《面向对象设计与构造》
Lec2-类的设计

2023

OO课程组

北京航空航天大学

主要内容

- 类的识别与设计
- 属性与方法的设计
- 迭代开发下的属性与方法设计
- 对象的运行时特性
- 如何开展有效测试
- 作业分析与建议

类的识别与设计

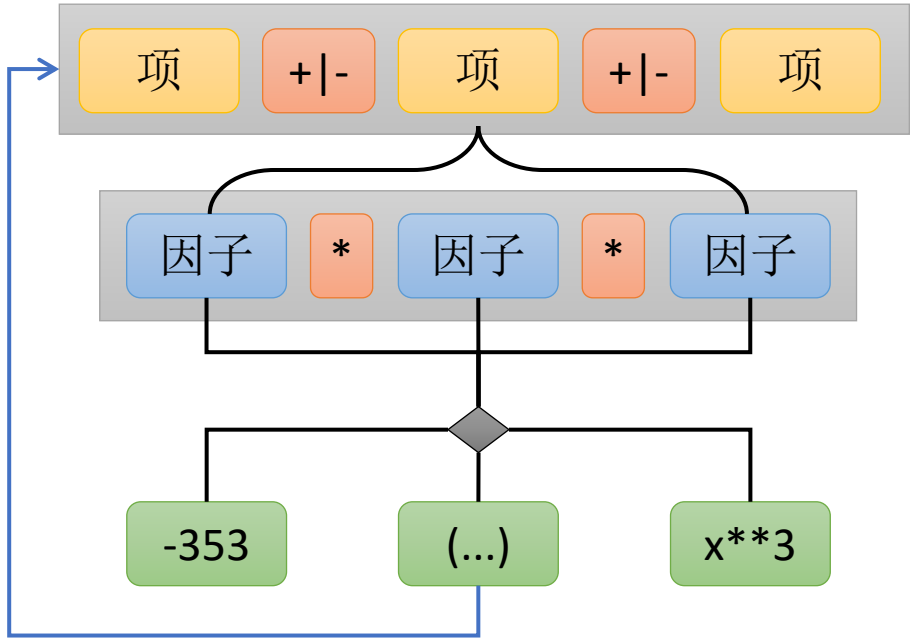
- 类是面向对象程序的**编程单位**
- 拿到一份指导书（需求）后
  - 理解和梳理功能要求-->功能结构
  - 理解和梳理数据要求-->数据结构
  - 按照功能和数据的关系整理成类
- 第一次作业
  - 功能：字符串变换，展开括号和必要的同类项合并
  - 数据：变元、表达式、复合结构

类的识别与设计

- 功能结构分析
  - 展开意味着计算
    - 最简单展开：+(...)+ -->直接去括号即可
    - 简单乘法展开：m*(...) -->对括号内结构进行顺次**遍历和变化**
    - 复杂乘法展开：(...)*(...) -->双**遍历和变化**
    - 幂展开：(...)**m -->m次**遍历和变化**
- 如果括号内数据的结构不定，功能的结构也无法设计

类的识别与设计

- 数据结构分析
  - 表达式由项组成，项之间通过加减运算符连接
  - 项由因子组成，因子之间通过乘运算符连接
  - 因子有三种
    - 常数因子：带符号整数
    - 表达式因子：复合结构 (表达式)
    - 变量因子：x**m
  - 三种结构
    - 顺序结构
    - 层次结构
    - 嵌套结构



类的识别与设计

- 表达式类(Expression)
  - 管理顺次相连的项对象
  - 管理作用于相邻项对象的加或减操作符
- 项类(Item)
  - 管理顺次相连的因子对象
- 因子类(Factor)
  - 常量因子(ConsFactor)：管理一个常量（带符号）
  - 变量因子(VarFactor)：管理一个变量、系数和幂
  - 表达式因子(ExprFactor)：引用到一个表达式对象

类的识别与设计

- 再看展开计算
  - 简单乘法展开
    - 针对表达式中的每个项乘以一个因子
  - 复杂乘法展开
    - 两个表达式中的项两两相乘
  - 幂展开
    - 多次应用乘法展开
- 展开之后的表达式中每一项
  - 表示形式可统一：**系数*变量**幂**
- 合并计算
  - 按照变量的幂来搜索同类项，合并系数

使用什么容器来管理（快速检索）？

类的三种主要角色

- 关注于输入输出处理的类
  - 输入扫描
    - 基于**层次化正则**
    - 基于**递归下降**
    - 结果：得到一系列因子（字符串）和连接操作符(字符串)
  - 格式化输出
    - 连接操作符的表示不必冗余存储
    - 必要的输出格式设置
    - 可配置

类的三种主要角色

- 关注于对象管理的类
  - 对象管理类提供统一的对象构造方法
    - 以切分后的字符串（或者解析后的数值）为输入，自动调用相关的类来构造对象
      - ConsFactor, ExprFactor, VarFactor
    - 通过Item类来管理这些因子对象，形成因子对象容器
    - 通过Expression类来管理项对象，形成项对象容器
  - 何时创建ExprFactor对象？
    - 创建时对输入字符串的扫描处理与输入输出类的扫描处理是否有区别？
  - 哪个类需要提供展开功能？
  - 哪些类需要提供合并功能？

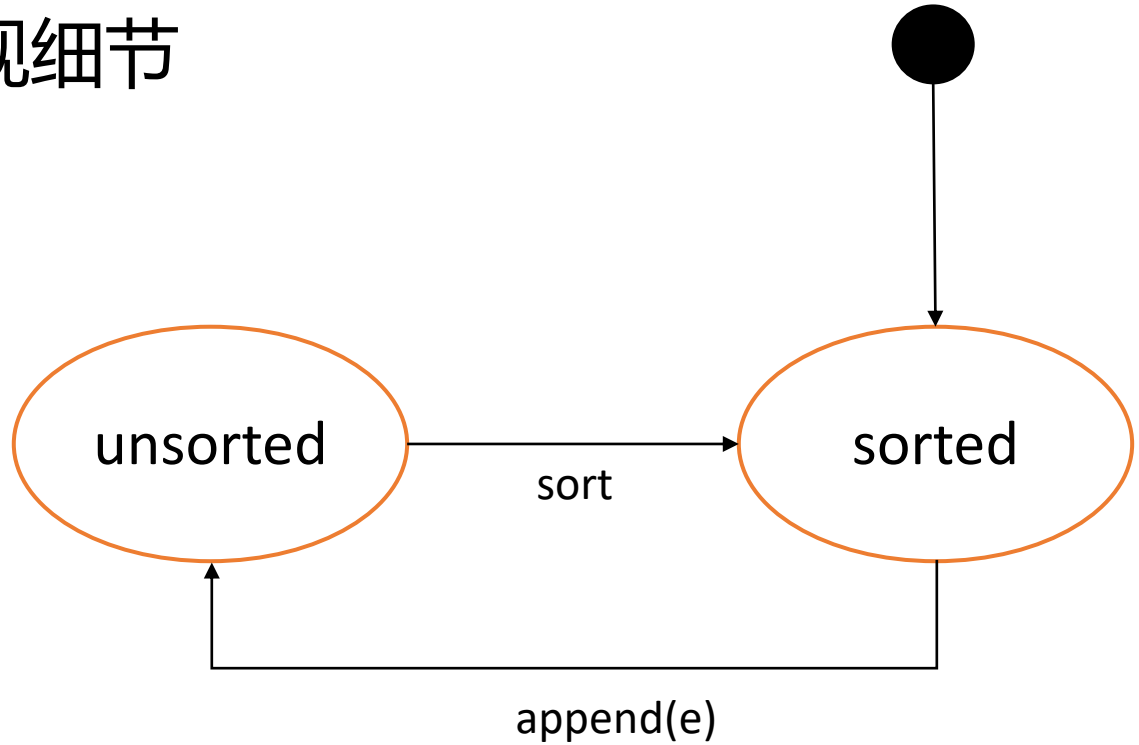
类的三种主要角色

- 关注于流程控制的类
  - 主控类
    - 提供main入口，实现主业务流程（只展开顶层逻辑），由其他类实现下一层次业务
  - 协控类
    - 与主控类协同，侧重主业务流程中的某个步骤，对下一层次业务处理进行封装
- 以第一次作业为例
  - 主控类
    - 创建“输入输出”对象、“对象管理”对象
    - 调用“输入输出”对象的“输入扫描与识别”方法
      - **解析输入，构造对象（并加入到“对象管理”对象）**
    - 调用“对象管理”对象的展开方法
    - 调用“对象管理”对象的合并方法
    - 调用“输入输出”对象的结果输出方法

属性与方法的设计

- 属性与方法构成了类的设计与实现细节
  - 属性定义对象状态
  - 方法定义作用于对象状态上的行为

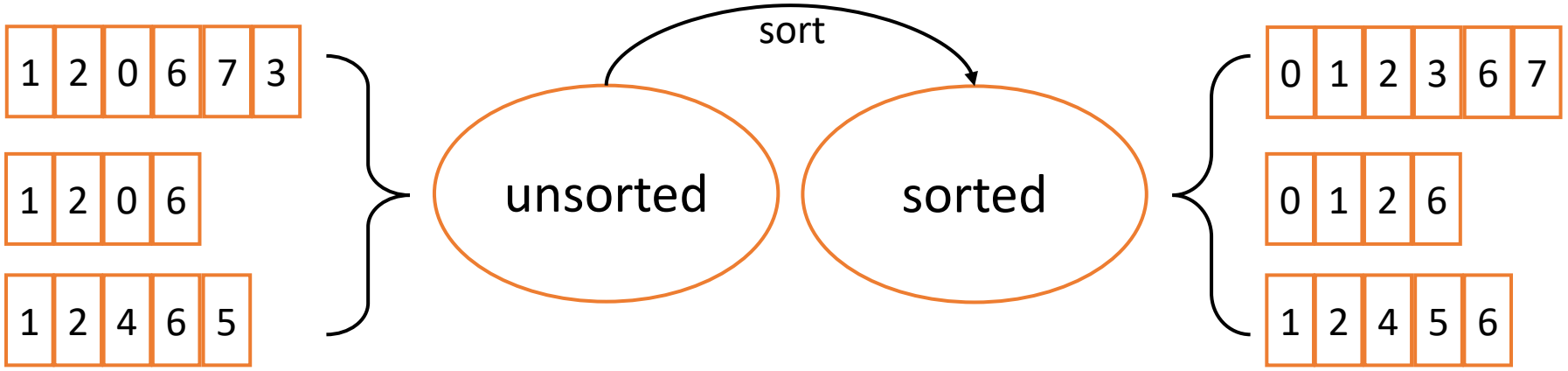
```
public class SortableArray{
    private Vector els;
    public void append(int e){els.add(e);}
    public void sort(){…}
}
```



(1)我们关心这个类的什么状态，隐藏了哪些内部状态？
(2)append是否一定会改变状态？
(3)如果append不改变状态，这个状态图如何调整？

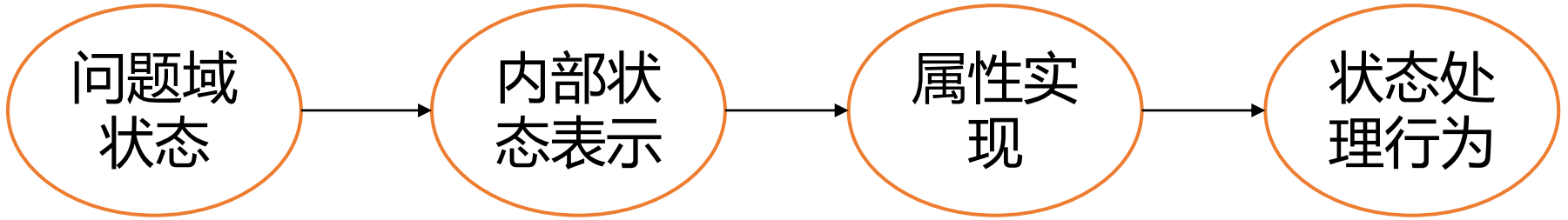
属性与方法的设计

- 对象状态
  - 内部状态 (memory image → internal state)
    - els中的所有元素
    - els中元素的顺序关系
  - 外部状态 (internal state → external state)
    - unsorted：存在i<j<k, (els[i] - els[j])*(els[j] - els[k]) <0
    - sorted: 任意i<j<k, (els[i] - els[j])*(els[j] - els[k]) >= 0



属性与方法的设计

- 依据需求中描述的问题域特征来定义状态
  - 图书馆系统中读者类的状态
  - 因子类的状态
- 针对对象状态，定义表示相应状态的属性和进行相应处理的方法
  - 有可能需要针对不同类别对象分别表示其状态
    - 常量因子类、变量因子类、表达式因子类



属性与方法的设计

- 一个对象该有哪些属性：依从性问题
  - 原则1：逻辑依从
    - 类的属性应该在问题域层次可见（依从问题域层次状态的表示需要）
      - 课程类、学生类
      - 表达式类、项类、因子类
  - 原则2：计算效率依从
    - 用于提高某些方法的计算效率，常常是推导属性(derived)
      - 数组长度、数组最大值、最小值
      - 幂展开，硬编码其展开公式，记录展开后各项的系数和幂

属性与方法的设计

- 针对给定类的每个属性进行检查
  - 如果去掉，该类所要存储和管理的数据是否完整？
  - 如果去掉，该类的行为受到什么影响？
- 实践中，很容易把应该属于其他类的属性放到当前类中
  - 如图书馆系统的读者类中很容易出现其借阅书的信息
  - 往往意味着这里存在类之间的**协作**
- 如果多个类在逻辑上涉及相同的数据怎么办？
  - 网络论坛系统，帖子(Post)类、帖子阅读类(PostRead)、回复通知消息类(Message)
  - 网络叫车系统(“嘀嘀”)，出租车调度类与出租车类

属性与方法的设计

- Case study
  - 学生成绩管理系统，功能包括选课、填报成绩、查询成绩、统计学分。
  - 根据这四个问题来整理Student和Course的属性

```
public class Student {
    private String stuID;
    private String stuName;
    private StuKind kind;
    private float totalcredits;
    private Course list[];
}
```

如何了解哪些同学选了某门课？

如何描述一个同学重修某门课？

是否需要了解一个同学什么时间修了某门课？

成绩作为Course的属性是否合适？

```
public class Course{
    private String courseID;
    private String courseName;
    private CourseKind kind;
    private float credit; //学分
    private float mark; //成绩
}
```

属性与方法的设计

- Case study
  - 学生成绩管理系统，功能包括选课、填报成绩、查询成绩、统计学分。
  - 请根据这四个问题来整理Student和Course的属性

```
public class Student {
    private String stuID;
    private String stuName;
    private StuKind kind;
    private float totalcredits;
    private CourseSelection list[];
}
```

```
public class CourseSelection{
    private Student student;
    private Course course;
    private Semester sem;
    private boolean reselection;
    private float mark; //成绩
    private float credit; //获得的学分
}
```

```
public class Course{
    private String courseID;
    private String courseName;
    private CourseKind kind;
    private float credit; //学分
    private CourseSelection list[];
}
```

学生如何查看有哪些课可选？

CourseSelection list定义为普通数组还是容器对象？

CourseSelection中的mark能否解决补考成绩问题？

属性与方法的设计

- 一般而言，方法可归纳为如下四种
  - 构造方法
    - 设置属性的初始值，即设置对象初始状态
  - 对象生成方法
    - 基于已有数据构造相关对象
  - 状态查询方法
    - 返回内部状态（即**属性值**）
    - 返回外部状态（执行内部状态到外部状态的转化）
  - 状态改变方法
    - 针对功能要求改变内部状态

属性与方法的设计

- 构造方法
  - 无参数构造方法，把对象初始化为“基准”状态
    - 把原子类型变量设置为默认初值，如0，false等
    - 设置对象变量（如果无法构造，则设为null）
    - 设置容器类变量（一般初始化为empty状态）
  - 有参数构造方法，按需构造对象的初始状态
- 对象生成方法
  - 随时可调用来生成所需的对象
  - 使用当前对象(this)所管理的数据和参数传递的数据
    - 如工厂类提供的方法
  - 对象管理类提供的对象构造封装

属性与方法的设计

- 状态查询方法
  - 盲目为每个属性添加一个get和set方法是没有意义的
  - 这类方法容易导致对象共享
    - 直接返回对象所管理的一个对象引用属性
    - 直接返回对象方法中创建的局部对象引用
  - 有些属性的取值不能允许外部访问
    - 账户对象的密码
  - 有些属性不能允许外部设置
    - 如电梯运行方向

属性与方法的设计

- 状态查询方法+状态改变方法
  - 启发式规则1：信息专家
    - 拥有相应属性的类就是相关信息的专家
    - 其他类需要向这个“专家”咨询什么？
    - “专家”需要对外界请求做哪些状态更新？
    - 例：学生成绩管理系统中的Student类

其他类需要它
(1)选一门课
(2)退选一门课
(3)登记一门课的成绩

其他类需要了解
(1)一个学生是否选了某门课
(2)一个学生的类别（本科生、研究生、…）
(3)学生的姓名
(4)学生总的学分
(5)学生选的某门课的成绩

```
public class Student {
    public boolean courseSelected(Course c){…}
    public StuKind getStuKind(){…}
    public String getName(){…}
    public float getTotalCredit(){…}
    public float getCourseMark(Course c){…}
    public boolean selectCourse(Course c){…}
    public boolean unselectCourse(Course c){…}
    public boolean recordMark(Course c, float m){…}
}
```

属性与方法的设计

- 状态查询方法+状态改变方法
  - 启发式规则2：控制专家
  - 一个类被赋予了拥有响应**系统事件**的能力----控制专家
  - 系统事件有哪些？处理这些事件需要哪些信息？
- 例：教务系统的课程管理
  - 有些系统层的事件需通知每个Student对象
  - 一门课因选课人数不足被取消
  - 预选的一门课，不幸抽签没中
  - 一门课的上课时间或地点调整

```
public class Student {
    public boolean courseSelected(Course c){…}
    public boolean selectCourse(Course c){…}
    public boolean unselectCourse(Course c){…}
    public void notifyLecChange(int courseID, int week,…){…}
    public void notifyCourseSelection(int courseID){…}
    public void courseCancellation(int courseID, int semster){…}
}
```

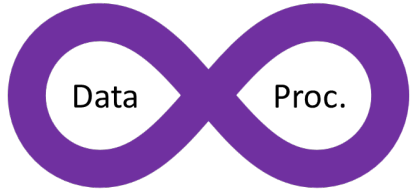
属性与方法的设计

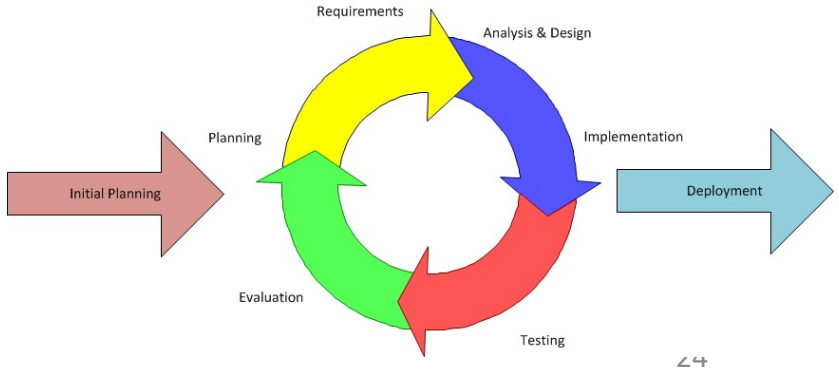
- 当为类设计一个方法时，要明确
  - 方法运行后达到的效果
  - 方法成功运行所依赖的前提条件
  - 方法运行时需要访问哪些数据
  - 方法运行时会改变哪些数据

```
public class Student {
    public boolean courseSelected(Course c){…}
    public StuKind getStuKind(){…}
    public String getName(){…}
    public float getTotalCredit(){…}
    public float getCourseMark(Course c){…}
    public boolean selectCourse(Course c){…}
    public boolean unselectCourse(Course c){…}
    public void notifyLecChange(int courseID, int week,…){…}
    public void notifyCourseSelection(int coursed, bool re){…}
    public void courseCancellation(int coursed, int semster){…}
    public boolean recordMark(Course c, float m){…}
}
```

迭代开发下的属性与方法设计

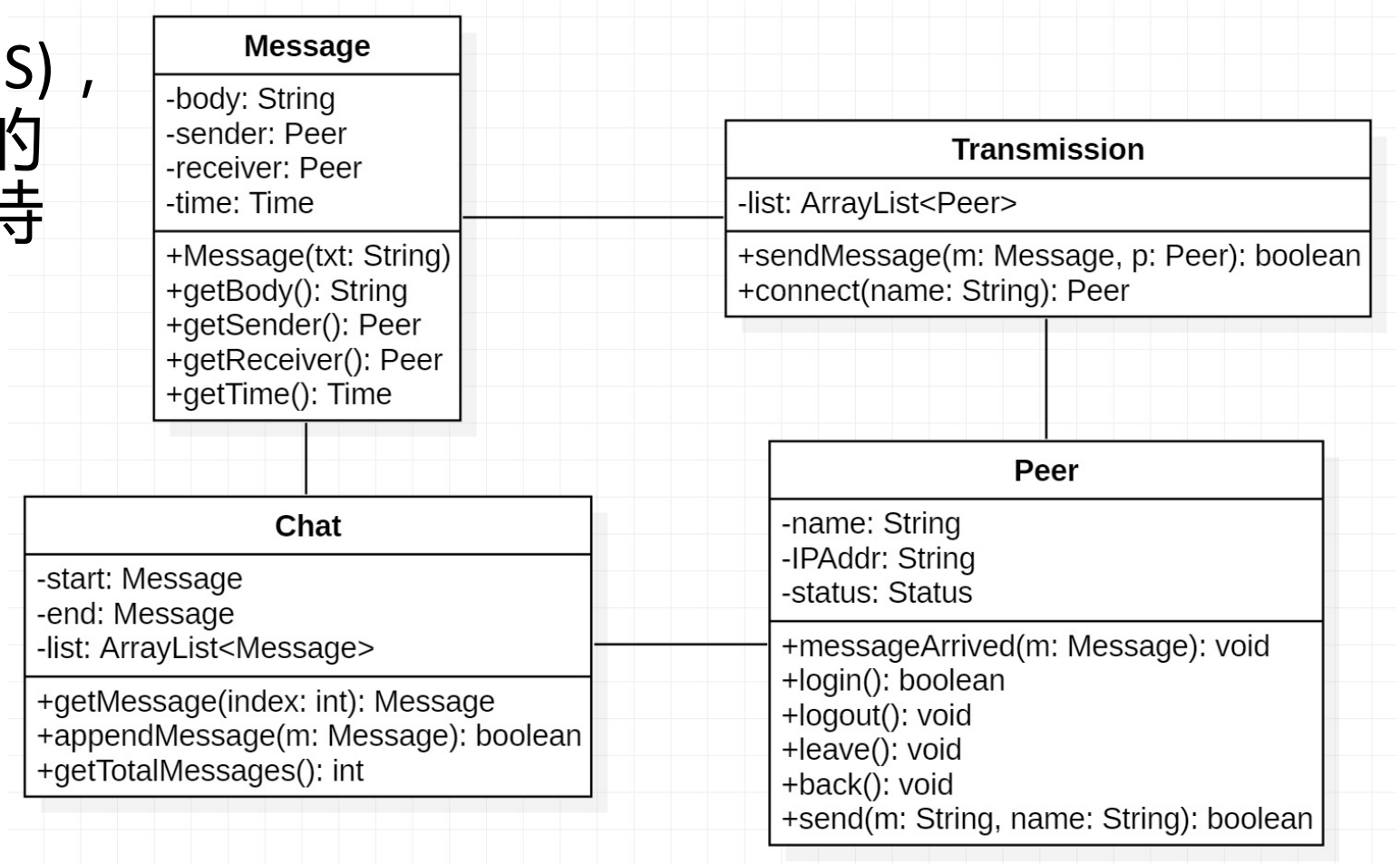
- 现实中很多软件的开发都采用迭代方法
  - 逐步安排开发，可以有效管理每次迭代的计划
  - 把还没有分析清楚的功能向后安排，避免需求的频繁变动
- 迭代开发对设计提出了较高要求
  - 避免功能一变，软件设计结构也跟着大变化
- **OO课采用迭代式作业设置**
- 迭代开发中的变化方式
  - 数据变化：要求处理更多类别的数据
  - 处理变化：要求提供更多的处理方式





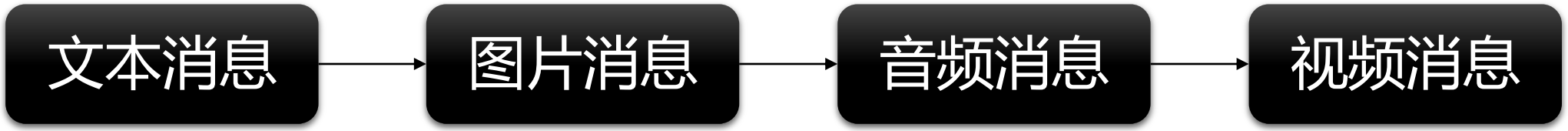
迭代开发下的属性与方法设计-Case Study

- 简易即时聊天系统(IMS)，当前版本支持点对点的单人间聊天，且只支持文本形态的聊天消息



迭代开发下的属性与方法设计-Case Study

- 数据迭代维度
  - 四个迭代版本



- 假设现在进行迭代版本2开发
- 需要对四个类的属性和方法做哪些调整？
  - 是否需要增加新的类？

迭代开发下的属性与方法设计-Case Study

- Message类
  - 增加属性来管理图片
  - 增加操作来提取图片
  - 增加构造方法来构造图片消息
- Peer类
  - 增加新的send操作，支持发送图片并构造消息
- Chat类
  - 不需要变化
- Transmission类
  - 不需要变化

任何一条Message必须同时管理图片和文本！

如果保持Message不变，引入一个PicMessage类呢？

迭代开发下的属性与方法设计-Case Study

- 功能处理迭代维度
  - 三个迭代版本

- 假设现在进行迭代版本2开发
- 需要对四个类的属性和方法做哪些调整？
  - 是否需要增加新的类？

迭代开发下的属性与方法设计-Case Study

- Message类
  - 不需要变化
- Peer类
  - 增加方法来进行消息广播发送（同时发给多个人）
  - 增加属性来管理与多个人的Chat记录
- Chat类
  - 不需要变化
- Transmission类
  - 增加方法来支持发送广播消息

迭代开发下的属性与方法设计-Case Study

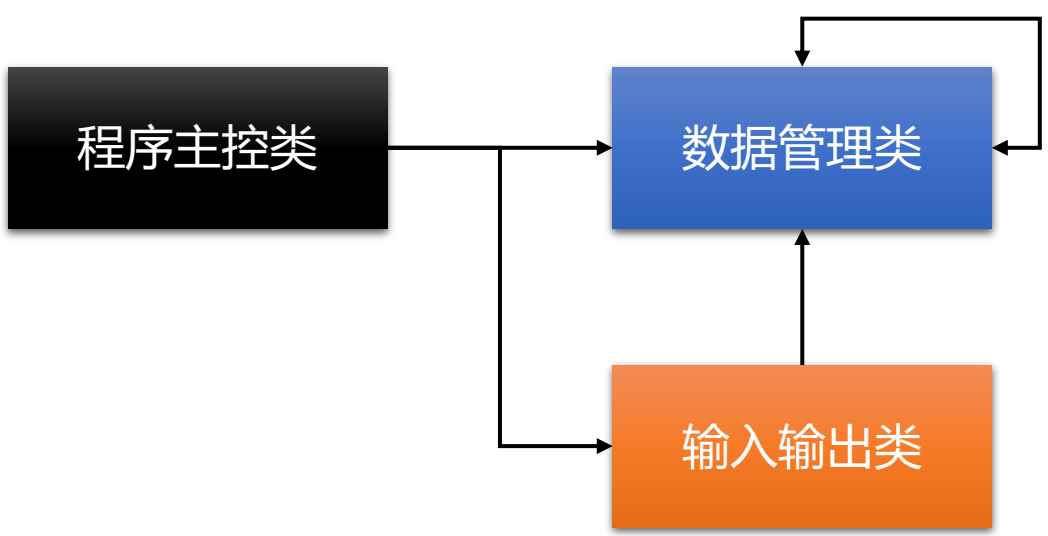
- 我们可以把迭代变化归结为两个维度
  - 数据维度
  - 处理维度
  - 现实中可能同时在这两个维度发生迭代
- 通过这个例子可以看到
  - 两个维度迭代都会带来属性和方法的变化
  - 这说明我们必须综合两个维度进行设计
- 目标是尽可能少改动设计结构
  - 前提是设计结构具有扩展能力
  - 如果不可以，就要在设计上进行大调整：重构

可以进一步思考：把两个维度的*V3*开发放在一起呢？

再次明确几个核心概念

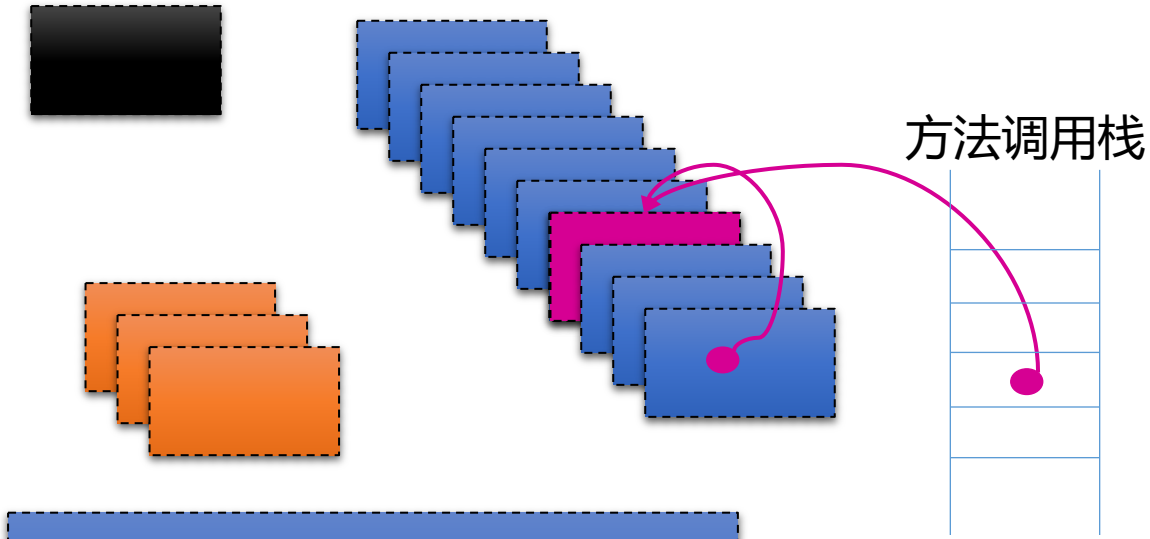
- 对象
  - 存储在内存中的一个结构化**数据体**
  - 任意时刻都拥有确定的取值状态，范围由相应的类定义
  - 包括属性和方法，均有可见性设定
- 对象引用(object reference)
  - 程序变量，**指向**内存中某个**类型相匹配**的实际对象
  - 与被引用的对象(referred object)区分：对象引用类型 vs 对象构建类型
- 对象共享
  - 两个或多个对象，**引用**到**相同对象**的结果
  - 共享可能会产生意想不到的结果（怎么对象状态发生了变化？！）

对象的运行时特性



代码(类)空间

运行时(对象)空间



对象中有哪些数据？
对象引用保存在何处？

对象的运行时特性

- 对象状态是否发生变化
  - 可变对象~不可变对象
- 对象状态外部是否能观察
  - 可观察状态~不可观察状态
- 对象是否能够容忍一些错误
  - 可容忍错误~不可容忍错误

对象状态的可变性

可变对象中可能包括不可变数据

- 可变对象(mutable object)
  - 对象状态可发生外部能够观察到的变化
- 不可变对象(immutable object)
  - 对象的属性不可以被改变
  - 或者对象的属性可以被改变，但是通过外部状态观察不到相应的变化
  - 典型代表：字符串对象
- 使用不可变对象能够降低逻辑复杂度，易于发现问题
  - 可能会导致内存消耗多

```
public class Poly{
    private int[] terms;
    private int deg;
    public Poly(int deg) {…}
    public Poly(int c, int n){…}

    public int degree(){return deg;}
    public int coeff(int d){…}

    public Poly add(Poly q){…}
    public Poly sub(Poly q){…}
}
```

mutable or immutable object?

对象状态的可变性

- 共享访问可变对象可能产生的风险
  - 对象所管理的数据被外部某对象**不受控制的访问**
    - 账户对象如果把账户数据暴露给外部，可能会破坏账户的关键信息
  - 局部对象被共享，导致其生命期**结束时间不确定**
    - 方法构造的局部对象在方法执行结束后其生命期就结束。
    - 如果局部对象被外部对象共享，其生命期结束时间就不确定，**其内存被回收时间也不确定**
  - 如果对象被多个线程不受控共享访问，**运行时**可能会**出错**
    - 多个储户线程同时访问（存、取）一个账户对象，会导致账户余额发生莫名其妙的变化

对象状态的可观察性

- 隐藏是OO设计的**第一原则**
  - 导致某些状态外部不可观察
- 通过方法调用可以把内部状态转化为外部状态
  - 部分可观察
  - 一个容器中管理的具体对象（三角形）对外隐藏
    - 通过size方法了解规模
    - 通过三角形特征值（周长、面积）来检索相关对象
- 有些内部状态无法直接被观察，但可以通过方法执行获得间接观察
  - 一个信用卡账号过去6个月的消费与还款情况无法直接观察
  - 可以通过透支额度获得间接观察
- 如果一个属性无法被观察，且对任何方法执行都无影响，则应去掉

对象的容错性

- 如果一个对象在运行时能够容忍出现的错误，则称具有容错性
  - 如果每个对象都具有容错性，整个程序一定是健壮的
- 容错性的两个条件
  - 能识别方法传入参数的错误
  - 能识别对象状态的错误
- 要做到这两点，必须有针对性的设计
  - 方法应该明确对输入参数的取值要求，一旦不满足，无法正确处理
  - 需要为每个类定义一个“健康”表，满足相应约束为健康，否则有错

如何开展有效测试？

- **设计输入**来导致程序不能完成其**需求**所要求提供的**输出**
  - 按照需求，对输入进行合理划分，形成输入划分层次（树）
  - 覆盖是基本策略，逐渐找到重点/弱点，发现bug的输入区域
- 了解程序的结构和算法逻辑，大致确定bug的可能范围
  - 通过分析程序的执行流程和调用路径，与输入建立关系
- 分析程序所使用的数据管理、输入处理结构和类库，了解其可能的局限性和不适用场景
  - 构造相应的输入特征

如何开展有效测试？

- 高质量的程序标志
  - 稳定，不会crash
  - 准确完成要求的功能
  - 能够识别异常输入
  - **结构清晰、逻辑简单**
- 如何设计测试输入？
  - 程序的输入内容是什么(分析作业要求)
  - 程序的输入格式是什么(分析作业要求)
  - 程序对输入做什么处理(阅读代码)
  - 程序的输出结果是什么(观察程序运行的反馈)

作业分析与建议

- 迭代开发
- 数据维度增加三角函数
  - 为展开和合并带来新的规则
- 处理维度增加动态展开规则，引入自定义函数
  - f(x,y,z)→f(1,x,sin(x))
- 允许出现**多层括号**，迭代展开
- 检查自己的设计和代码
  - 是否能够修修补补“过一年”？
  - 重构势在必行！

作业分析与建议

- 难以通过风格检查中的设计规范时(bad smell)，也是重构时机
- 重构可以解决的问题
  - 长的方法流程（面条代码）：切成片，形成多个方法
  - 重类(heavy class)：按照方法与数据的交互关系分解形成多个类
  - 过度中心化的类聚集交互：某些类需要协调所有事情，职责分派出去
- 重构后的关键事项
  - 与重构前的代码做对比测试，要上量，想办法自动化
  - 分析重构效果，结构图对比
- 重构应是你工具箱中的一个重要工具！

作业分析与建议

- 继续夯实**层次结构**
  - 函数因子是一种因子（可以参与到项中）
  - 自定义函数表达式是一种表达式
- 规则化的展开
  - 函数调用展开：实参对形参的替换
  - 多层括号展开：迭代调用单层展开规则
- 输入字符串扫描
  - 三角函数、函数调用、自定义函数有自己独特的特征
  - 扩展层次化的**正则表达式**或**递归下降方法**