廣東技術師範大學
Guangdong Polytechnic Normal University

《系统分析与设计》课程

Instructor: Wen Jianfeng
Email: wjfgdin@qq.com

系统分析与设计

第6讲：设计模式-结构型

提纲

- 代理模式（Proxy）
- 组合模式（Composite）

		Purpose 目的		
		Creational 创建型（**5**种）	**Structural** 结构型（**7**种）	**Behavioral** 行为型（**11**种）
Scope 范围	**Class** 类	Factory Method [工厂方法]	Adapter (class) [适配器（类）]	Interpreter [解释器] Template Method [模板方法]
	Object 对象	Abstract Factory [抽象工厂] Builder [建造者 / 生成器] Prototype [原型] Singleton [单例]	Adapter (object) [适配器（对象）] Bridge [桥接] **Composite** [组合] Decorator [装饰器] Façade [门面 / 外观] Flyweight [享元] **Proxy** [代理]	Chain of Responsibility [责任链 / 职责链] Command [命令] Iterator [迭代器] Mediator [中介者] Memento [备忘录] Observer [观察者] State [状态] Strategy [策略] Visitor [访问者]

结构型设计模式

- Structural patterns are mostly concerned with object composition or in other words how the entities can use each other. Or yet another explanation would be, they help in answering "How to build a software component?“

  结构型设计模式主要关注对象的组合，或者说实体间怎样才能互相调用。另一种解释是，结构型设计模式有助于回答“怎样构建一个软件组件”的问题。

结构型设计模式

- In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.

  结构型设计模式通过指定一种简单的途径来实现实体间的关系，从而简化设计。

代理模式
Proxy Pattern

代理模式--现实世界中的例子

- Have you ever used an access card to go through a door? There are multiple options to open that door i.e. it can be opened either using access card or by pressing a button that bypasses the security. The door's main functionality is to open but there is a proxy added on top of it to add some functionality.

  有多种方式可以通过一个门禁，如可以用门禁卡，可以按按钮。门的主要功能是打开，但有一个代理加在它上面以添加一些功能。

代理模式--理解要点

- Using the proxy pattern, a class represents the functionality of another class.
  使用代理模式，一个类表现出另一个类的功能。

代理模式--理解要点

- A proxy, in its most general form, is a class functioning as an interface to something else. A proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked.（翻译见下页）

代理模式--理解要点

代理最一般的形式是：作为其他东西的接口的类。

代理是一个包装器对象或代理对象，客户调用它以访问其背后真正提供服务的对象。

使用代理能简单地转发到真正的对象，或者能提供额外的逻辑。

在代理中，能提供额外的功能。例如，在真实对象中有耗资源的操作时，可以进行缓存；或者在调用真实对象的操作前检查前置条件。

代理模式 – Intent【意图】

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

  给某一个对象提供一个代理，并由代理对象控制对原对象的引用

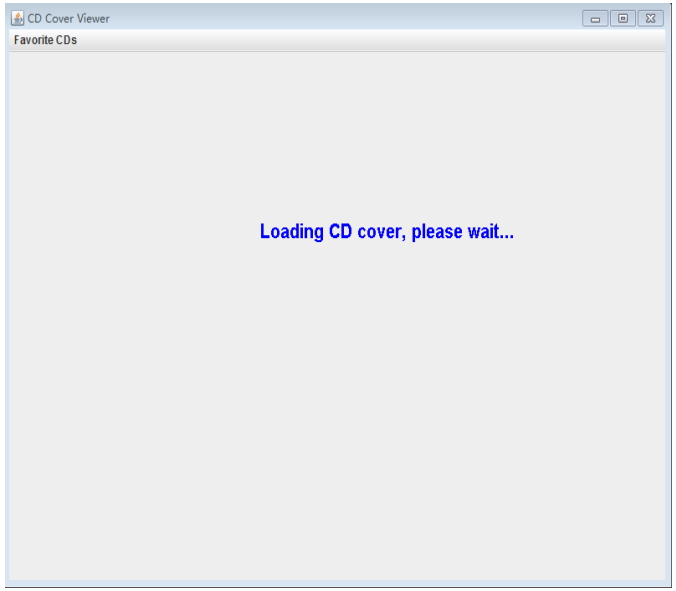
  - Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create or in need of securing.
    - Virtual Proxy：虚拟代理控制对创建时需消耗较大资源的对象的访问
    - Remote Proxy：远程代理控制对远程对象的访问
    - Protection Proxy：保护代理控制对需访问权限的对象的访问

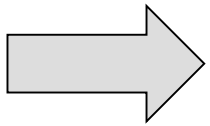
代理模式 – Motivation【动机】

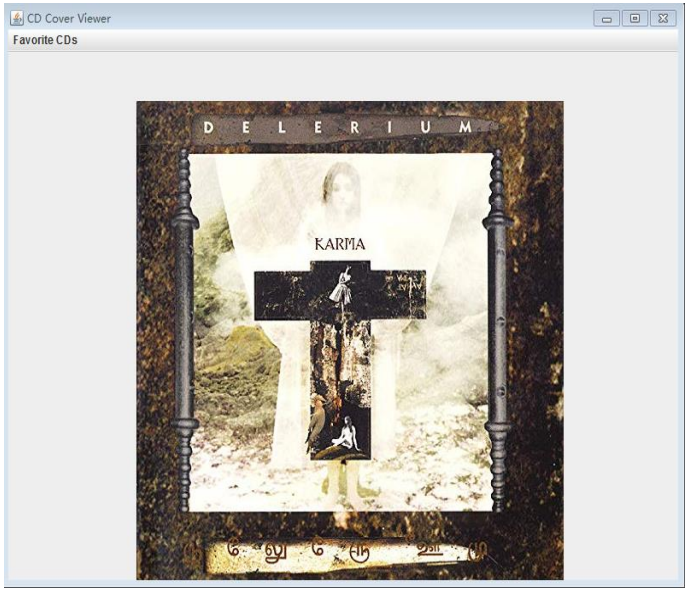
- 动机
  - 在某些情况下，一个客户不想或者不能直接引用一个对象，此时可以通过一个称之为“代理”的第三者来实现间接引用。代理对象可以在客户端和目标对象之间起到中介的作用，并且可以通过代理对象去掉客户不能看到的内容和服务或者添加客户需要的额外服务。
  - 通过引入一个新的对象（如小图片和远程代理对象）来实现对真实对象的操作或者将新的对象作为真实对象的一个替身，这种实现机制即为代理模式，通过引入代理对象来间接访问一 个对象，这就是代理模式的模式动机。

代理模式 – Motivation【动机】

- 例如：假设要开发一个显示音乐CD的封面图片的功能，图片需要通过网络获取
  - 由于网络负载和带宽的原因，图片加载需要耗费一点时间
  - 在等待图片加载时，我们希望先显示一段文本信息作为提示，当图片加载完成后，再把文本替换成图片

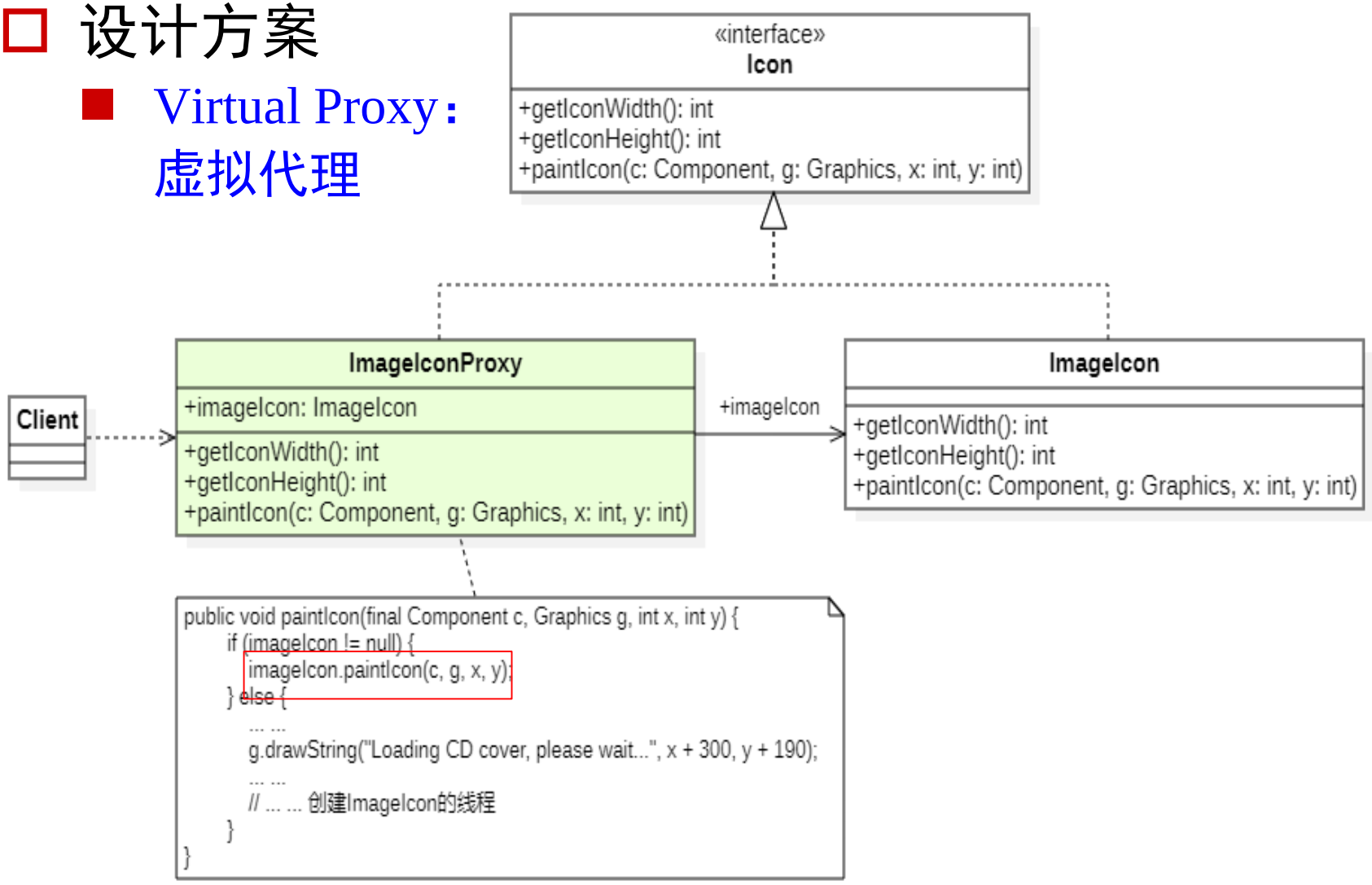






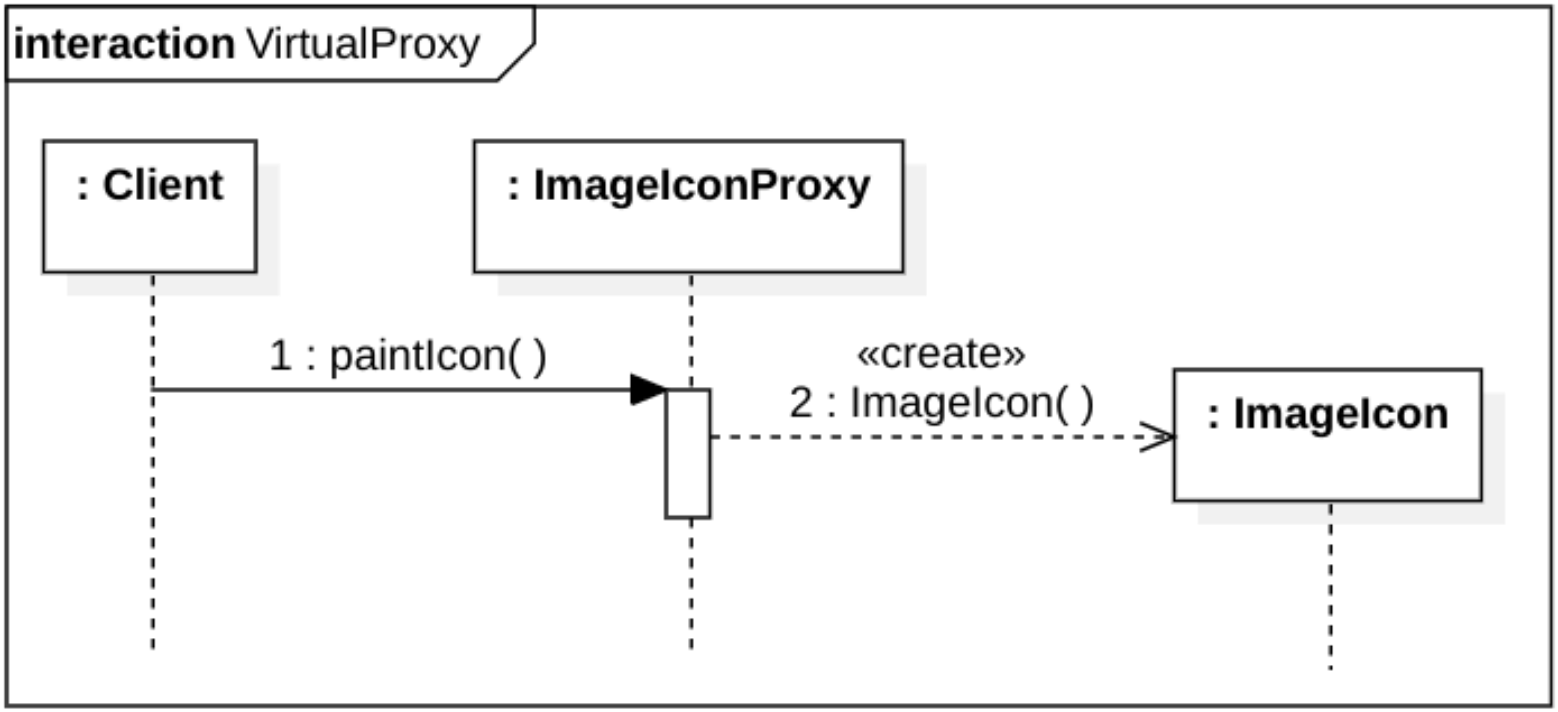
代理模式 – Motivation【动机】

- 设计方案
  - Virtual Proxy：虚拟代理



代理模式 – Motivation【动机】

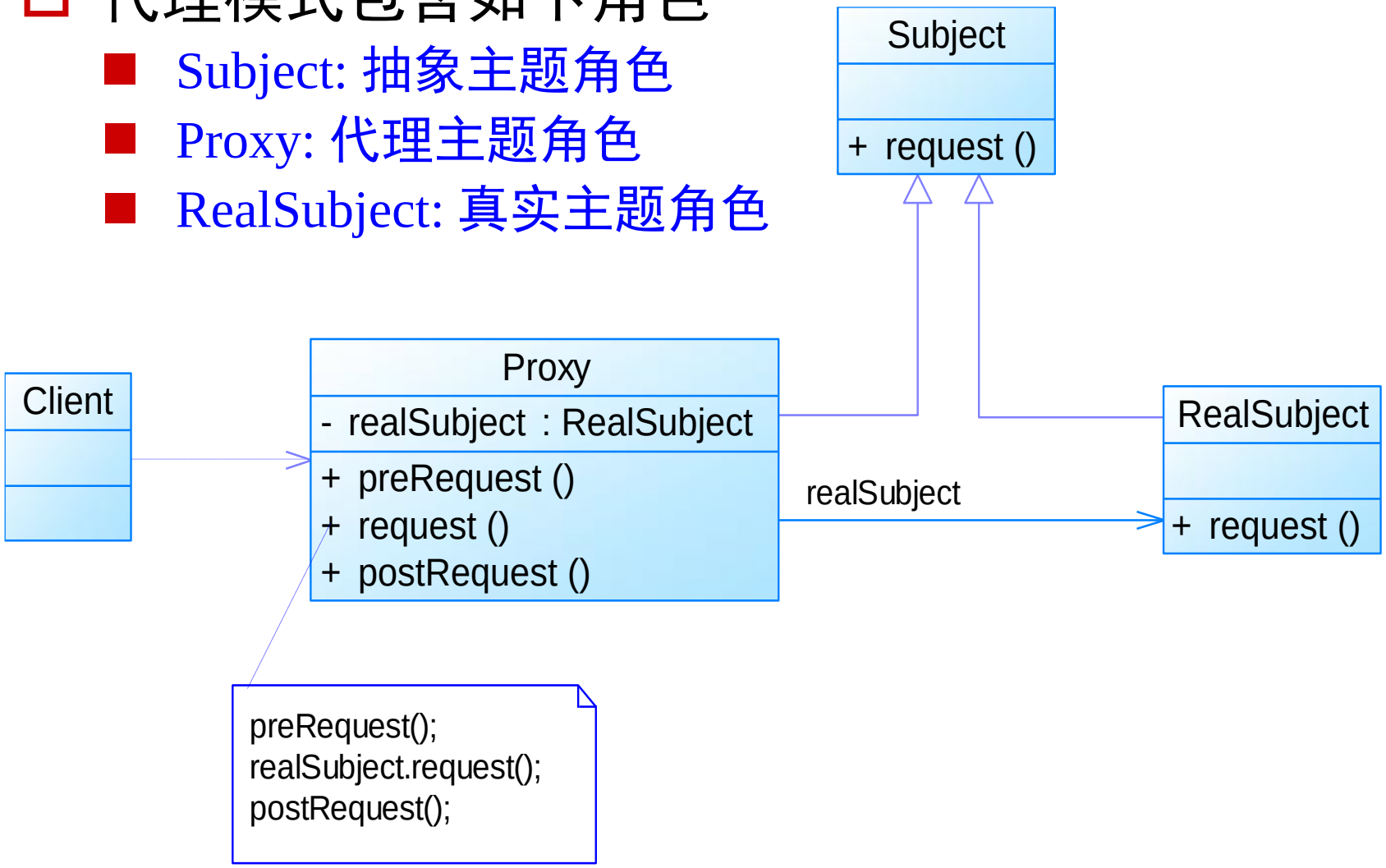
- 设计方案
  - Virtual Proxy：虚拟代理



```
public void paintIcon(final Component c, Graphics g, int x, int y) {
    if (imageIcon != null) {
        imageIcon.paintIcon(c, g, x, y);
    } else {
        g.setColor(Color.BLUE);
        g.setFont(new Font("", Font.BOLD, 20));
        g.drawString("Loading CD cover, please wait...", x + 300, y + 190);
        if (!retrieving) {
            retrieving = true;
            retrievalThread = new Thread(new Runnable() {
                public void run() {
                    try {
                        imageIcon = new ImageIcon(imageURL, "CD Cover");
                        c.repaint();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            });
            retrievalThread.start();
        }
    }
}
```

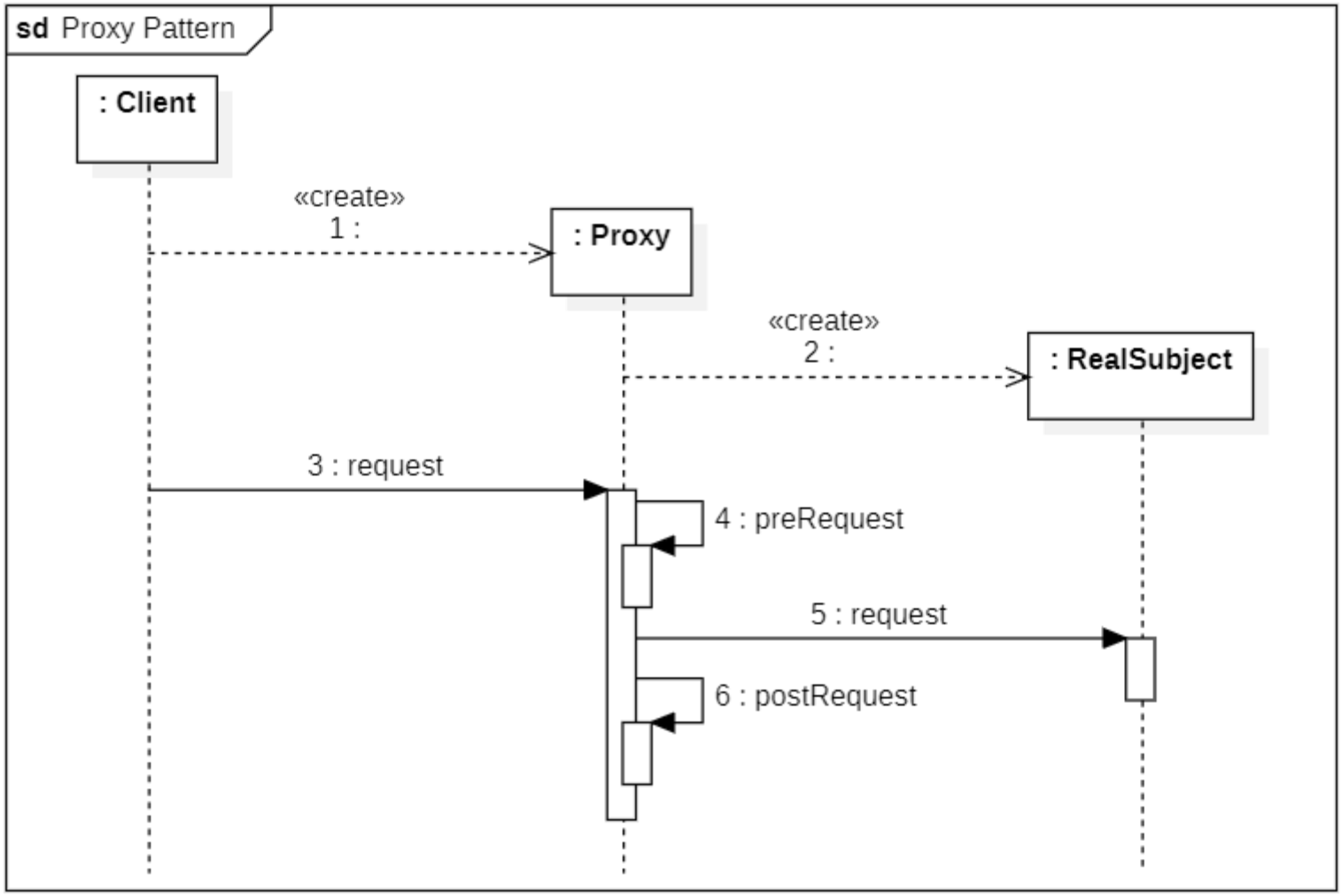
代理模式 – Structure【结构】

- 代理模式包含如下角色
  - Subject: 抽象主题角色
  - Proxy: 代理主题角色
  - RealSubject: 真实主题角色



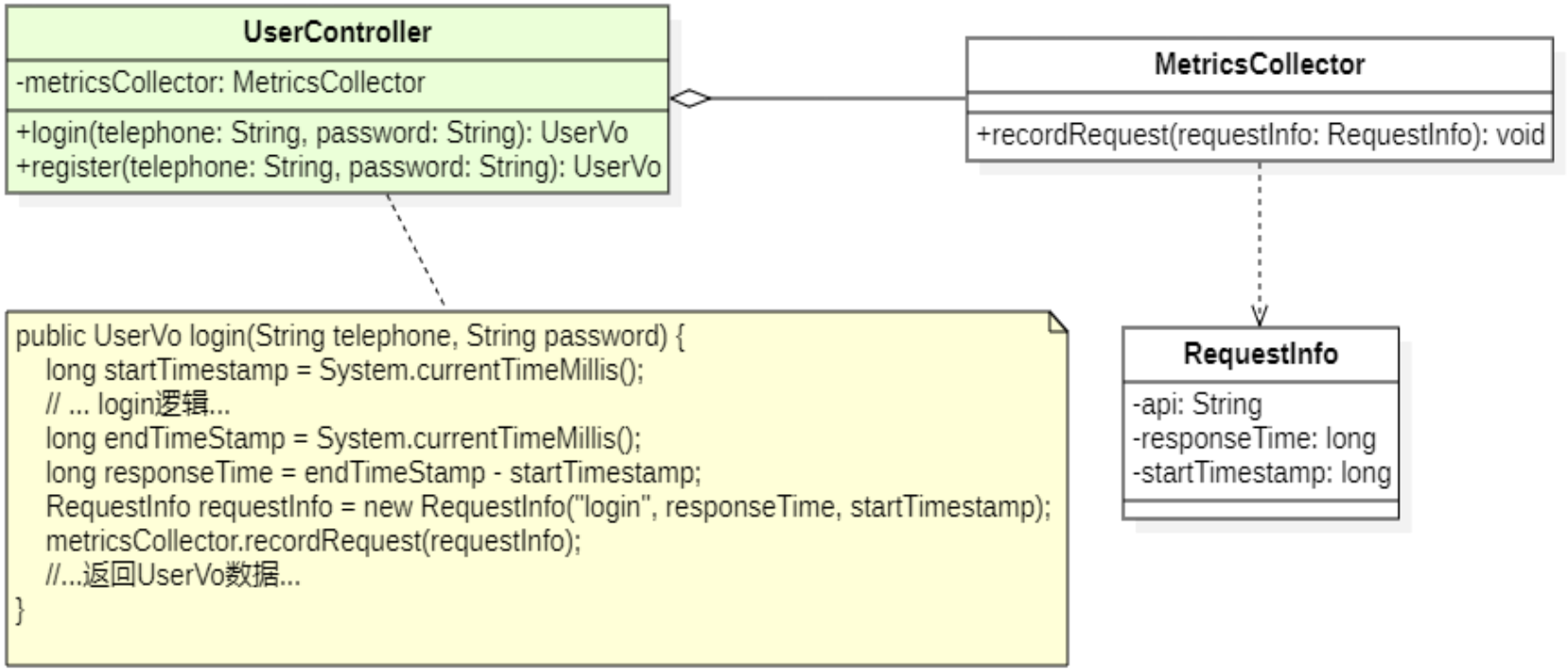
代理模式 – Structure【结构】

- 代理模式的顺序图



代理模式 – Virtual Proxy

- 举例
  - 开发一个性能计数器，用来收集接口（如用户登录、注册）请求的原始数据，如访问时间、处理时长等
  - 初始设计：

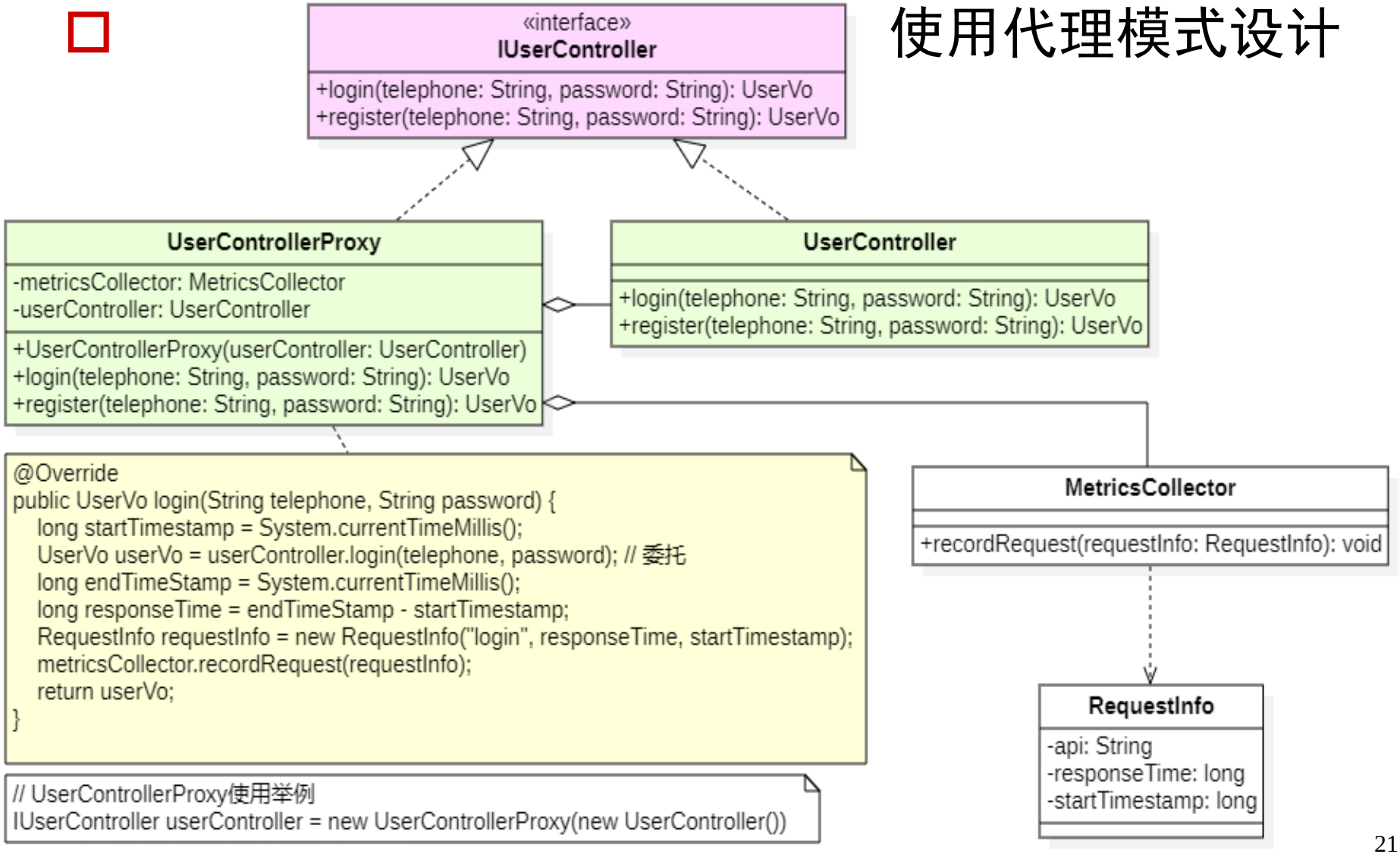


代理模式 – Virtual Proxy

- 初始设计存在的问题：
  - 性能计数器框架代码侵入到业务代码中，跟业务代码高度耦合
  - 收集接口请求的代码跟业务代码无关，业务类职责应该要单一，只聚焦业务处理

代理模式 – Virtual Proxy

- 使用代理模式设计



```
public class UserControllerProxy implements IUserController {
  private MetricsCollector metricsCollector;
  private UserController userController;

  public UserControllerProxy(UserController userController) {
    this.userController = userController;
    this.metricsCollector = new MetricsCollector();
  }

  @Override
  public UserVo login(String telephone, String password) {
    long startTimestamp = System.currentTimeMillis();

    UserVo userVo = userController.login(telephone, password); // 委托

    long endTimeStamp = System.currentTimeMillis();
    long responseTime = endTimeStamp - startTimestamp;
    RequestInfo requestInfo = new RequestInfo("login", responseTime,
                                              startTimestamp);
    metricsCollector.recordRequest(requestInfo);

    return userVo;
  }
  ... ...
}
```

代理模式 – Virtual Proxy

- 在上例的代理模式设计中，代理类和原始类需要实现相同的接口。但是，如果被代理的类是外部系统的类（即我们无法修改它去实现新添加的接口），这时该怎么办呢？
  - 解决方法是：使用继承的方式，即代理类继承原始类

```
public class UserControllerProxy extends UserController {
  private MetricsCollector metricsCollector;

  public UserControllerProxy() {
    this.metricsCollector = new MetricsCollector();
  }
  public UserVo login(String telephone, String password) {
    ... ...
    UserVo userVo = super.login(telephone, password);
    ... ...
  }
  ... ...
}
```

代理模式 – 【适用情形】

- 根据代理模式的使用目的，常见的代理模式有以下几种类型：
  - 虚拟代理（Virtual Proxy）
    - 如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，真实对象只在需要时才会被真正创建
  - 远程代理（Remote Proxy）
    - 为一个位于不同的地址空间的对象提供一个本地的代理对象，这个不同的地址空间可以是在同一台主机中，也可是在另一台主机中
    - 如：JDK的RMI机制（https://docs.oracle.com/javase/tutorial/rmi/overview.html）

代理模式 – 【适用情形】

- 保护代理（Protection Proxy）
  - 控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限
- 智能引用代理（Smart Reference）
  - 当一个对象被引用时，提供一些额外的操作
  - 如：将对象被调用的次数记录下来等

代理模式 – 【优缺点】

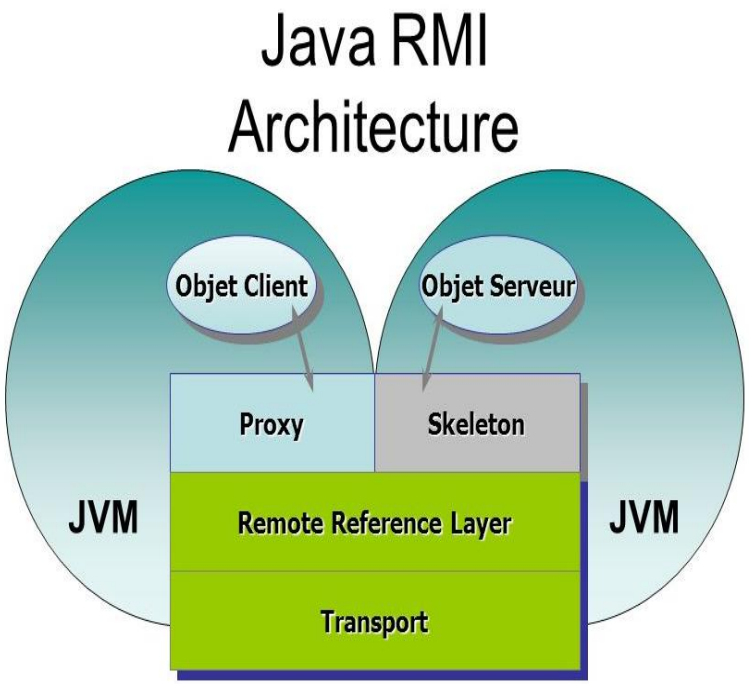
- 代理模式的优点
  - 代理模式能够协调调用者和被调用者，在一定程度上降低了系统的耦合度
  - 远程代理使得客户端可以访问在远程机器上的对象，远程机器可能具有更好的计算性能与处理速度，可以快速响应并处理客户端请求
  - 虚拟代理通过使用一个小对象来代表一个大对象，可以减少系统资源的消耗，对系统进行优化并提高运行速度
  - 保护代理可以控制对真实对象的使用权限

代理模式 – 【优缺点】

- 代理模式的缺点
  - 由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢
  - 实现代理模式需要额外的工作，有些代理模式的实现非常复杂

代理模式 – 在JDK中的应用

- Java JDK中的代理模式例子
  - 远程方法调用（RMI）
    - RMI：Remote Method Invocation
    - 在本地创建一个对象作为远程对象的代理
    - https://docs.oracle.com/javase/tutorial/rmi/index.html



组合模式
Composite Pattern

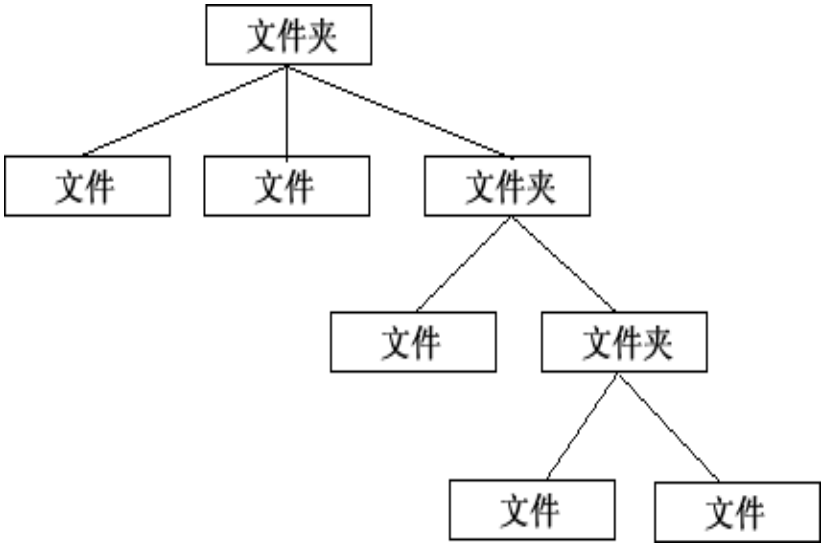
组合模式 – Intent【意图】

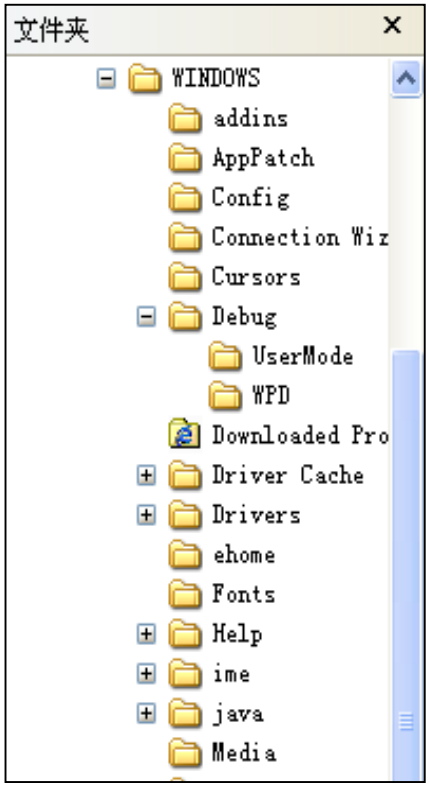
- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

  将一组对象组织成树形结构，以表示一种“部分-整体”的层次结构。组合模式让客户可以统一单个对象（即叶子对象）和组合对象（即容器对象）

组合模式 – Motivation【动机】

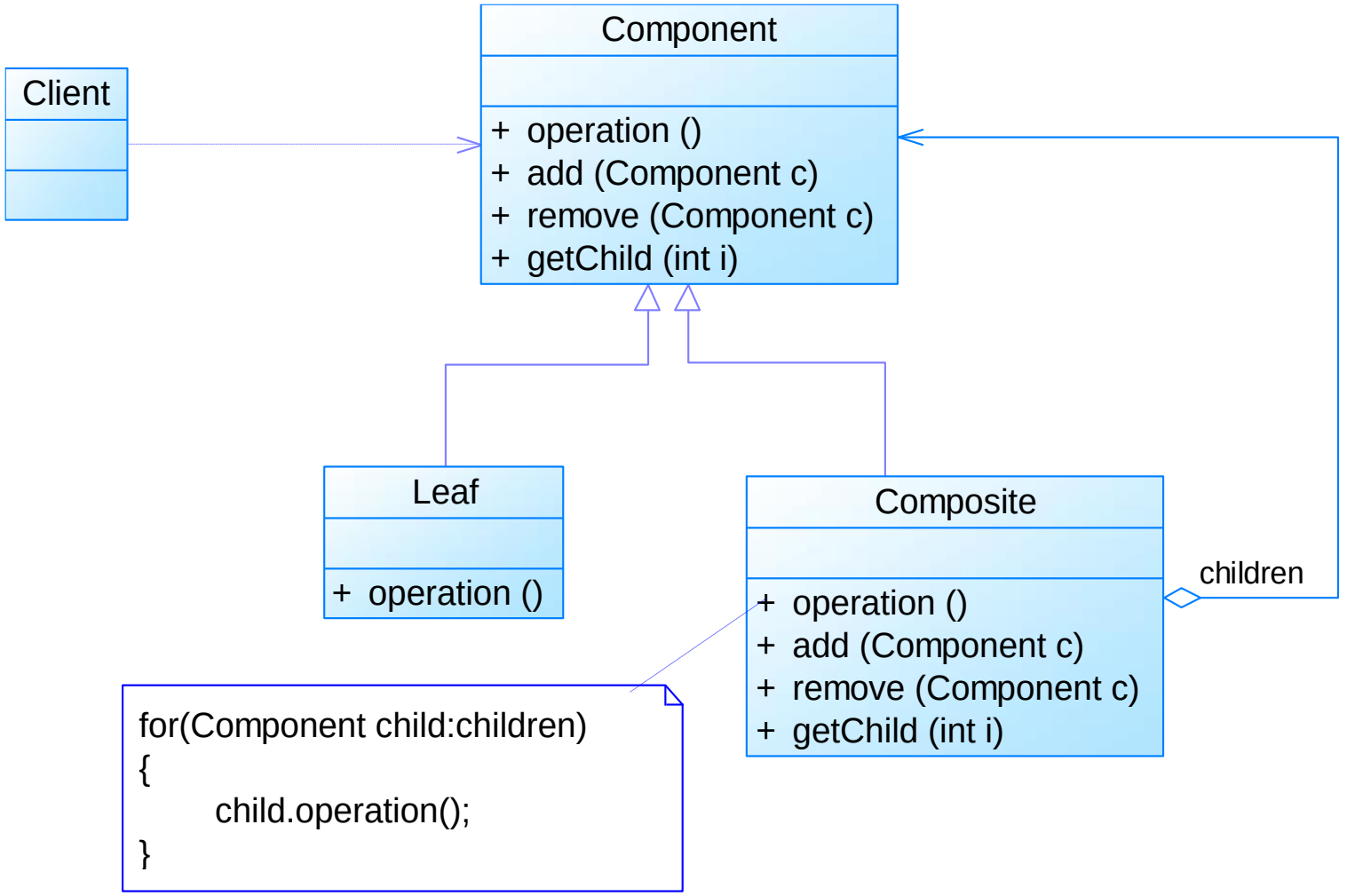
- 对于树形结构，当容器对象（如文件夹）的某一个方法被调用时，将遍历整个树形结构，对所有成员对象（包括容器对象和叶子对象，如子文件夹和文件）的同一方法进行调用执行（递归调用）
  - 如：删除某个目录下的子目录或文件





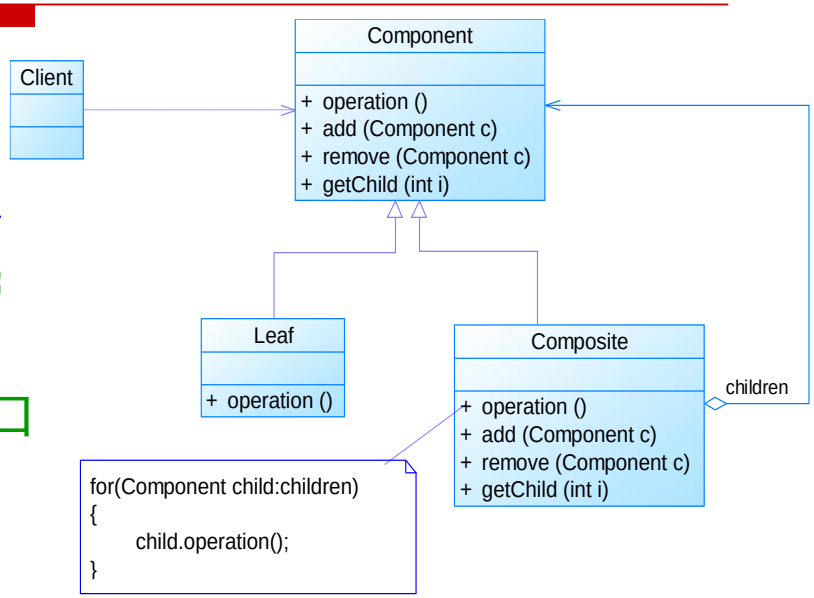
组合模式 – Structure【结构】

- 组合模式设计类图



组合模式 – Structure【结构】

- 组合模式包含如下角色：
  - Component—抽象构件角色
    - 为参加组合的对象声明接口；
    - 实现接口默认的行为；
    - 声明访问和管理子构件的接口
  - Leaf—叶子构件角色
    - 表示组合中的叶子对象，叶子没有子对象；
    - 它定义组合中的原子对象的行为
  - Composite—容器构件角色
    - 表示组合中有子对象的对象，即容器对象；
    - 定义容器对象的行为；
    - 存储子对象；
    - 实现Component接口中与子对象相关的操作

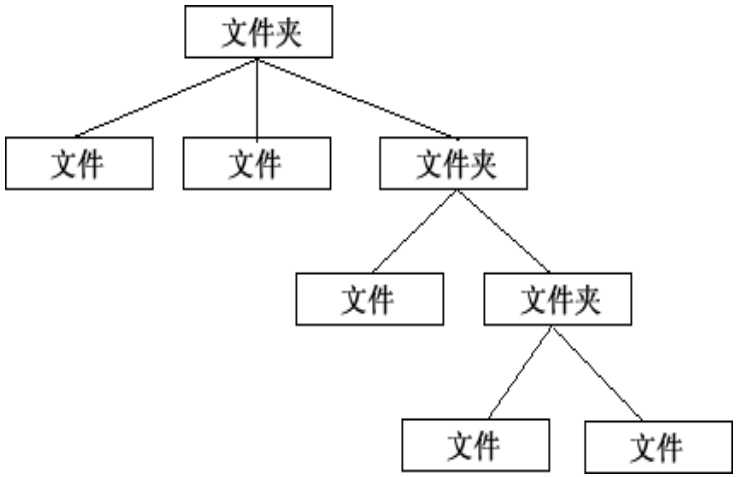


组合模式 – Structure【结构】

- 分析
  - 组合模式的关键是定义了一个抽象构件类Component，它既可以代表叶子Leaf，又可以代表容器Composite。针对Component类进行编程，无须知道它到底表示的是叶子还是容器，可以对其进行统一处理
  - 容器对象Composite与抽象构件类Component之间还建立一个聚合关联关系，在容器对象中既可以包含叶子，也可以包含容器，以此实现递归组合，形成一个树形结构

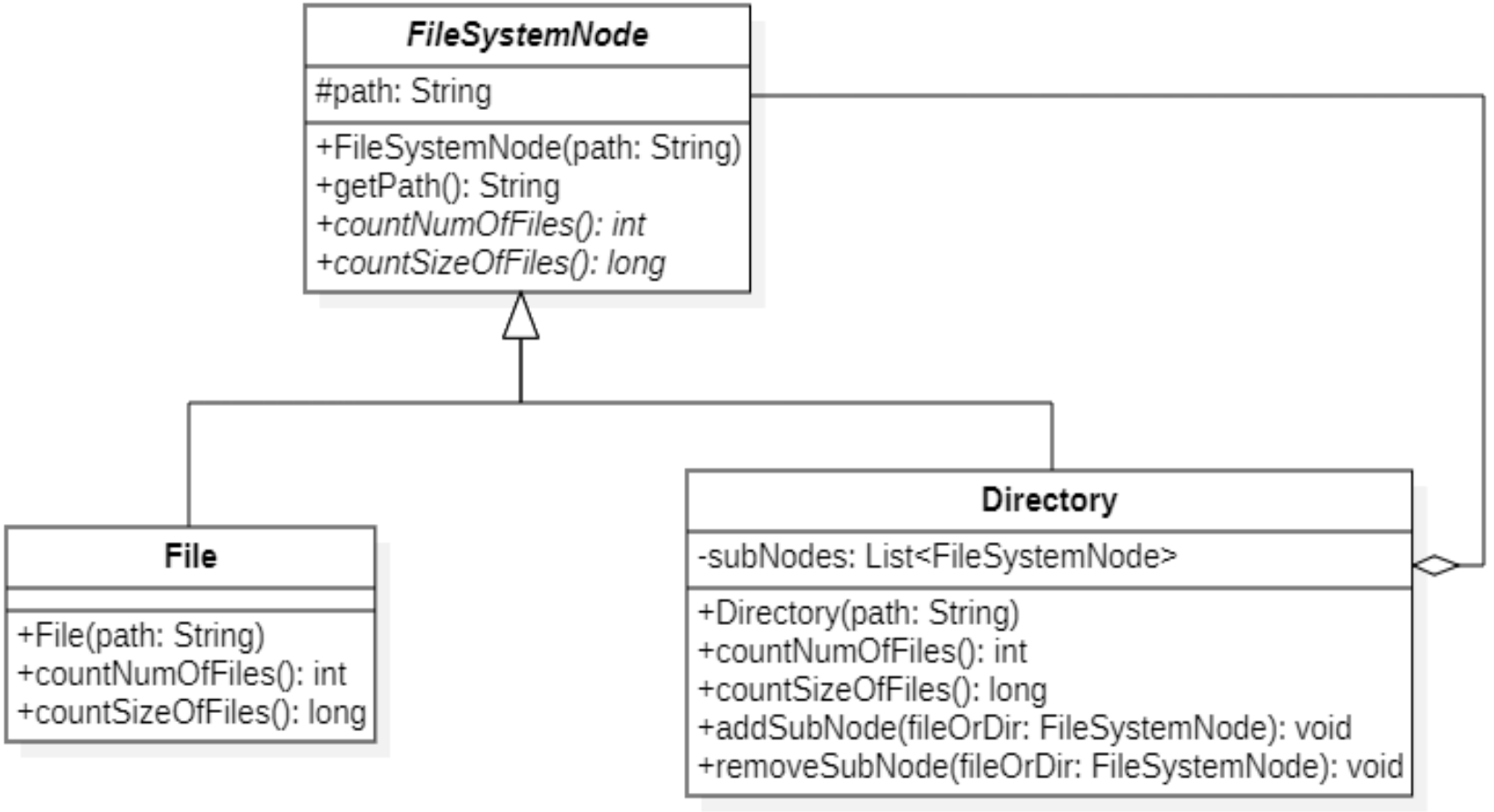
组合模式 – 举例

- 举例：文件目录
  - 在计算机文件系统中有目录（或称文件夹），目录里面有文件或者子目录，在子目录里面还会有其他文件或子目录
  - 用组合模式实现下面这些功能：
    - 动态地添加、删除某个目录下的子目录或文件；
    - 统计指定目录下的文件个数；
    - 统计指定目录下的文件总大小。



组合模式 – 举例

- 设计类图


FileSystemNode
#path: String
+FileSystemNode(path: String)
+getPath(): String
+countNumOfFiles(): int
+countSizeOfFiles(): long
File
+File(path: String)
+countNumOfFiles(): int
+countSizeOfFiles(): long
Directory
-subNodes: List<FileSystemNode>
+Directory(path: String)
+countNumOfFiles(): int
+countSizeOfFiles(): long
+addSubNode(fileOrDir: FileSystemNode): void
+removeSubNode(fileOrDir: FileSystemNode): void

```
public abstract class FileSystemNode {
    protected String path;
    public FileSystemNode(String path) {
        this.path = path;
    }
    public abstract int countNumOfFiles();
    public abstract long countSizeOfFiles();
    public String getPath() {
        return path;
    }}
```

```
public class File extends FileSystemNode {
    public File(String path) {
        super(path);
    }
    @Override
    public int countNumOfFiles() {
        return 1;
    }
    @Override
    public long countSizeOfFiles() {
        java.io.File file = new java.io.File(path);
        if (!file.exists())
            return 0;
        return file.length();
    }}
```

```
public class Directory extends FileSystemNode {
    private List<FileSystemNode> subNodes = new ArrayList<FileSystemNode>();

    public Directory(String path) {
        super(path);
    }

    @Override
    public int countNumOfFiles() {
        int numOfFiles = 0;
        for (FileSystemNode fileOrDir : subNodes) {
            numOfFiles += fileOrDir.countNumOfFiles();
        }
        return numOfFiles;
    }

    @Override
    public long countSizeOfFiles() {
        long sizeofFiles = 0;
        for (FileSystemNode fileOrDir : subNodes) {
            sizeofFiles += fileOrDir.countSizeOfFiles();
        }
        return sizeofFiles;
    }
    ... ...
}
```

组合模式 – 优缺点

- 组合模式的优点
  - 可以清楚地定义分层次的复杂对象，表示对象的全部或部分层次，使得增加新构件也更容易
  - 客户调用简单，客户可以一致地使用组合结构或其中单个对象
  - 定义了包含叶子对象和容器对象的类层次结构，叶子对象可以被组合成更复杂的容器对象，而这个容器对象又可以被组合，这样不断递归下去，可以形成复杂的树形结构
  - 更容易在组合体内加入对象构件，客户不必因为加入了新的对象构件而更改原有代码

组合模式 – 优缺点

- 组合模式的缺点
  - 使设计变得更加抽象，对象的业务规则如果很复杂，则实现组合模式具有很大挑战性，而且不是所有的方法都与叶子对象子类有关联
  - 增加新构件时可能会产生一些问题，很难对容器中的构件类型进行限制

组合模式 – 【适用情形】

- 适用情形
  - 需要表示一个对象整体或部分层次，在具有整体和部分的层次结构中，希望通过一种方式忽略整体与部分的差异，可以一致地对待它们
  - 让客户能够忽略不同对象层次的变化，客户可以针对抽象构件编程，无须关心对象层次结构的细节
  - 对象的结构是动态的并且复杂程度不一样，但客户需要一致地处理它们

组合模式 – 在JDK中的应用

- Java JDK里的组合模式
  - java.awt.Container
    - add(Component)

```
public class Container
extends Component
```

  - java.util.Map
    - putAll(Map)

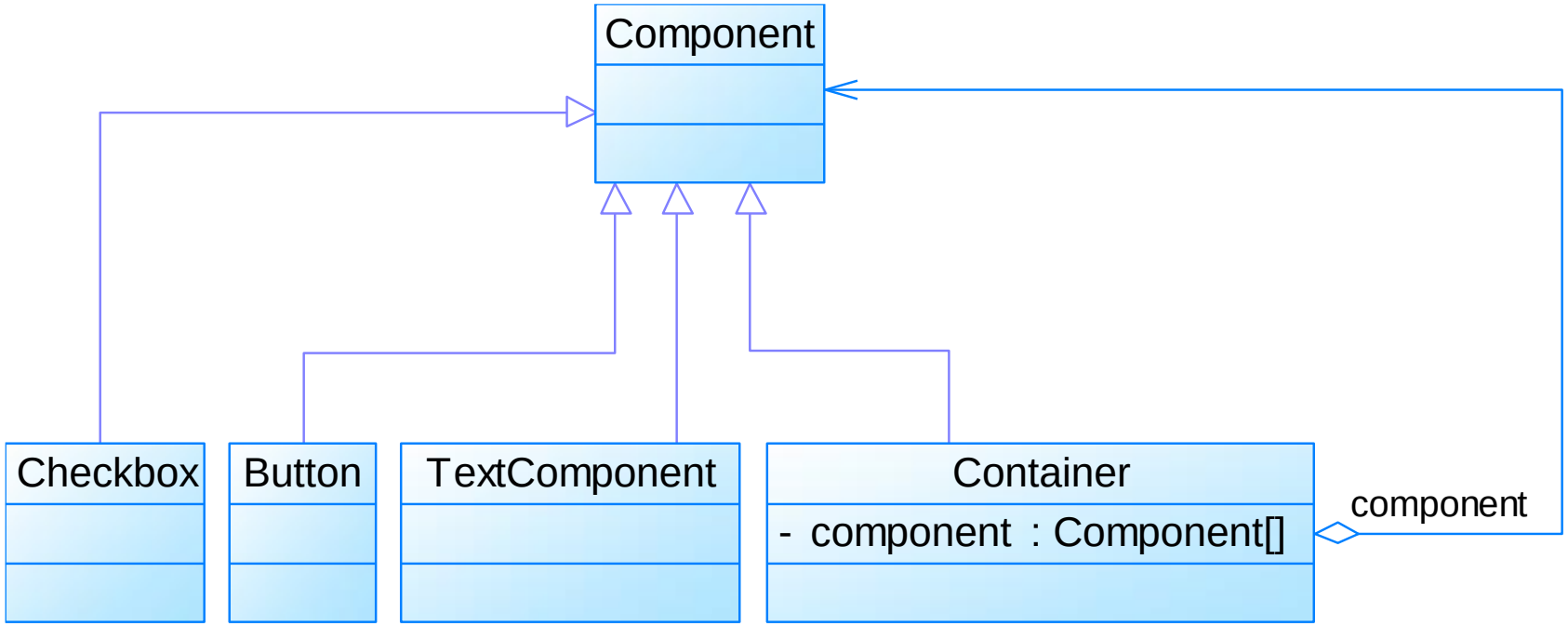
```
public interface Map<K,V>
```

  - java.util.Set
    - addAll(Collection)

```
public interface Set<E>
extends Collection<E>
```

组合模式 – 应用

- Java JDK里的组合模式



Thank you!