第7讲
OO程序的分析与设计原则

OO2019课程组

北京航空航天大学计算机学院

提纲

- 面向对象之“思想”什么
- 面向对象之“分析思想”
- 实现之前的设计
- 实现之后的设计检查
- 作业

面向对象之“思想”什么

- 面向对象思想(object-oriented thinking)是经常见到的词汇
- 什么是面向对象思想
  - 一种思维方式，以对象为视角的思维方式
  - 有哪些对象？
  - 对象做什么？
  - 对象之间有什么连接？
- 在不同阶段的面向对象思想
  - 分析阶段：理解和识别需求中的“对象”
  - 设计阶段：构造“对象”来实现需求
  - 实现阶段：使用程序语言来实现“对象”
  - 测试阶段：逐个检查“对象”的功能和性能，然后对“对象集成”进行测试

面向对象之“分析思想”

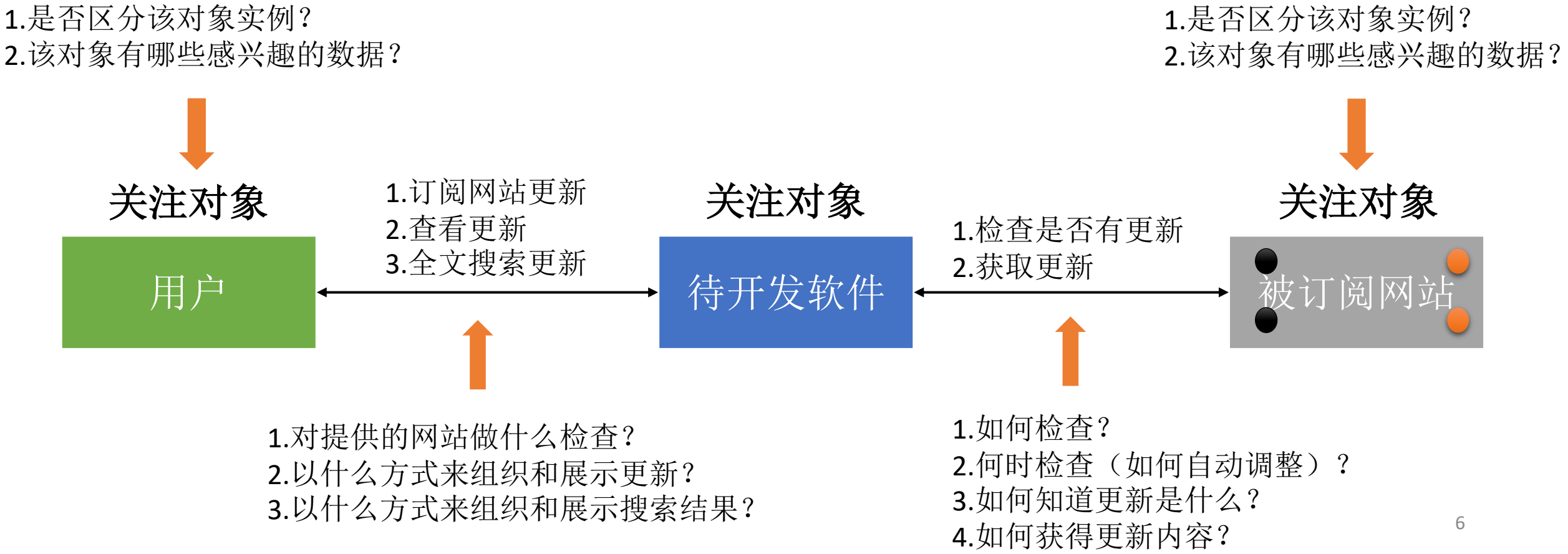
- 对象化思维来理解软件需求
  - 识别类-----数据、控制、设备...
  - 识别类的职责-----管理数据、实施控制/控制策略、输入输出处理
  - 基于类来分析软件功能-----多个类协同的方式来阐述功能
- 需求一般都是从用户视角来规定软件的功能和性能
  - 输入、输出
  - 平台
  - 数据
- 我们目前的作业要求在层次上相当于软件需求
  - 实际上比一般的软件需求要细致，介于软件需求和设计之间

面向对象之“分析思想”

- 例子
  - 新闻类、博客类网站每天会以不确定的频率发布新的消息
  - 用户难以知道什么时候有信息更新
  - 不能要求用户使用浏览器刷新网站页面来获得信息更新
  - 开发一个网站内容更新订阅系统，功能要求如下：
    - 能够根据用户需要订阅一个网站的内容更新
    - 能够自动获得网站内容的更新，包括主题、日期和信息片段(snippet)
    - *能够自适应网站内容更新的频度*
    - 能够对更新的主题、日期、信息片段进行有效管理
    - 提供对更新的全文搜索
    - 能够把来自不同网站的相同更新进行合并

面向对象之“分析思想”

- 1.交互关系分析：待开发的**软件**与**用户**、**外部环境**有哪些交互？

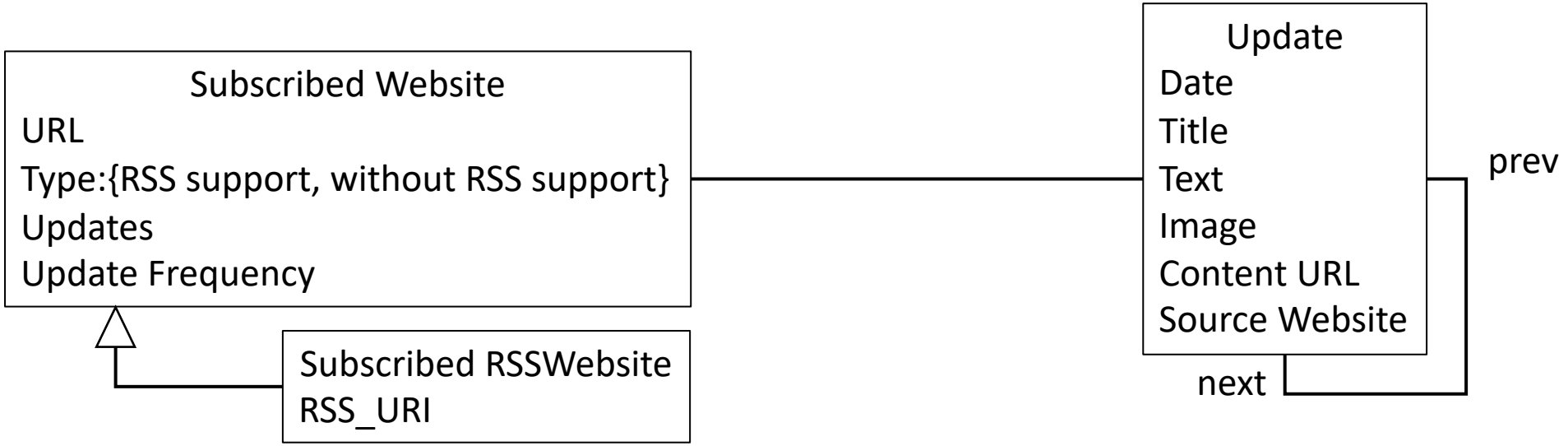


面向对象之“分析思想”

- 通过这种对象分析，确定了待开发软件的边界、对象之间的交互关系、对象交互的数据与时间特征
  - 用户交互的**数据特征**：提供待订阅网站、提供更新搜索词
  - 用户交互的**时间特征**：交互频度？交互持续时间？是否有多用户并发？
  - 被订阅网站交互的数据特征：网页内容(html)、更新列表文件(xml)、网页层次结构(html)
  - 被订阅网站交互的时间特征：更新频度、通讯速度
- 在对象分析过程中不仅理解了软件需求，更是细化了需求，并识别出潜在的问题
  - 如何找到网页中有更新？
    - 有更新列表文件的情况：按照更新列表来识别和提取更新
    - 无更新列表文件的情况：按照给定的页面来识别和提取更新

面向对象之“分析思想”

- 2.进入“待开发系统”对象内部进行分析
  - 运行平台：单机/Web服务器？
  - 管理哪些数据：被订阅网站、网站内容更新
  - 对数据的管理手段：维护订阅网站列表、管理更新、管理订阅网站与更新之间的关系、维护订阅网站的更新检查频度、检查不同订阅网站的更新是否相同、检查订阅网站是否有更新、提取订阅网站的更新

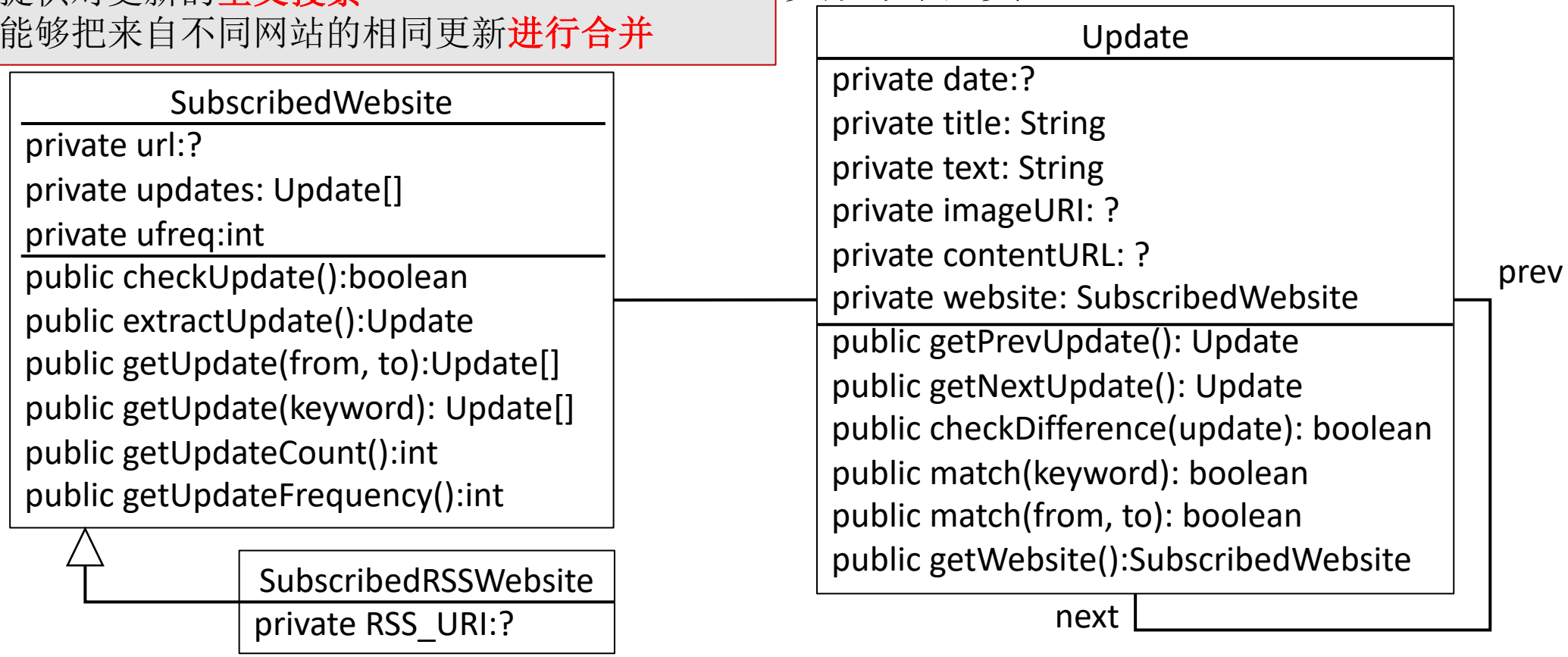


面向对象之“分析思想”

检查是不是所有的需求都能够得到满足(如何满足是设计问题)

- 能够根据用户需要**订阅一个网站内容更新**
  能够**自适应**网站内容更新的频度
  能够**获得网站内容的更新**
  能够**管理更新**的主题、日期、信息片段(snippet)
  提供对更新的**全文搜索**
  能够把来自不同网站的相同更新**进行合并**

进行分析
类及其职责



实现之前的设计

- 识别并发行为
- 增加额外的数据管理类
- 更好的刻画数据中的结构
- 简化类方法的职责
- 设计类之间的协同
- 空间与时间的平衡

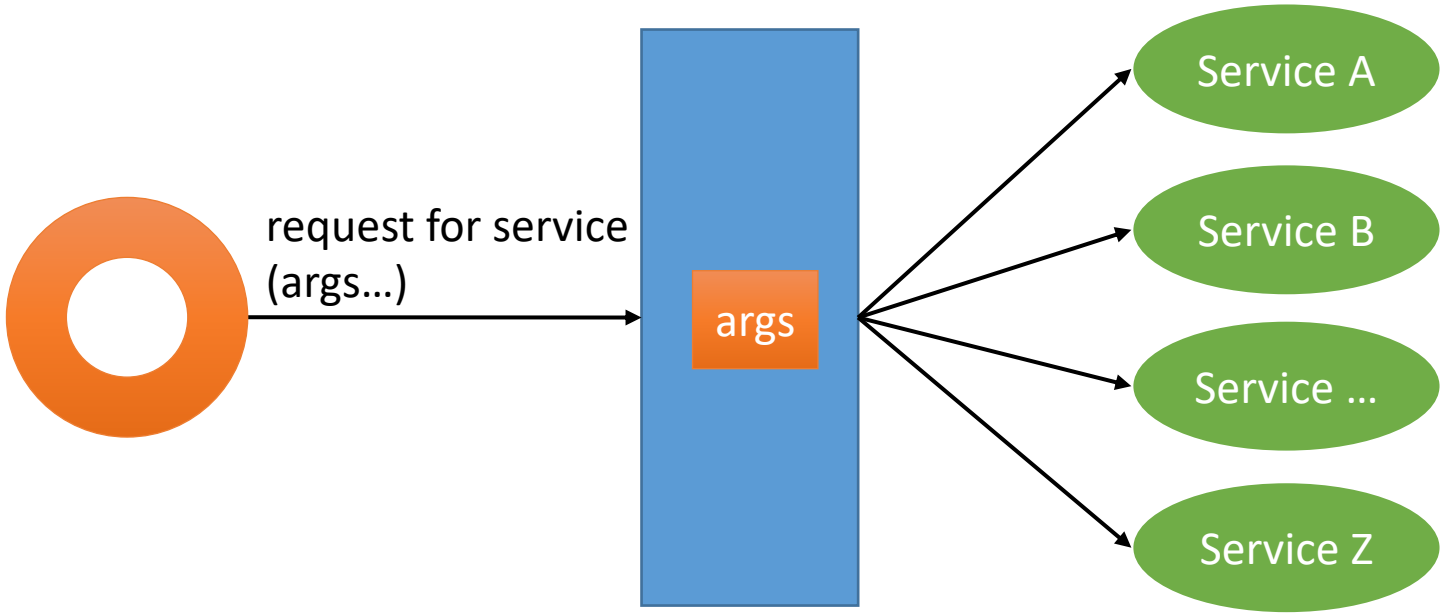
实现之前的设计

多线程与多种线程

- 识别并发行为
  - 待实现软件与外部多个对象进行相对独立的交互
    - 按照交互特征进行分类，一个类别对应一个线程设计
    - 订阅网站数量可以很多，一般相互独立
      - 有RSS支持的网站和无RSS支持的网页
      - 对应两类不同的线程设计：RSSWebSiteChecker、WebPageChecker
  - 待实现软件要处理的数据有显著的重复模式，且处理相对独立
    - Web系统的日志处理：日志记录着用户访问Web系统的行为，具有典型的模式；处理目标是提取用户行为、出现的问题、系统响应时间等，相对独立
    - 扫描一个规模较大的字符串数组看是否出现某个关键词：重复模式是不断在一个局部的数组中查找关键词的出现情况
    - 计算两个大矩阵的乘积：重复模式即为一个矩阵的行向量乘以另一矩阵的列向量

实现之前的设计

- 识别并发行为
  - 设想一个软件持续监听某网络端口以获得一种特定请求“request for service”，根据服务参数的不同，该软件需要进行不同的处理。



需要几类线程来支撑这个软件？

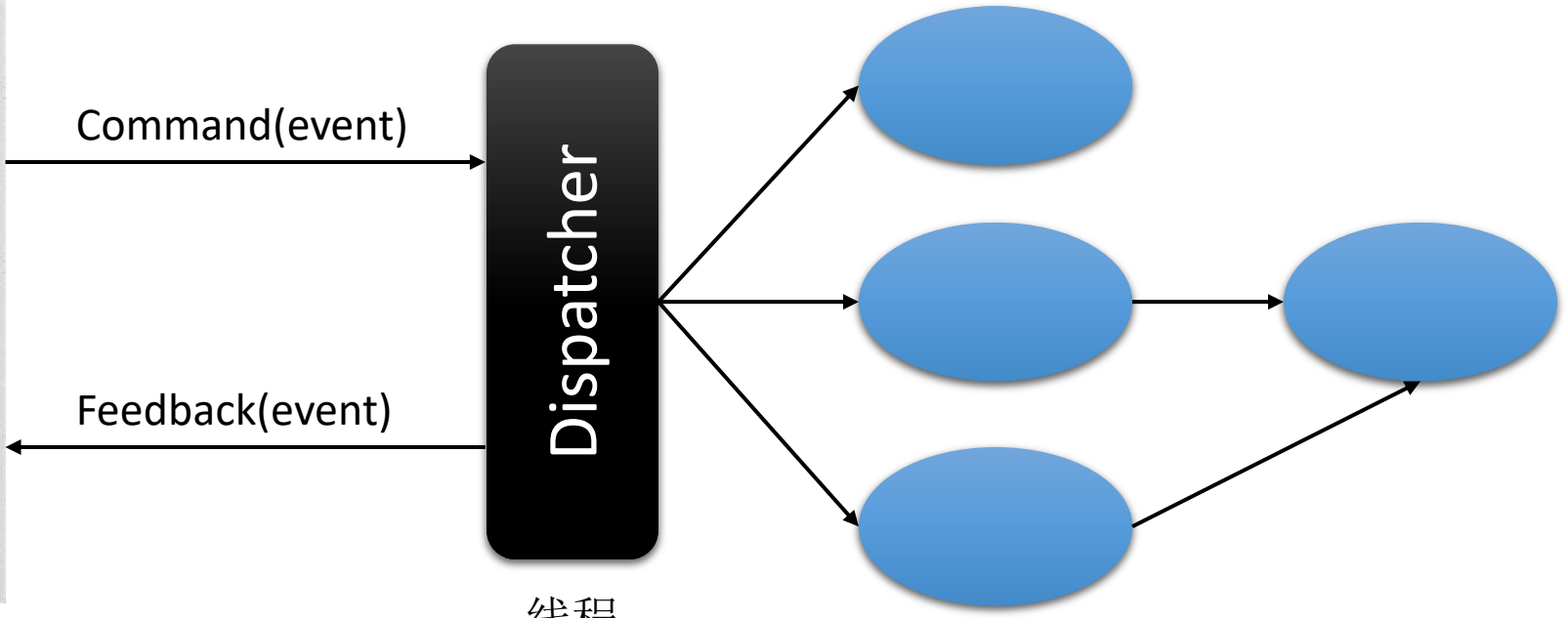
实现之前的设计

- 三类并发行为模式
  - 人机交互的并发
  - 多client请求处理的并发
  - 工作流处理的并发

人机交互中的并发

- Event dispatch
  - 确保界面响应的即时性，随时可以发出command，且在感兴趣的数据对象状态发生变化时即时在GUI上观察到相应的feedback

内置的线程控制机制

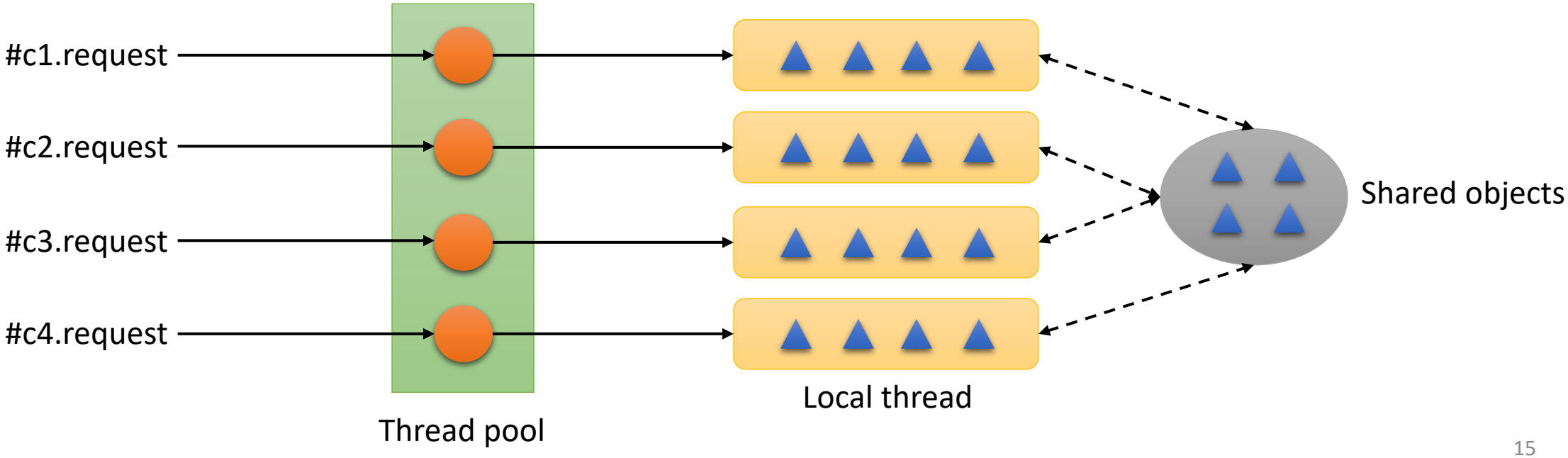


线程

数据对象

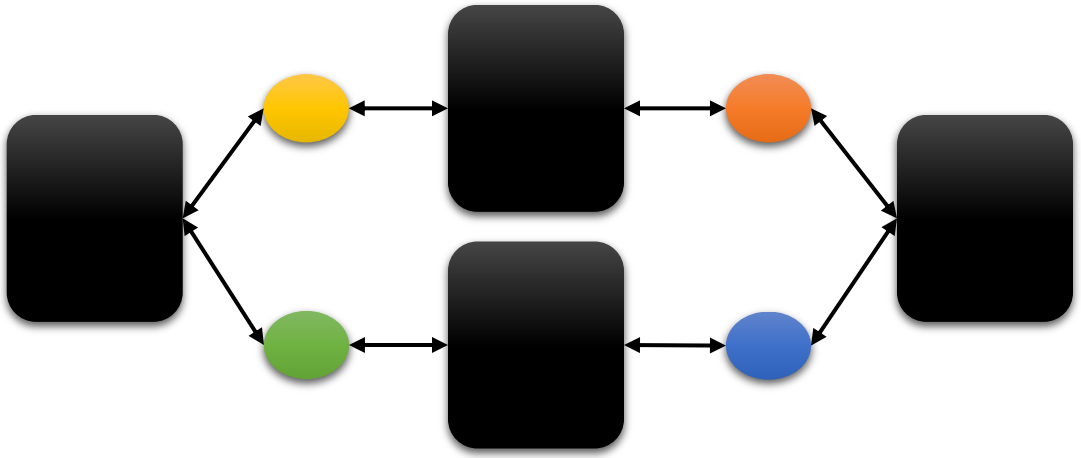
多Client请求处理的并发

- Isolate requests from different clients
  - 每个client请求的发出时机不同步
  - 每个client请求涉及的数据可能具有privacy要求
  - 隔离式处理可提高系统的吞吐率、可靠性和安全性(security)



工作流处理的并发

- 工作流Work flow
  - 多个任务之间相对独立，但在特定数据上形成依赖关系
  - 整体具有Pipeline特征
- Isolate producer and consumer
  - 通过共享数据访问控制来隔离依赖关系
  - 最复杂的一种并发处理模式



再谈同步控制锁

- synchronized使用的是JVM内置的锁机制，称为物理锁
  - 简洁、易用。大部分情况下都能解决问题
- 读-写冲突和写-写冲突是最突出的问题，如何避免这类冲突，同时又获得尽可能高的性能是一个重要问题
- Read Write Lock：逻辑锁
  - 记录正通过该逻辑锁的#readingReaders, #writingWriters, #waitingWriters
  - 控制规则
    - (writingWriters == 0) → 获得read lock
      - 如果要考虑写优先，则可以进一步对waitingWriters进行限制
    - (writingWriters == 0 && readingReaders == 0) → 获得write lock
  - Java.util.concurrent.locks.ReadWriteLock, Java.util.concurrent.locks.ReentrantReadWriteLock,

http://tutorials.jenkov.com/java-concurrency/read-write-locks.html

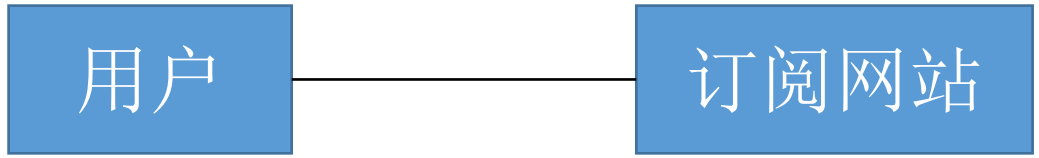
使用锁需要注意的问题

- 存在return时，锁可能没有释放
- Lock/unlock之间的语句如果抛出异常，也可能造成锁没有释放
- 如果想要一个方法成对的使用lock/unlock，可以使用finally
- 做好深入了解try/catch机制的准备

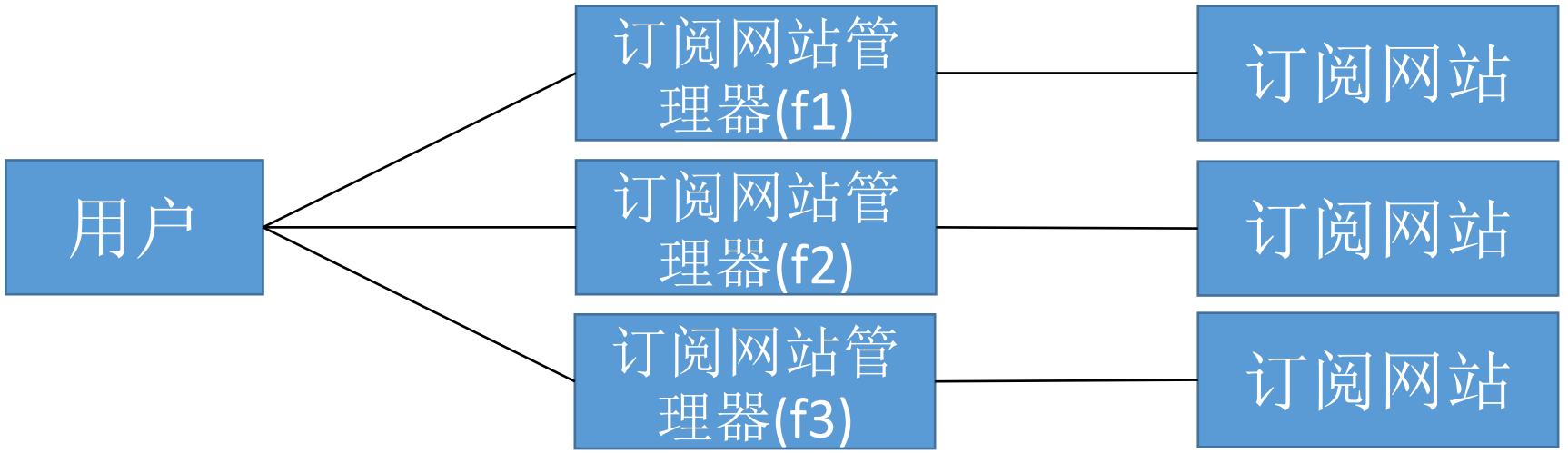
```
void method {
   lock();
   try {
      …
   }
   finally {
      unlock()
   }
}
```

实现之前的设计

- 增加额外的数据管理类
  - 数据管理并不简单是增加一个数组或容器的问题
  - 网站内容更新订阅系统中，如何管理用户订阅的网站？



如何对订阅网站进行内容更新检查？
如果是多用户怎么办？



实现之前的设计

- 增加额外的数据管理类
  - 需要对被管理对象的特征和行为进行分类，按照类别进行管理==》简化相应的管理机制和行为逻辑
  - 出租车呼叫系统如何管理出租车和乘客？
    - 出租车：位置、信用、公司
    - 乘客：VIP、普通会员和普通乘客

实现之前的设计

- 更好的刻画数据中的结构
  - 很多数据都可以使用字符串来表示，但是字符串并不一定都能描述数据固有的逻辑结构
    - 电话号码
    - URL地址
  - 如果软件对相应数据的处理不涉及数据的逻辑结构
    - 字符串可以满足功能要求，但是不利于将来扩展和分析
  - 如果软件对相应数据的处理涉及数据的逻辑结构
    - 字符串不能满足功能要求
  - 结构化～非结构化

实现之前的设计

- 简化类方法的职责
  - 尽量保持每个方法只做一件事情
  - 不同方法不可避免存在交叉的行为，或者一个方法正好可以做两件事情
  - SubscribedWebsite中谁负责更新网站检查频率？
    - 方案A：单独增加一个方法updateFrequency(int freq)
    - 方案B：在checkUpdate中来更新频率
    - 方案C：在extractUpdate中来更新频率

SubscribedWebsite
private url:? private updates: Update[] private ufreq:int
public checkUpdate():boolean public extractUpdate():Update public getUpdate(from, to):Update[] public getUpdate(keywork): Update[] public getUpdateCount():int public getUpdateFrequency():int

实现之前的设计

- 简化类方法的职责
  - 建议采用方案B：在checkUpdate中来更新频率
    - 首先SubscribedWebsite的管理已经按照其更新频率进行了分类
    - checkUpdate的执行具有简单的周期性
    - 如果连续*[三次]*发现未更新，则调整更新频率，同时把相应对象调整到别的管理类别中
    - 即简化了方法职责，又保证了效率，同时减少了反复检查导致的CPU浪费
    - 如果连续三次发现都已经更新了呢？
  - 从方法职责单一性角度，把频率计算和更新单独设计成一个私有方法，被checkUpdate调用

Gmail大致就是按照这个思路来动态调整对POP3邮箱新邮件的检查

SubscribedWebsite
private url:? private updates: Update[] private lastNhits: int private ufreq:int
public checkUpdate():boolean public extractUpdate():Update public getUpdate(from, to):Update[] public getUpdate(keywork): Update[] public getUpdateCount():int public getUpdateFrequency():int private calcFrequency():int

实现之前的设计

- 设计类之间的协同
  - 从上面所说的多个角度会增加新的类、方法和属性
  - 这个过程就是设计
  - 仅仅使用这些单视角的方法进行设计会导致整体性的丧失
    - 做事情多的类倾向于做更多的事情
    - 管理数据多的类倾向于管理更多的数据
  - 需要从协同角度开展整体性设计
    - 在已有类的基础上，针对软件功能(和一些核心类的方法)来设计协同
      - Sit-together to work
      - Peer-to-peer
      - Client/server

实现之前的设计

- 空间与时间的平衡
  - 完成需求功能设计之后，其实还可以做的更好
  - 网站更新订阅系统
    - 网站不仅内容会发生变化，结构也可能会发生变化
    - 非RSS网站的内容变化建立在网页快照的对比基础之上
    - 这会带来什么问题？
  - 通过缓存一些历史状态，可以加速处理，提高程序执行效率
    - 几乎所有的服务器设计都会使用cache机制

SubscribedRSSWebsite
private RSS_URI:?
private cachedSitePage: String

SubscribedWebsite
private url:?
private updates: Update[]
private lastNhits: int
private ufreq:int
private cachedSiteMap: Map
public checkUpdate():boolean
public extractUpdate():Update
public getUpdate(from, to):Update[]
public getUpdate(keywork): Update[]
public getUpdateCount():int
public getUpdateFrequency():int
private calcFrequency():int

实现之后的设计检查

- 经典的5个设计原则检查(SOLID)
  - SRP、OCP、LSP、ISP、DIP
- 我们还需要强调的几个设计原则检查

设计模式：用以解决特定（具有普遍性）问题的一种方案，如对象构造工厂、职责代理等

体系结构：软件模块被组织/集成为系统的结构，如MVC，Pipeline

设计原则：关于设计的整体要求和约束，通过满足设计原则来获得好的设计质量

SOLID之SRP

- Single Responsibility Principle
  - 每个类或方法都只有一个明确的职责
  - 类职责：使用多个方法，从多个方面来综合维护对象所管理的数据
  - 方法职责：从某个特定方面来维护对象的状态（更新、查询）

```
public class Elevator{
    //fields such as floor, status, …
    public void move(int dest_floor){
        //让电梯运动到目标楼层
    }

    public void Scan4TakingRequest (Queue q)
    {
        //扫描请求队列来寻找可以捎带的请求
    }
}
```

类/方法职责多，就意味着逻辑难以封闭，容易受到外部因素变化而变化，导致类/方法不稳定。

SOLID之OCP

Open/closed principle. In object-oriented programming, the **open/closed principle** states "software entities (classes, modules, functions, etc.) should be **open** for extension, but **closed** for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code.

Open/closed principle - Wikipedia
https://en.wikipedia.org/wiki/Open/closed_principle

Bertrand Meyer

- Open Close Principle
  - 无需修改已有实现(close)，而是通过扩展来增加新功能(open)
- 当扩展电梯系统支持ALS调度时
  - 改写Scheduler
  - 扩展Scheduler
- Employee类的display()方法
  - 普通员工：可以显示员工的基本信息和状态信息
  - 部门经理：把部门的其他所有员工也按照某种方式显示出来
  - 假设先实现了普通员工相应的功能，再来实现部门经理功能，怎么办？

SOLID之LSP

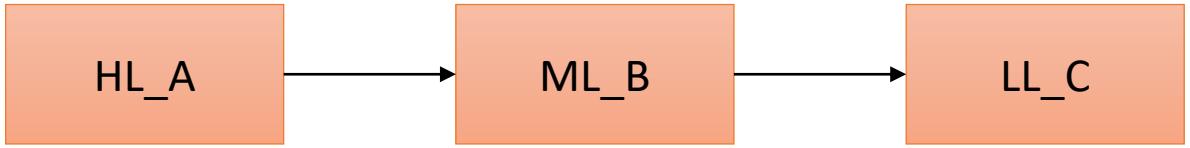
- Liskov Substitution Principle
  - 任何父类出现的地方都可以使用子类来代替，并不会导致使用相应类的程序出现错误。
    - BaseClass b = new BaseClass(…) → BaseClass b = new DerivedClass(…)
  - 子类虽然继承了父类的属性和方法，但往往也会增加一些属性和方法，可能会破坏父类的相关约束
  - 例：Queue和SortedQueue类
    - Queue类提供了一个getLastInElement()方法，即返回最近一次入队列的元素，其实现是返回队列尾部的元素
    - SortedQueue类则对队列中的元素进行排序，每次有新元素加入队列时，按照元素之间的大小关系插入到特定的位置
    - 此时调用SortedQueue的getLastInElement会怎么样？如何解决这个问题？

SOLID之ISP

- Interface Segregation Principle
  - 当一个类实现一个接口类的时候，必须要实现接口类中的所有接口，否则就是抽象类(abstract class)，不能实例化出对象
  - 软件设计经常需要设计接口类，来规范一些行为。避免往一个接口类中放过多的接口
  - 例：Payment是一个接口类，用来规范电子商务中的付款方式
    - 信用卡付款: 一组接口
    - 储蓄卡付款: 一组接口
    - 支付宝付款: 一组接口
  - 假设Payment同时把这三类付款方式都纳入作为接口
    - 你开设了一个店铺，要纳入一个平台必须要使用Payment接口
    - 每个商品都要实现这三个接口
    - 有什么问题？如何解决？

SOLID之DIP

DIP: Dependency Inversion Principle



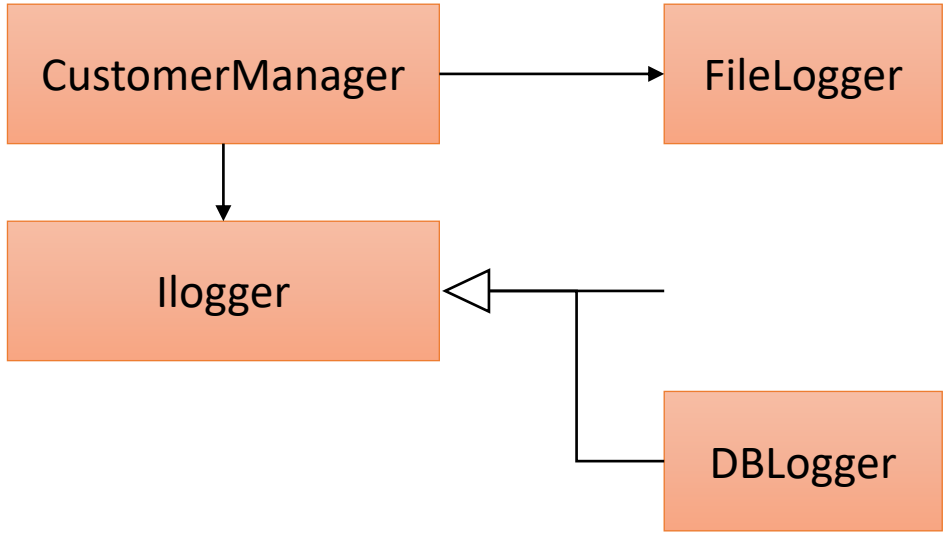

```
public class CustomerManager
{
     …field attributes here…
     public void Insert(Customer c)
     {
          try{
               //Insert logic
          }catch (Exception e){
              FileLogger f = new FileLogger();
              f.LogError(e);
          }
     }
}
public class FileLogger
{
    public void LogError(Exception e){//Log Error in a physical file }
}
```

CustomerManager依赖于Filelogger，
即把异常信息记录到具体存储文件中

在数据库存储环境中想要重用
CustomerManager类，怎么办？

SOLID之DIP

A. High-level modules should not depend on low-level modules. Both should depend on **abstractions**.
B. Abstractions should not depend on details. Details should depend on abstractions.




```
public class CustomerManager
{
     …field attributes here…
     private Ilogger mylogger;
     public CustomerManager(Ilogger logger){mylogger = logger;}
     public void Insert(Customer c)
     {
          try{
               //Insert logic
          }catch (Exception e){
             mylogger.LogError(e);
          }
     }
}
```

```
interface Ilogger
{
    public void LogError(Exception e);
}
public class FileLogger implements Ilogger
{
    public void LogError(Exception e)
    {//Log Error in a physical file }
}

public class DBLogger implements Ilogger
{
    public void LogError(Exception e)
    {//Log Error in a DB }
}
```

其他重要的设计原则

- 层次化抽象原则，按照问题域逻辑关系来识别类；
- 责任均衡分配原则，避免出现God类和Idiot类；
- 局部化原则，类之间不要冗余存储相同的数据，方法之间不能够出现控制耦合；
- 完整性原则，一个类需要提供针对相应数据进行处理的完整方法集。完整是个相对概念，一般来说是相对于问题域需求。
- 重用原则（共性抽取原则），把不同类之间具有的共性数据或处理抽象成继承关系，避免冗余；
- 显式表达原则，显式表达所有想要表达的数据或逻辑，不使用数组存储位置或者常量来隐含表示某个特定状态或数据；
- 信任原则，一个方法被调用时，调用者需要检查和确保方法的基本要求能够被满足，获得调用结果后需要按照约定的多种情况分别进行处理；
- 懂我原则，所有类、对象、变量、方法等的命名做到“顾名思义”。

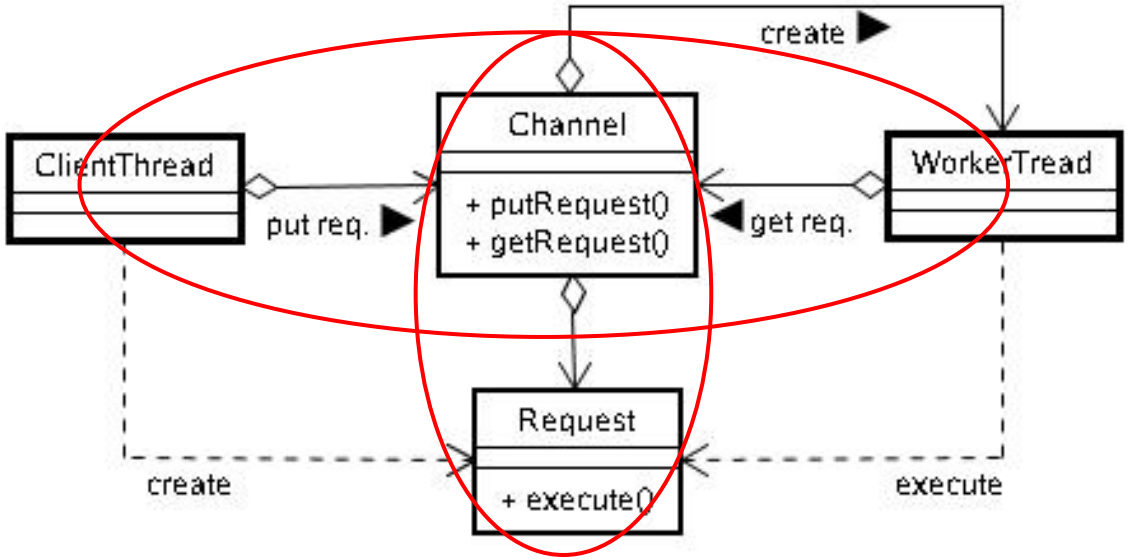
Worker Thread模式

- 在一个车间里，有很多工人负责组装产品
- 不同的客户会将很多装有产品部件的箱子送到车间里来
- 工人拿到客户的产品部件箱子，按里面的说明书组装成产品
- 当一个箱子组装完以后，工人就会组装下一个
- 如果全部组装完成，则工人进入等待状态

参见《图解Java多线程设计模式》

类的一览表

类名	说明
Main	测试程序行为的类
ClientThread	表示发出工作请求的线程的类
Request	表示工作请求的类
Channel	接受工作请求并将工作请求交给工人线程的类
WorkerThread	表示工人线程的类



ClientThread-Channel-WorkerThread:
Producer-Consumer pattern

Channel-Request:
command pattern

Main类

```
public class Main {
    /**
     * 会创建一个雇佣了五个工人线程的Channel实例，并将其共享给三个ClientThread实例
    */
    public static void main(String[] args) {
        Channel channel = new Channel(5); //工人线程的个数
        channel.startWorkers();
        new ClientThread(“Alice", channel).start();
        new ClientThread(“Bob", channel).start();
        new ClientThread(“Chris", channel).start();
    }
}
```

- 有三个客户，分别是”Alice”, “Bob”, “Chris”，客户会不断提交工作请求
- new Channel(5)的5，是创建的工人线程个数
- channel.startWorkers()是将工人线程挨个启动

Channel类

```
public class Channel {
   private static final int MAX_REQUEST = 100;
   private final Request[] requestQueue;
   private final WorkerThread[] threadPool;
   private int tail, head, count;

   public Channel(int threads) {
      this.requestQueue = new Request[MAX_REQUEST];
      head = 0; tail = 0; count = 0;
      threadPool = new WorkerThread[threads];
      for (int i = 0; i < threadPool.length; i++) {
         threadPool[i] = new WorkerThread("Worker-" + i, this);
      }
   }

   public void startWorkers() {
      for (int i = 0; i < threadPool.length; i++) {
         threadPool[i].start();
      }
   }
```

- 创建了要求数量的工人线程，并启动
- put/take Request的时候，什么是需要保护的竞争资源？

```
   public synchronized void putRequest(Request request) {
      while (count >= requestQueue.length) {
         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      requestQueue[tail] = request;
      tail = (tail + 1) % requestQueue.length;
      count++;
      notifyAll();
   }

   public synchronized Request takeRequest() {
      while (count <= 0) {
         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      Request request = requestQueue[head];
      head = (head + 1) % requestQueue.length;
      count --;
      notifyAll();
      return request;
   }
}
```

Request类

```
public class Request {
    private final String name;
    private final int number;
    private static final Random random = new Random();

    public Request(String name, int number) {
        this.name = name;
        this.number = number;
    }

    public void execute() {
        System.out.println(Thread.currentThread().getName() + " executes " + this);
        try {
            Thread.sleep(random.nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public String toString() {
        return "[ Request from " + name + " No. " + number + " ]";
    }
}
```

- 是工作请求类
- 有一个execute方法，每一个请求可能要做的东西都不一样，工人需要的是按说明书来操作，因此，请求需要有一个execute方法来让工作按图操作
- 请求提供操作手册的好处是：当我们建立Request类的子类，并将其传递给Channel，WorkerThread也能够正确的处理。执行工作所需要的所有信息，都定义在Request参与者里。所以即使建立出多态的Request参与者，增加工作的种类，Channel参与者和Worker参与者都不需要进行任何修改。

ClientThread类

```
public class ClientThread extends Thread {
    private final Channel channel;
    private static final Random random = new Random();

    public ClientThread(String name, Channel channel) {
        super(name);
        this.channel = channel;
    }

    public void run() {
        try {
            for (int i = 0; true; i++) {
                Request request = new Request(getName(), i);
                channel.putRequest(request);
                Thread.sleep(random.nextInt(1000));
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- 委托者角色
- 不断的向channel去提交组装申请
- 这里channel是否能使用单例模式来进行获取？

WorkThread类

```
public class WorkerThread extends Thread {
   private final Channel channel;

   public WorkerThread(String name, Channel channel) {
      super(name);
      this.channel = channel;
   }

   public void run() {
      while (true) {
         Request request = channel.takeRequest();
         request.execute();
      }
   }
}
```

- Worker角色从Channel中获取Request，并进行工作，当一项工作完成后，它会继续去获取另外的Request
- 这里channel是否能使用单例模式来进行获取？

类名	类比说明
Main	主控类
ClientThread	请求parsing线程
Request	乘客请求及其执行**(需要改造)**
Channel	请求调度，**是否需要多个？**
WorkerThread	电梯线程

Request的execute方法如何改造才可适应于电梯系统？

作业

- 本次作业是多部多线程智能电梯
  - 继续上周作业进行增量扩展
  - 三部电梯：运行速度不同、能够停靠的楼层有差异（诸如高区电梯和低区电梯）、载客人数限制不同
    - 进一步细化电梯运行状态
  - 乘客请求可能无法直达，拆分成几个（换乘）？如何拆分（本质还是一个调度问题）？
    - 是不是一层调度有点捉急？
  - 调度的优化不是本次作业训练的目的和要点。**重点在于请求调度和请求执行之间的分离（参考worker thread模式的设计思想）**