

Design Patterns

Sommerville, Chapter 18

Instructor: Peter Baumann

email: p.baumann@jacobs-university.de

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 an efficient one – *learn from others!*
- Similar patterns occur over and over
 - Do not reinvent the wheel
 - Sharing knowledge of problem solving
 - Facilitate communication between programmers
 - Write elegant and graceful code
- Computer programming as art [Donald Knuth]
 - See conceptual beauty

Semiotics: Aspects of Language Use

- **Syntax**
 - how to write it (grammar)
 - Ex:
if (condition) statement;
if [condition]; then statement; fi
- **Semantics**
 - what to express (how it is evaluated)
 - Ex:
conditional evaluation
- **Pragmatics**
 - how to apply
 - Ex:
"goto considered bad"
- **Meta language**
 - describe the language of discourse
 - Ex:
BNF grammars

www.cs.sfu.ca/~cameron/Teaching/383/syn-sem-prag-meta.html

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

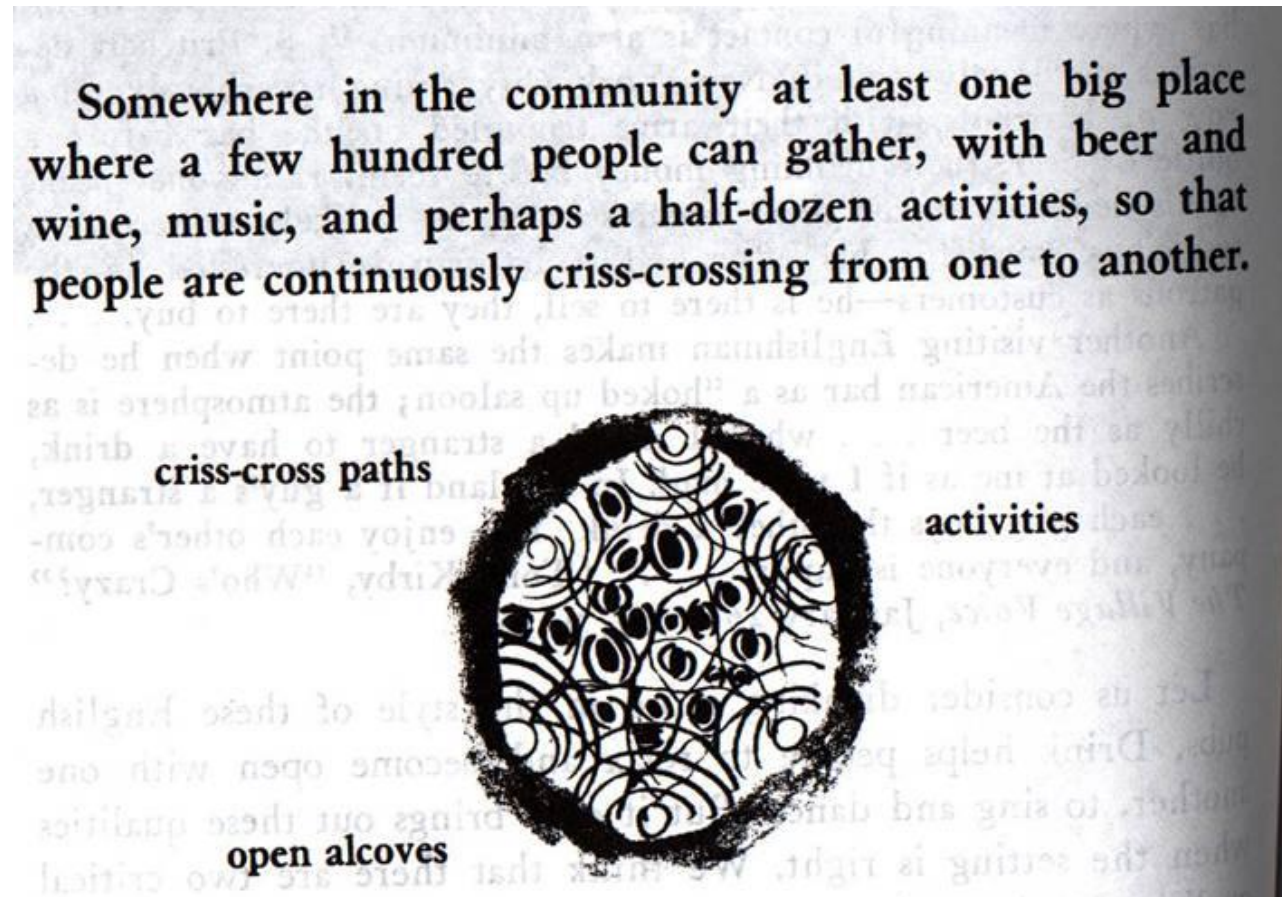
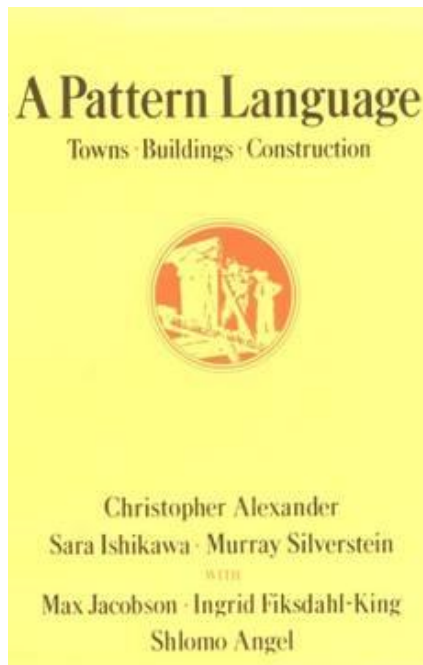
History of Design Patterns

- Architect: Christopher Alexander
 - *A Pattern Language* (1977)
 - *A Timeless Way of Building* (1979)
- “Gang of four”
 - Erich Gamma
 - Richard Helm
 - Ralph Johnson
 - John Vlissides
- *Design Patterns: Elements of Reusable Object-Oriented Software* (1995)

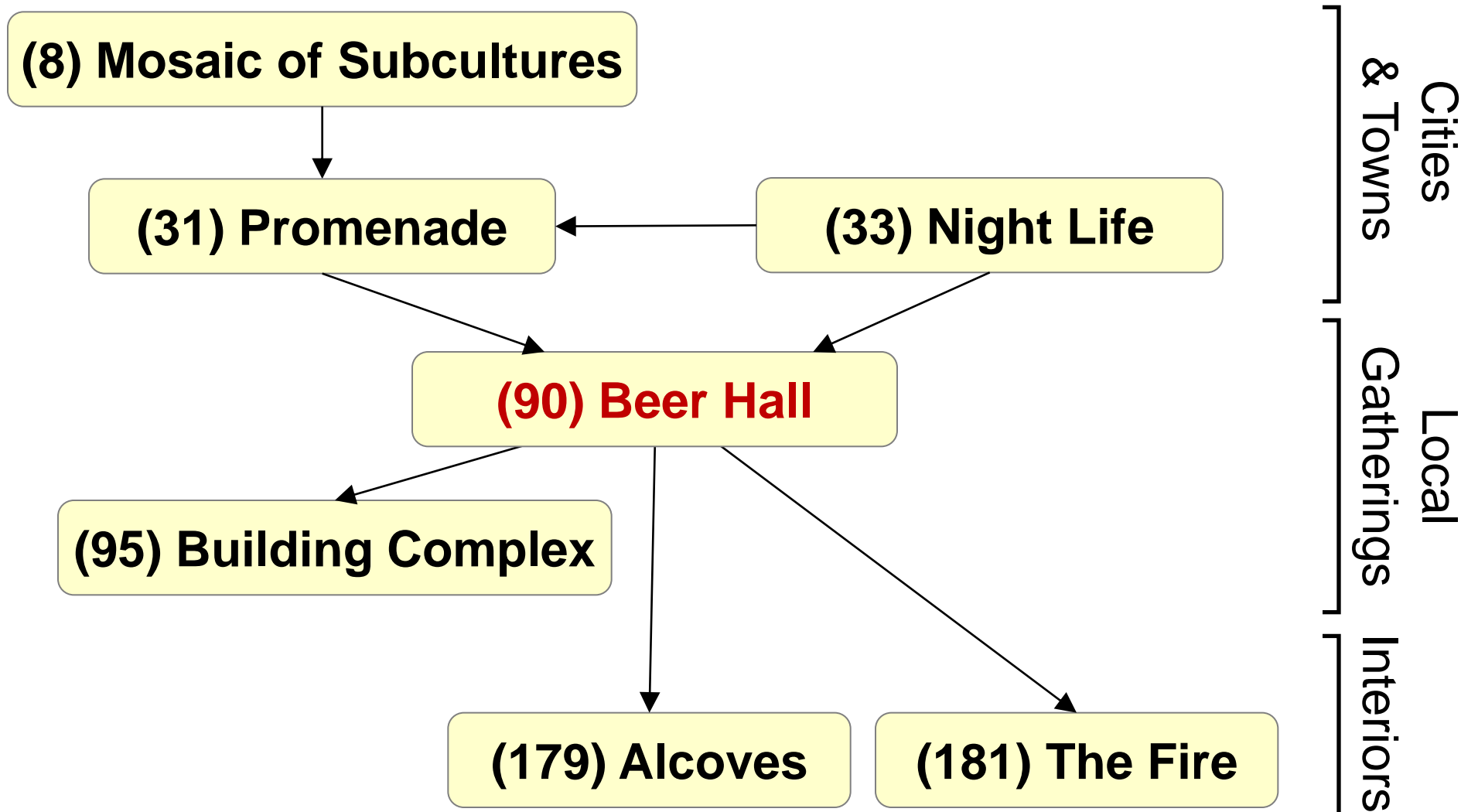
Design Patterns in Architecture

- First used in architecture [C. Alexander]

- Ex. How to create a beer hall where people socialize?



Design Patterns in Architecture



Pattern Elements

- Name
 - A meaningful pattern identifier
- Description
- Problem / Applicability description
- Solution description
 - Not concrete design but template for design solution that can be instantiated in different ways
- Consequences
 - The results and trade-offs of applying the pattern

Patterns by Example: 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
-instance : Singleton
-Singleton() +Instance() : Singleton

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;
```

Singleton, Revisited

Problems:

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

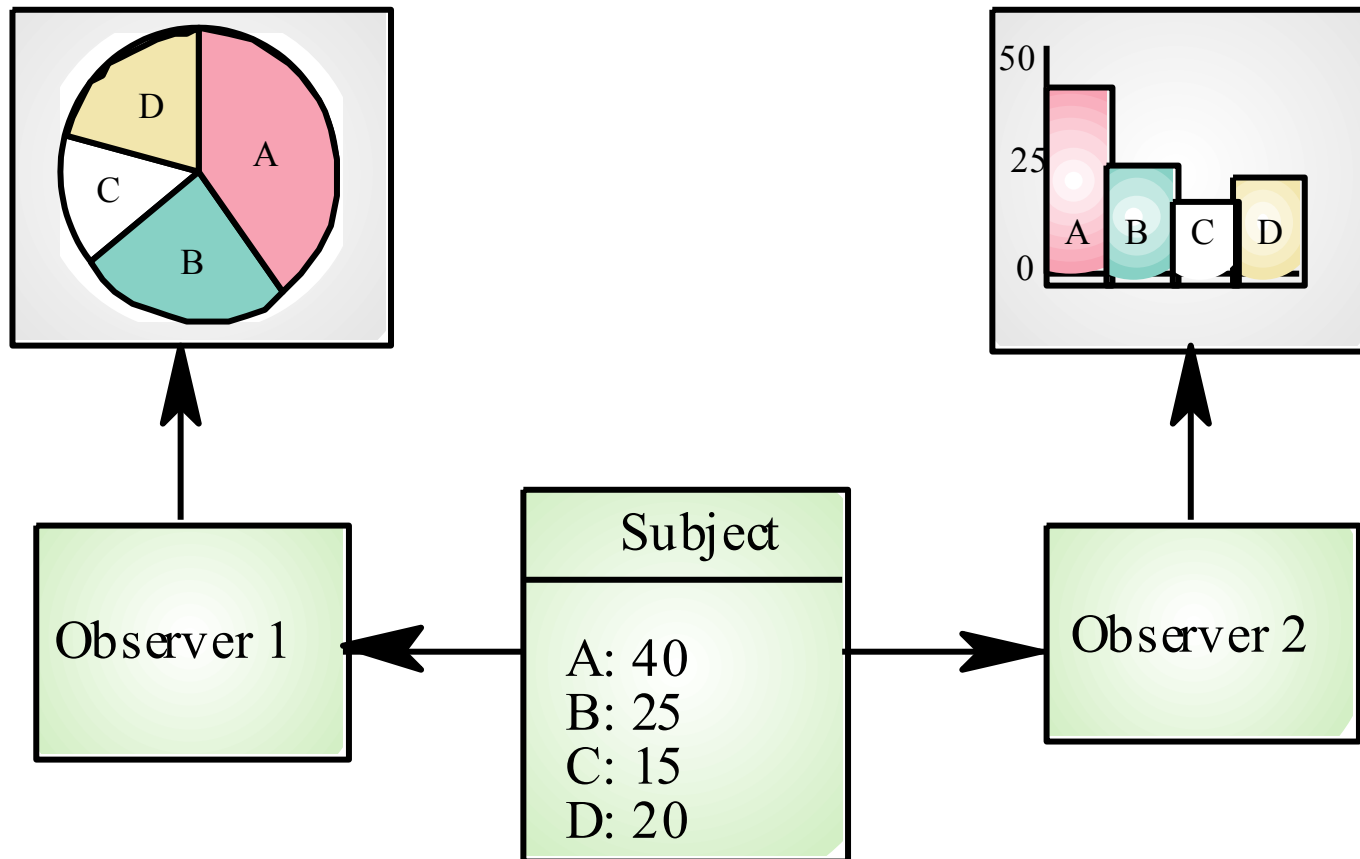
// Singleton pattern

```
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&);
}
```

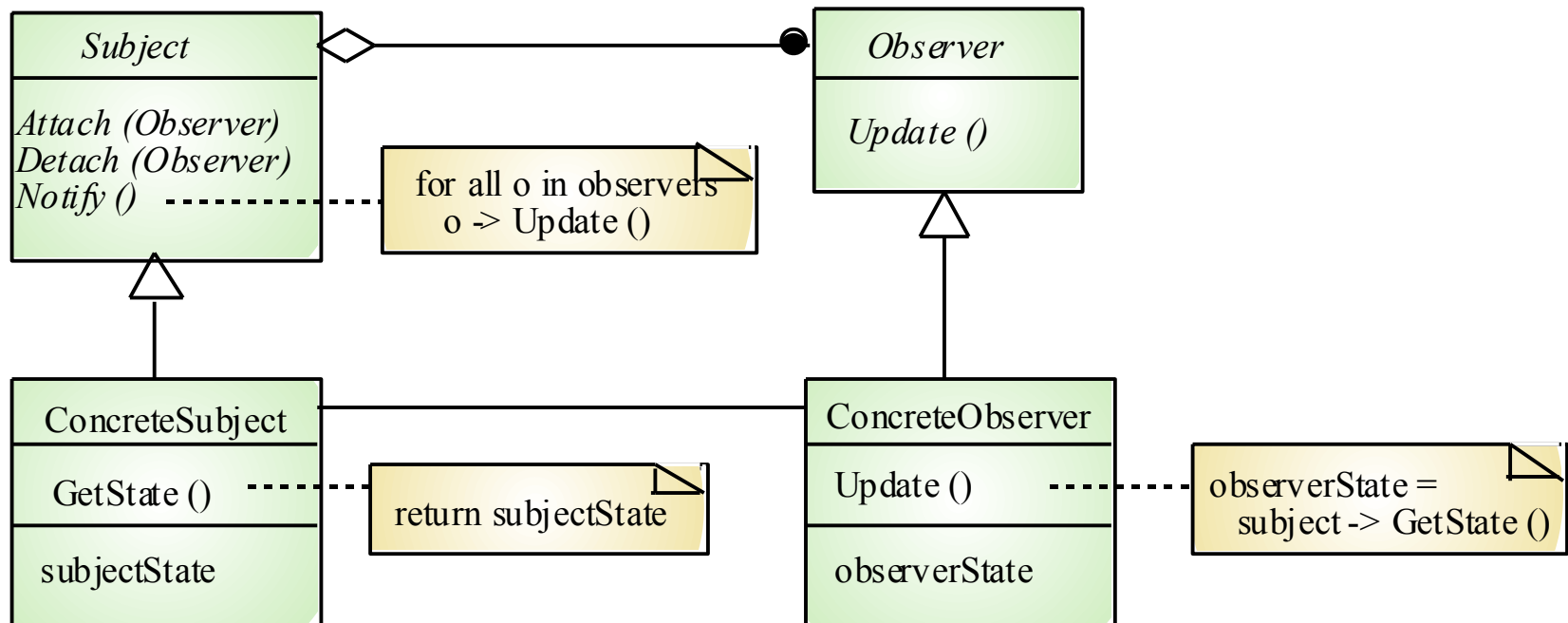
Multiple displays enabled by Observer



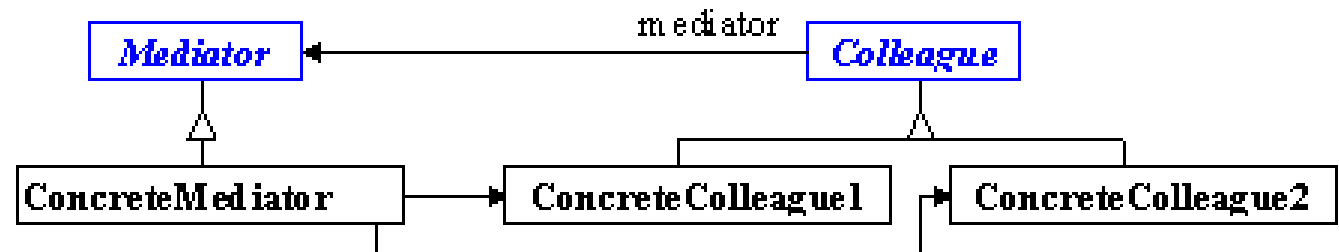
The Observer Pattern

- 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

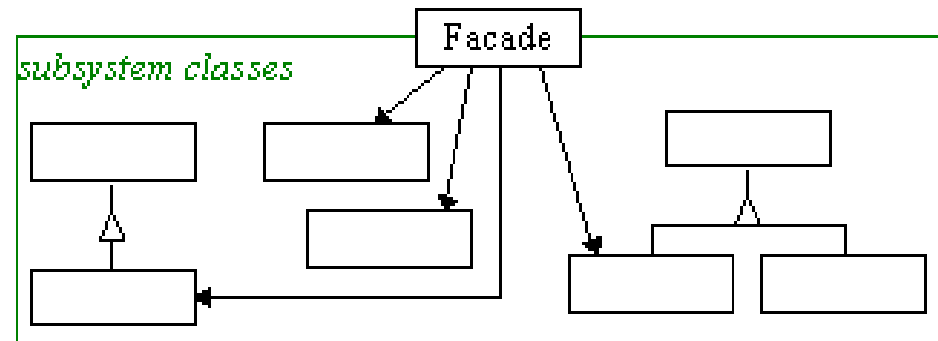
■ Consequences

- Limits subclassing; Decouples colleagues; Simplifies object protocols; Abstracts how objects cooperate; Centralizes control

The Façade Pattern

■ Description

- Provides a **unified interface** to a **set of interfaces** in a subsystem
- Defines a **higher-level interface** that makes subsystem easier to use



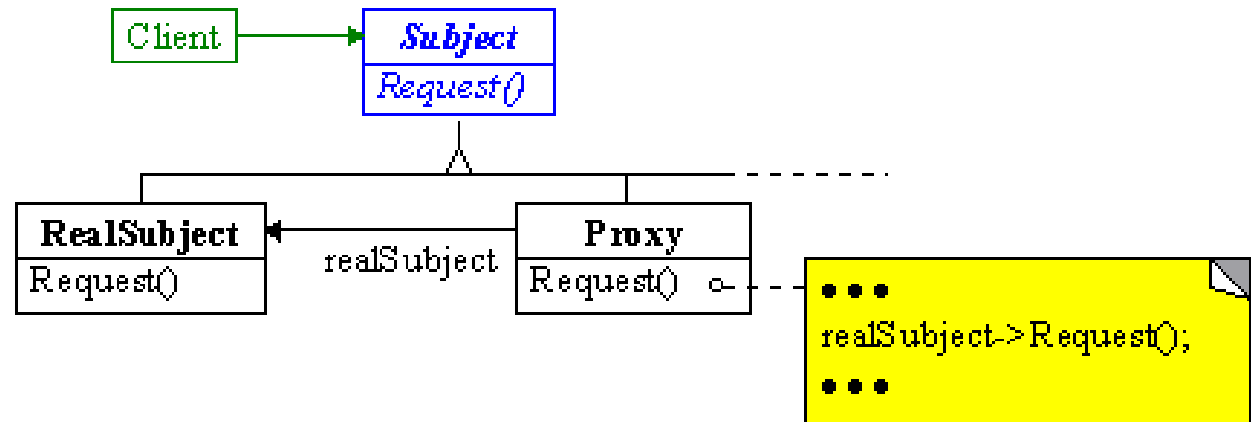
■ Applicability

- Need to provide a **simple** interface to a **complex** system
- Need to **decouple** a subsystem from its clients
- Need to provide an **interface to a software layer**

■ Consequences

- Shields clients from subsystem components
- Promotes weak coupling between the subsystem and its clients

The Proxy Pattern



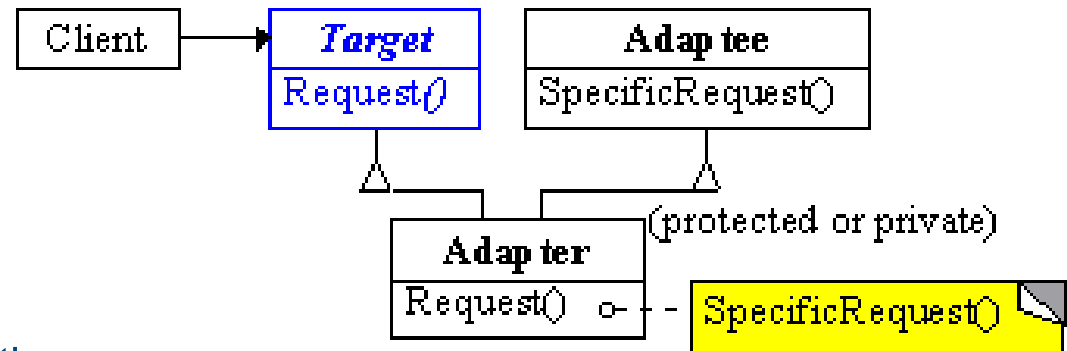
■ Description

- Provide a surrogate or placeholder for another object to control access to it

■ Problem / Applicability

- Remote proxies can hide the fact that a real object is in **another address space**
- Virtual proxies can **create expensive objects** on demand
- Protection proxies can **control access** to an object
- Smart references can perform **additional action** above a simple pointer

The Adapter Pattern



■ Description

- Adapter lets classes work together that could not otherwise because of incompatible interfaces

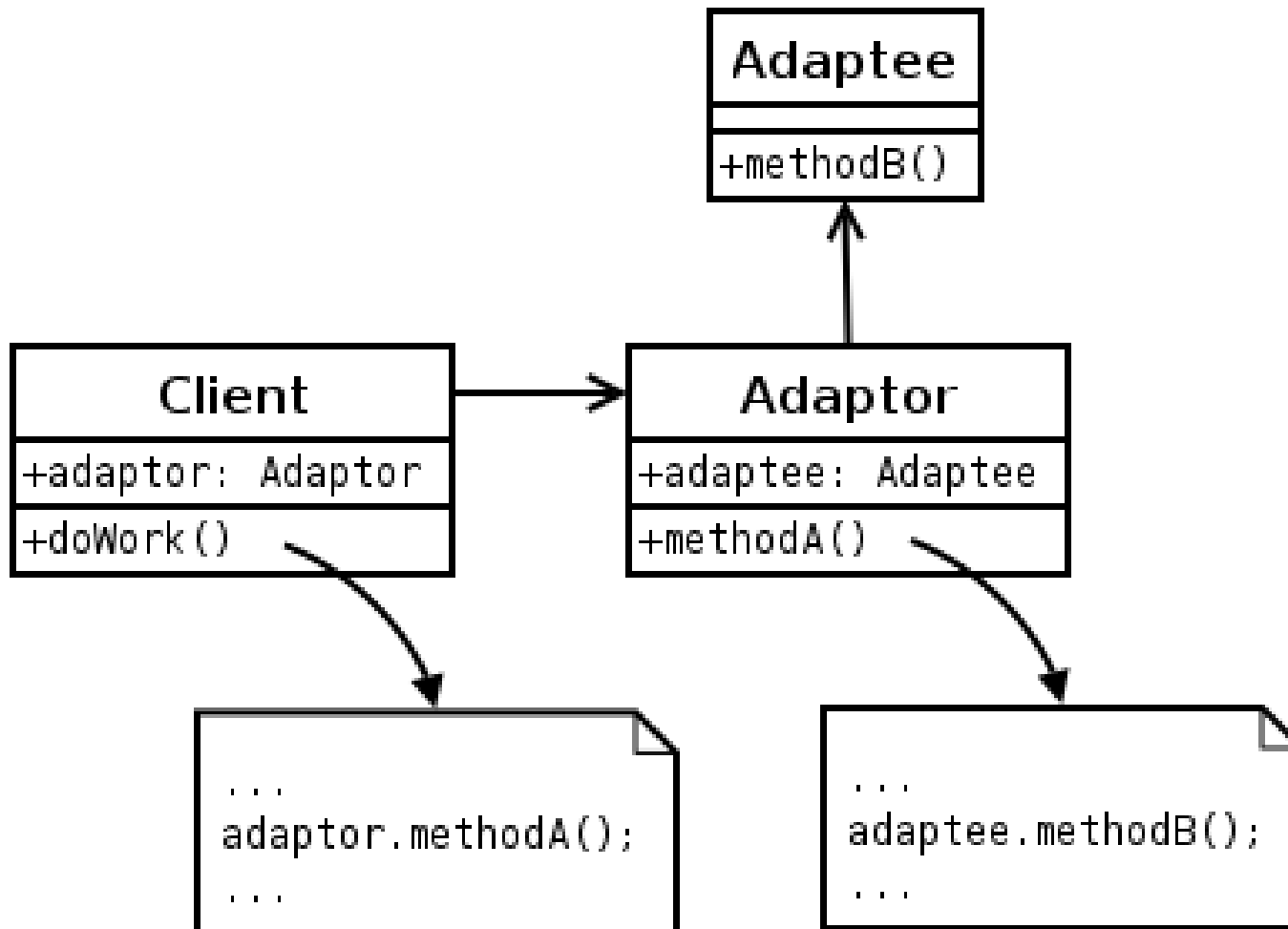
■ Problem / Applicability

- 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

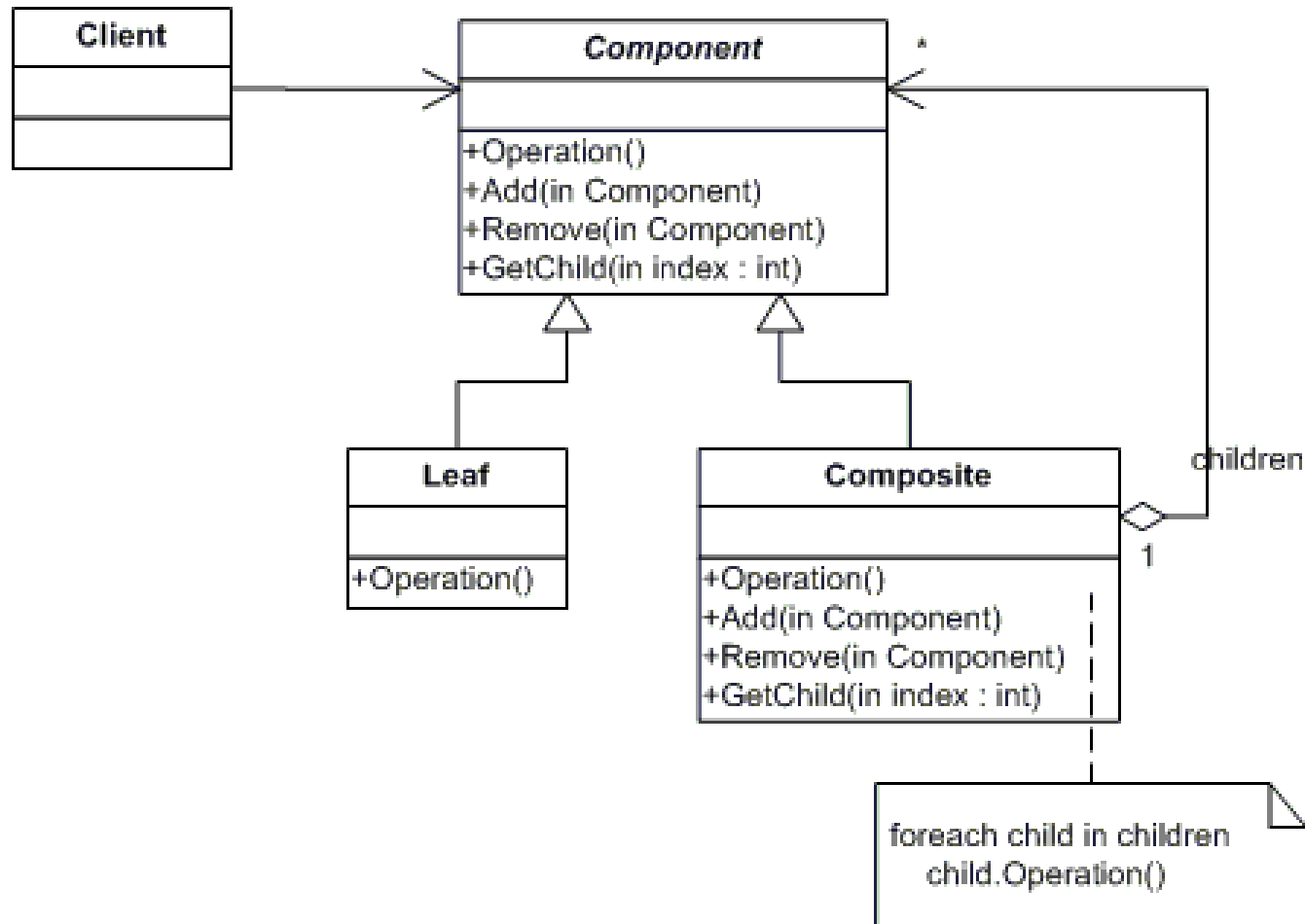
Adapter: Another View [Wikipedia]



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



Types of Patterns

■ Creational

- Abstract Factory Creates an instance of several families of classes
- Builder Separates object construction from its representation
- Factory Method Creates an instance of several derived classes
- Prototype A fully initialized instance to be copied or cloned
- Singleton A class of which only a single instance can exist

■ Structural Patterns

- Adapter Match interfaces of different classes
- Bridge Separates an object's interface from its implementation
- Composite A tree structure of simple and composite objects
- Decorator Add responsibilities to objects dynamically
- Façade A single class that represents an entire subsystem
- Flyweight A fine-grained instance used for efficient sharing
- Proxy An object representing another object

Types of Patterns (contd.)

■ Behavioral Patterns

- Chain of Resp. A way of passing a request between a chain of objects
- Command Encapsulate a command request as an object
- Interpreter A way to include language elements in a program
- Iterator Sequentially access the elements of a collection
- Mediator Defines simplified communication between classes
- Memento Capture and restore an object's internal state
- Observer A way of notifying change to a number of classes
- State Alter an object's behavior when its state changes
- Strategy Encapsulates an algorithm inside a class
- 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!*