()
    {
        // thread-safe in C++ 11
        static LoadBalancer balancer;
        return &balancer;
    }
…
}
```

```
// SingletonApp test

LoadBalancer *b1 = LoadBalancer::GetLoadBalancer();
LoadBalancer *b2 = LoadBalancer::GetLoadBalancer();

if( b1 == b2 )
    cout << "same instance"  << endl;
```

For the experts:
In JavaScript, use closure

Singleton, Revisited

Problems:

- Subclassing
- Copy constructor
- Destructor: when?
- Static vs. heap

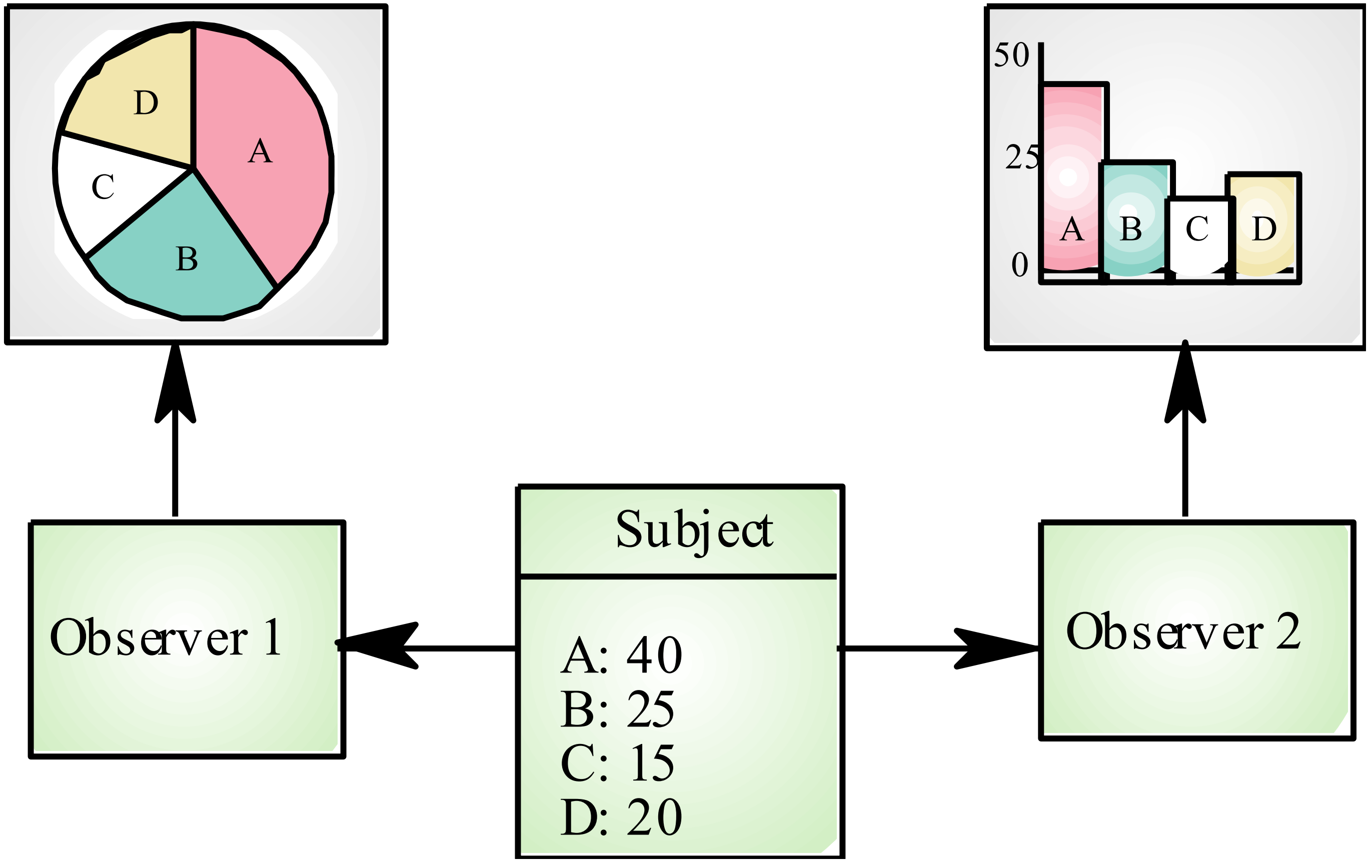
```
// Singleton pattern

class Singleton
{
public:
    static Singleton* Instance()
    {
        static Singleton instance;
        return &instance;
    }
private:
  Singleton() {}
}
```

```
// Singleton -- modified example

class Singleton
{
public:
    static Singleton* Instance()
    {
        static Singleton instance;
        return &instance;
    }
private:
    Singleton() {}
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
}
```

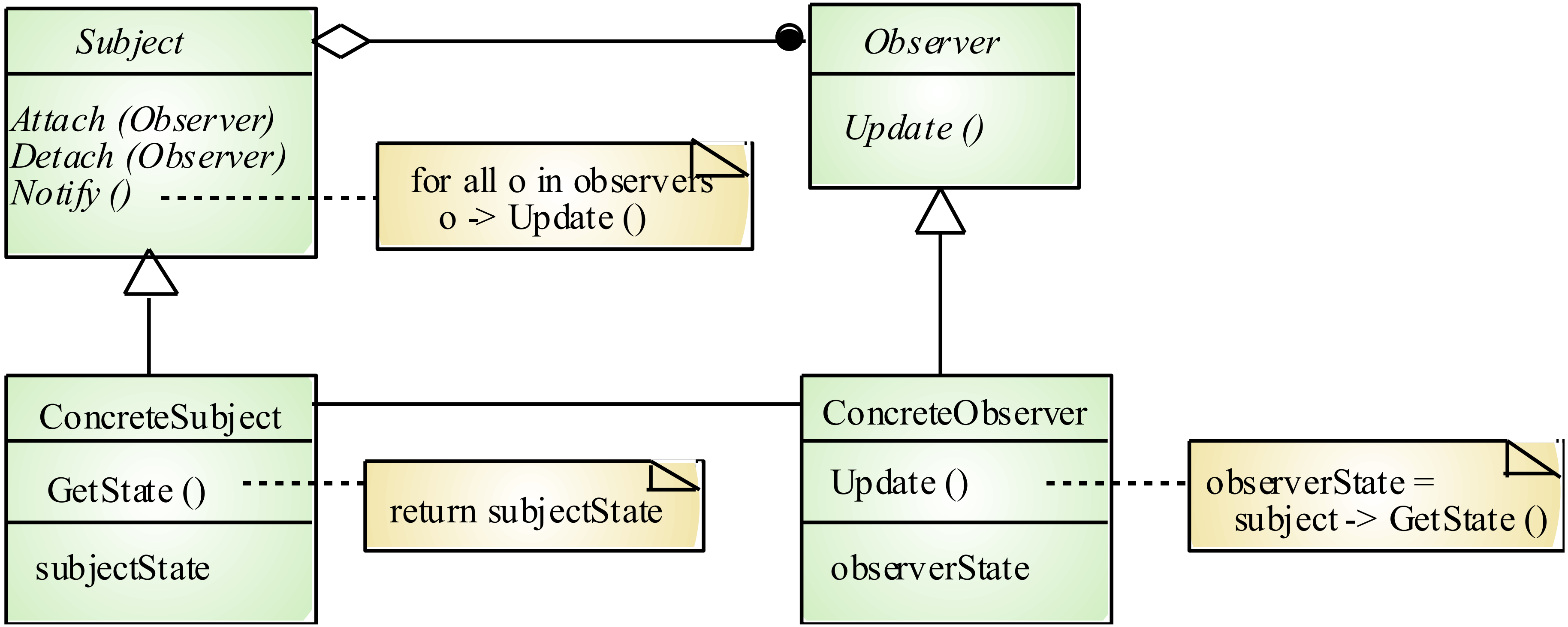
Multiple displays enabled by Observer



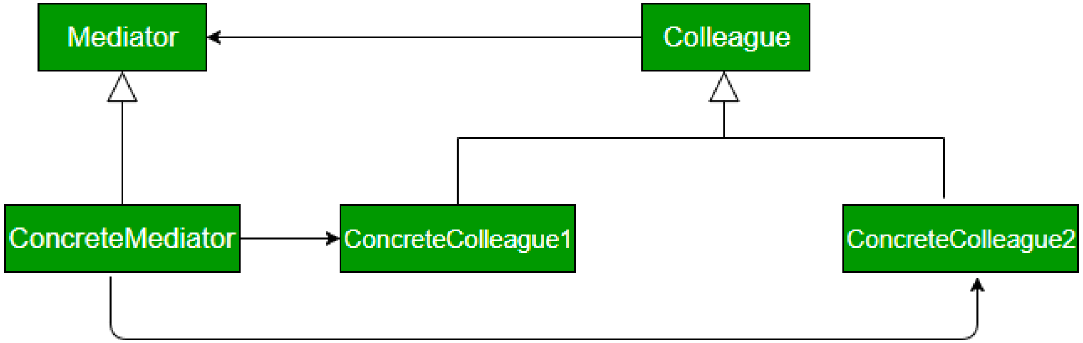
The Observer Pattern

- Name
  - Observer
- Description
  - Separates the display of object state from the object itself
- Problem / Applicability
  - Used when multiple displays of state are needed
- Solution
  - See slide with UML description
- Consequences
  - Optimizations to enhance display performance are impractical

The Observer Pattern

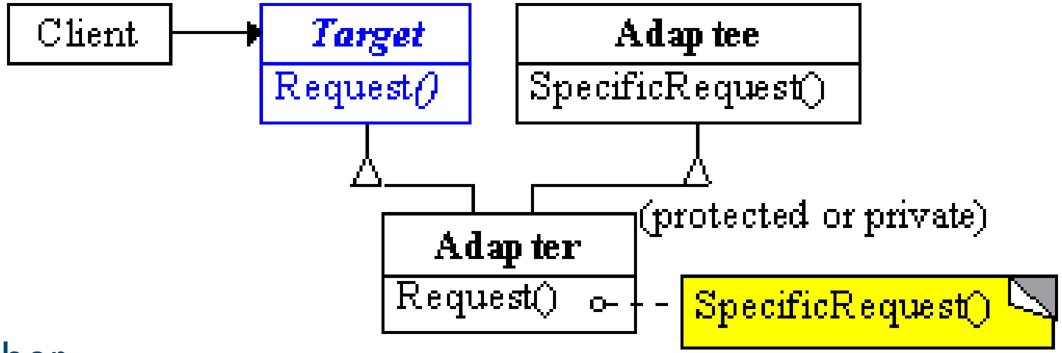


The Mediator Pattern



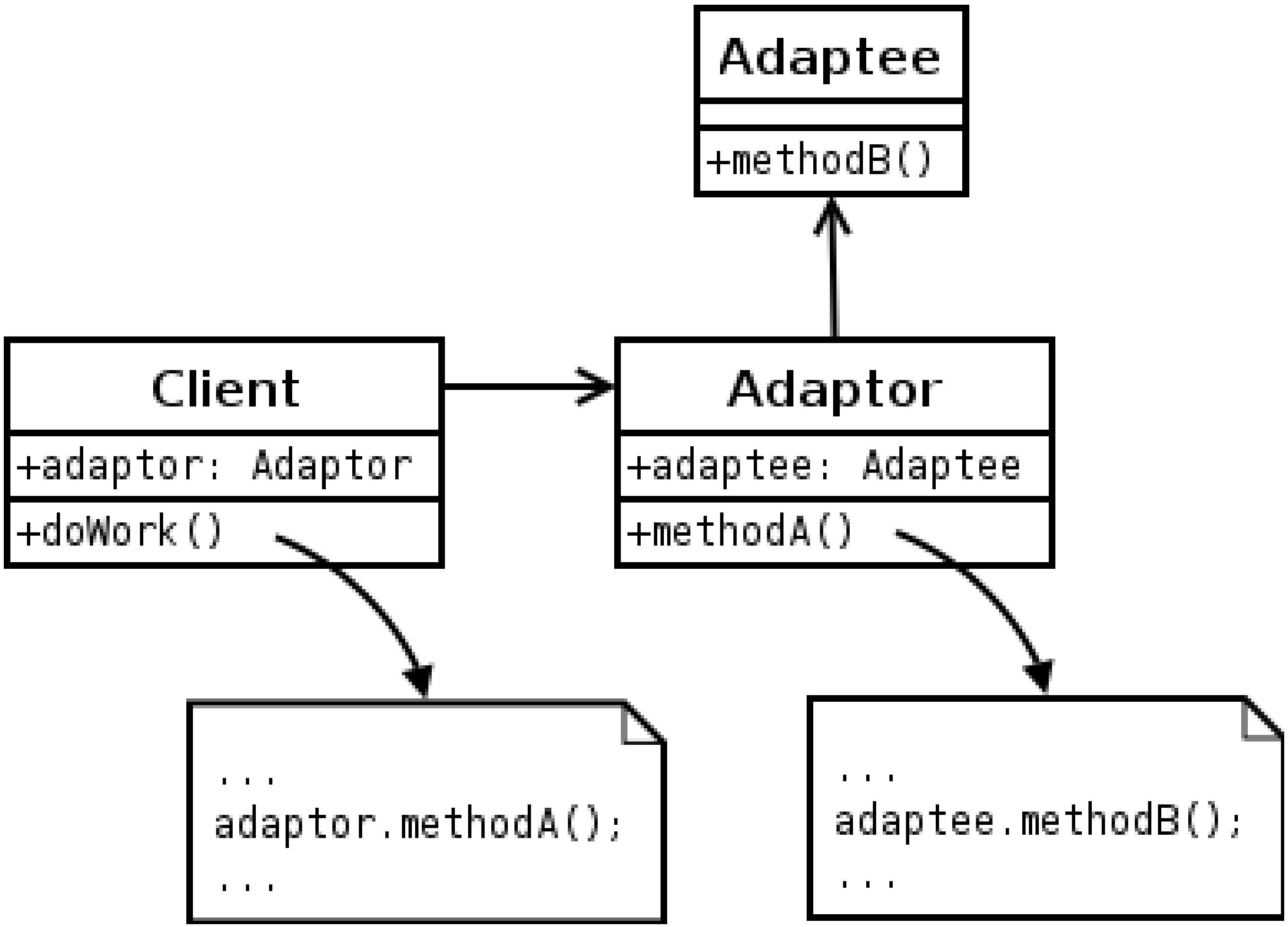
- Description
  - Define an object that encapsulates how a set of objects interact
  - Mediator promotes loose coupling by keeping objects from referring to each other explicitly
- Problem / Applicability
  - Complex interaction exists
- Consequences
  - Limits subclassing; Decouples colleagues; Simplifies object protocols; Abstracts how objects cooperate; Centralizes control

The Adapter Pattern



- Description
  - Adapter lets classes work together
    that could not otherwise because of incompatible interfaces
- Problem / Applicability
  - Need to use an existing class whose interface does not match
  - Need to make use of incompatible classes
- Consequences
  - Class adapter commits to the concrete Adapter class

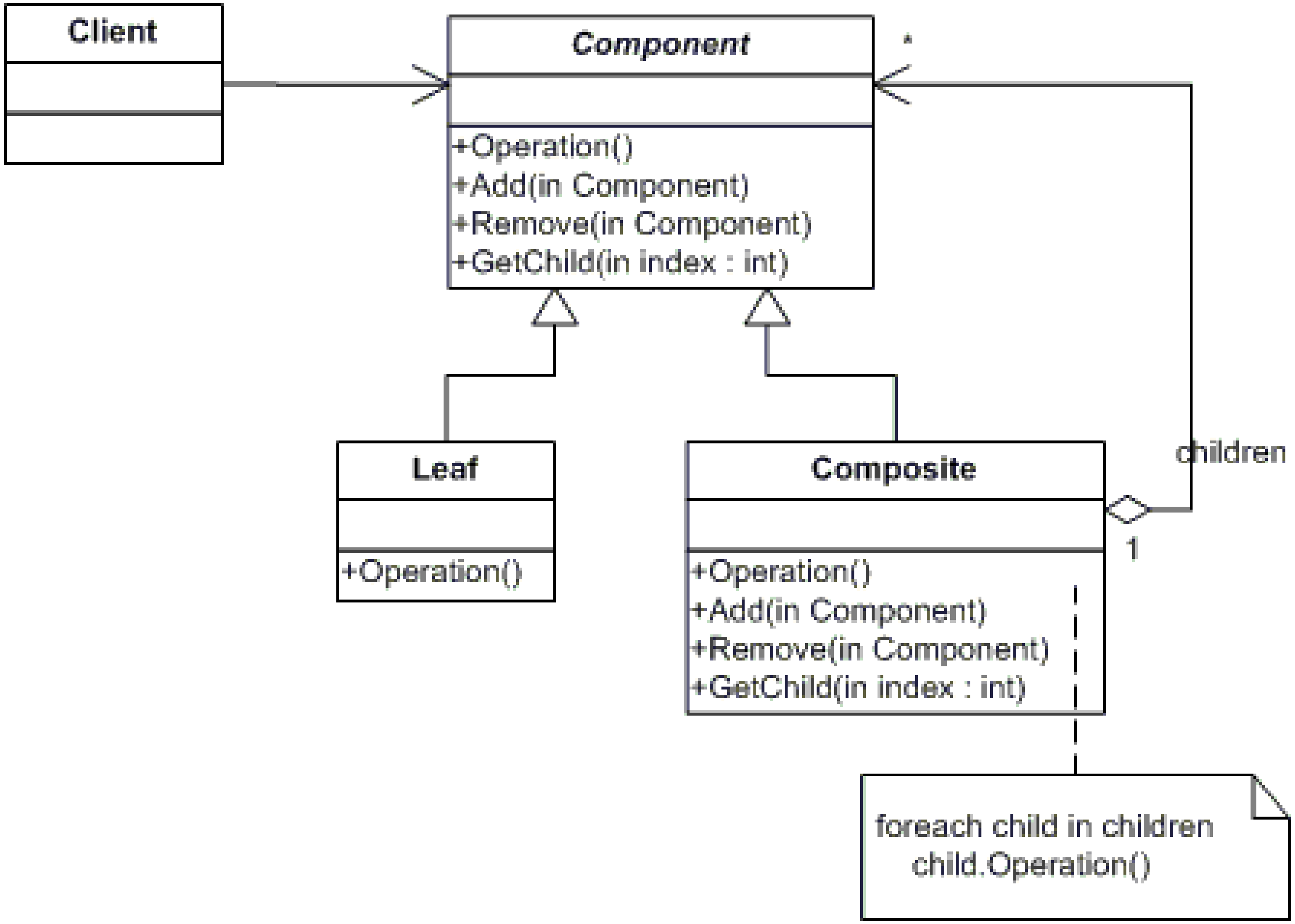
Adapter: Another View [Wikipedia]



Composite Pattern

- Definition
  - Compose objects into tree structures to represent part-whole hierarchies
  - Composite lets clients treat individual objects and compositions of objects uniformly
- Problem / Applicability
  - Any time there is partial overlap in the capabilities of objects

Composite Pattern UML Diagram



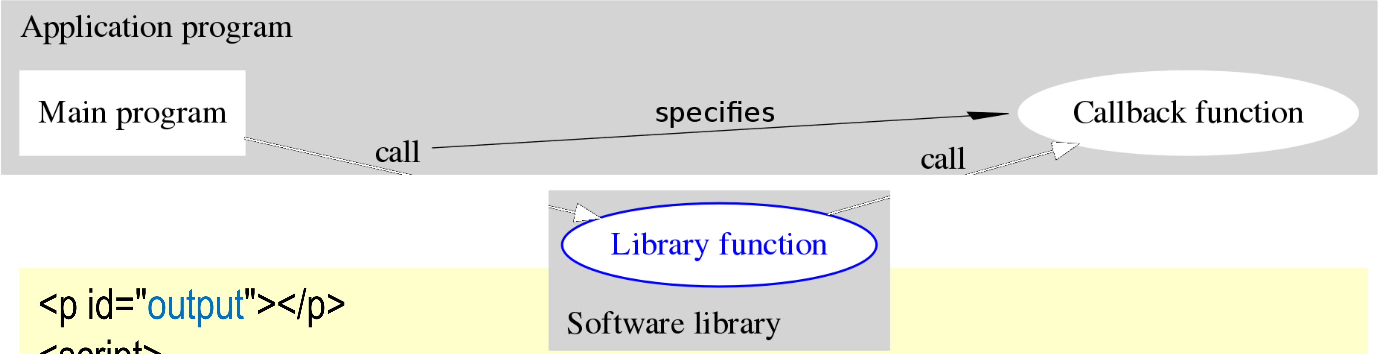
Some Modern Patterns

- Inversion of control
- Dependency injection

Inversion of Control [Pattern]

Hollywood Principle:
“Don't call us, we'll call you"

- Framework…
  - …first constructs an object (such as a controller)
  - …then passes control flow to it
- Principle:
  function pointers



- DOM example:

```
<p id="output"></p>
<script>
    var registeredListener = function () {
        document.getElementById("output").innerHTML = "listener called thru click";
    }
    document.addEventListener( "click", registeredListener, true );
    document.getElementById("output").innerHTML = "event handler registered";
</script>
```

Dependency Injection Pattern

- Description
  - object /function receives other objects/functions it requires, instead of creating them directly
- Problem / Applicability
  - separate concerns of constructing objects and using them → loosely coupled programs
- Solution
  - Analogy cars: uniform driver (client) interface,
    gas/diesel/electric engine injected by factory is unimportant to client
- Consequences
  - makes implicit dependencies explicit, helps solving these problems:
    - *How can a class be independent from the creation of the objects it depends on?*
    - *How can an application, and the objects it uses support different configurations?*

Types of Patterns

- Creational, ex:

Factory	Creates an instance of several families of classes
Builder	Separates object construction from its representation
Singleton	A class of which only a single instance can exist

- Structural, ex:

Adapter	Match interfaces of different classes
Composite	A tree structure of simple and composite objects
Decorator	Add responsibilities to objects dynamically
Proxy	An object representing another object

- Behavioral, ex:

Mediator	Defines simplified communication between classes
Observer	A way of notifying change to a number of classes
Template Method	Defer the exact steps of an algorithm to a subclass
Visitor	Defines a new operation to a class without change

Summary

- Design patterns = generic, re-usable design templates for OOP
  - Code templates, to be adapted by programmer
  - Faster, safer implementation through re-use
- three types of patterns: creational, structural, and behavioral
- Design pattern catalog
  - http://www.dofactory.com/net/design-patterns#list
- *It‘s practice – show it in interviews!*