软件架构与中间件

涂志莹 tzy_hit@hit.edu.cn

苏统华 thsu@hit.edu.cn

哈尔滨工业大学

软件架构与中间件

Software Architecture and Middleware

第2章

软件架构的传统风格

课程内容


软件架构与中间件的内涵及要点
软件架构的传统风格
软件架构的评审、测试与优化
软件架构与中间件
计算层的软件架构技术
表示层的软件架构技术
数据层的软件架构技术

第2章 软件架构的传统风格

2.1 软件架构风格概述

2.2 主程序-子过程风格

2.3 面向对象风格

2.4 数据流风格

2.5 事件驱动风格

第2章 软件架构的传统风格

2.6 解释器风格

2.7 分层结构

2.8 模型-视图-控制器

2.9 本章作业

2.3
面向对象风格

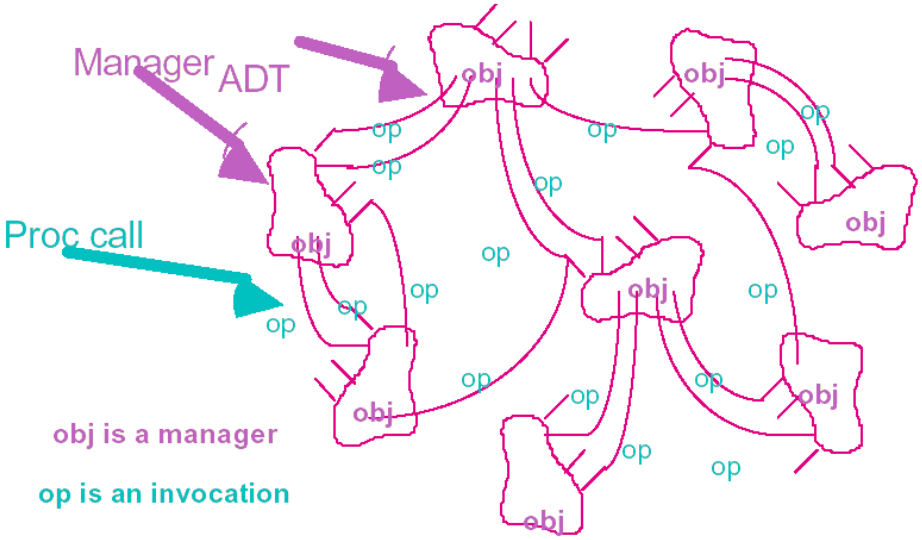
1、风格解析

2、风格应用

2.3.1 面向对象风格解析

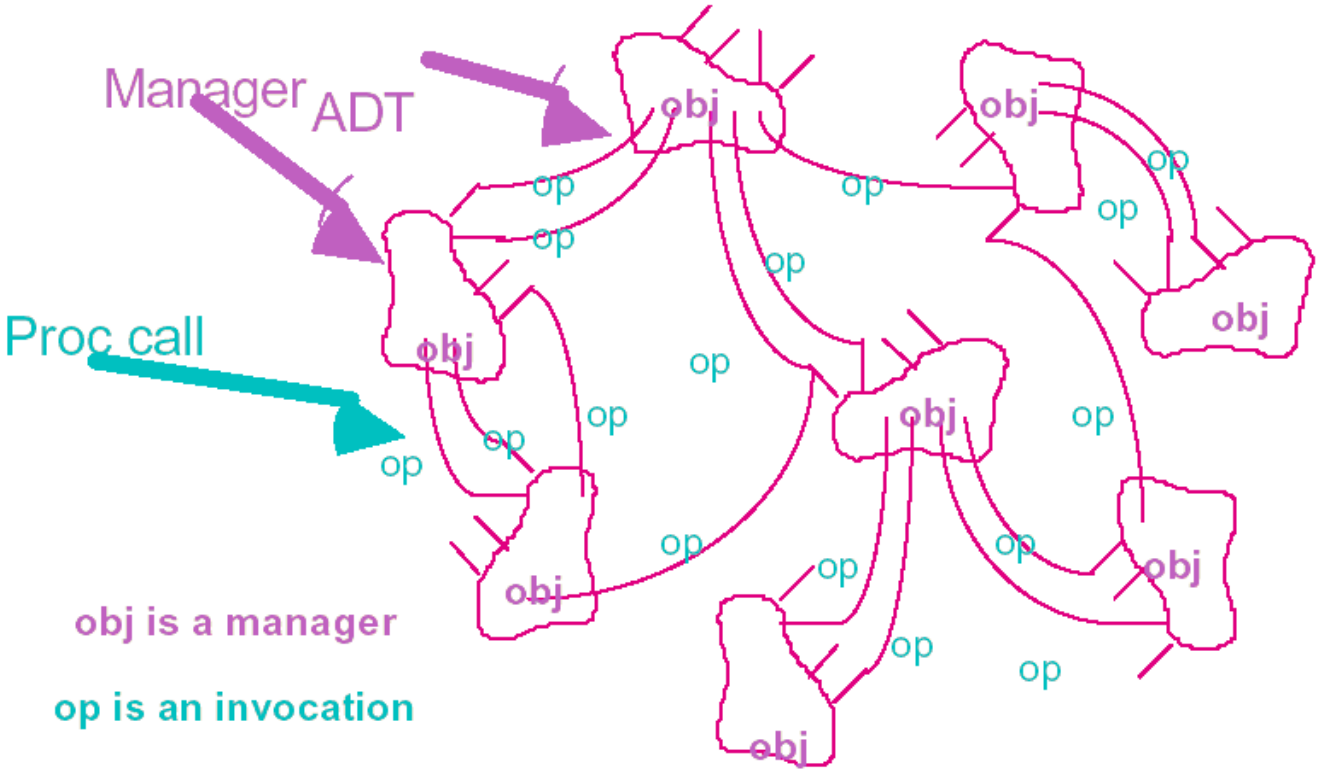
面向对象风格

- Class is an abstract data type (ADT).
- The system is viewed as a collection of objects rather than as functions with messages passed from object to object. (系统被看作对象的集合)
- Each object has its own set of associated operations (每个对象都有一个它自己的操作集合。数据及作用在数据上的操作被封装成抽象数据类型——对象)



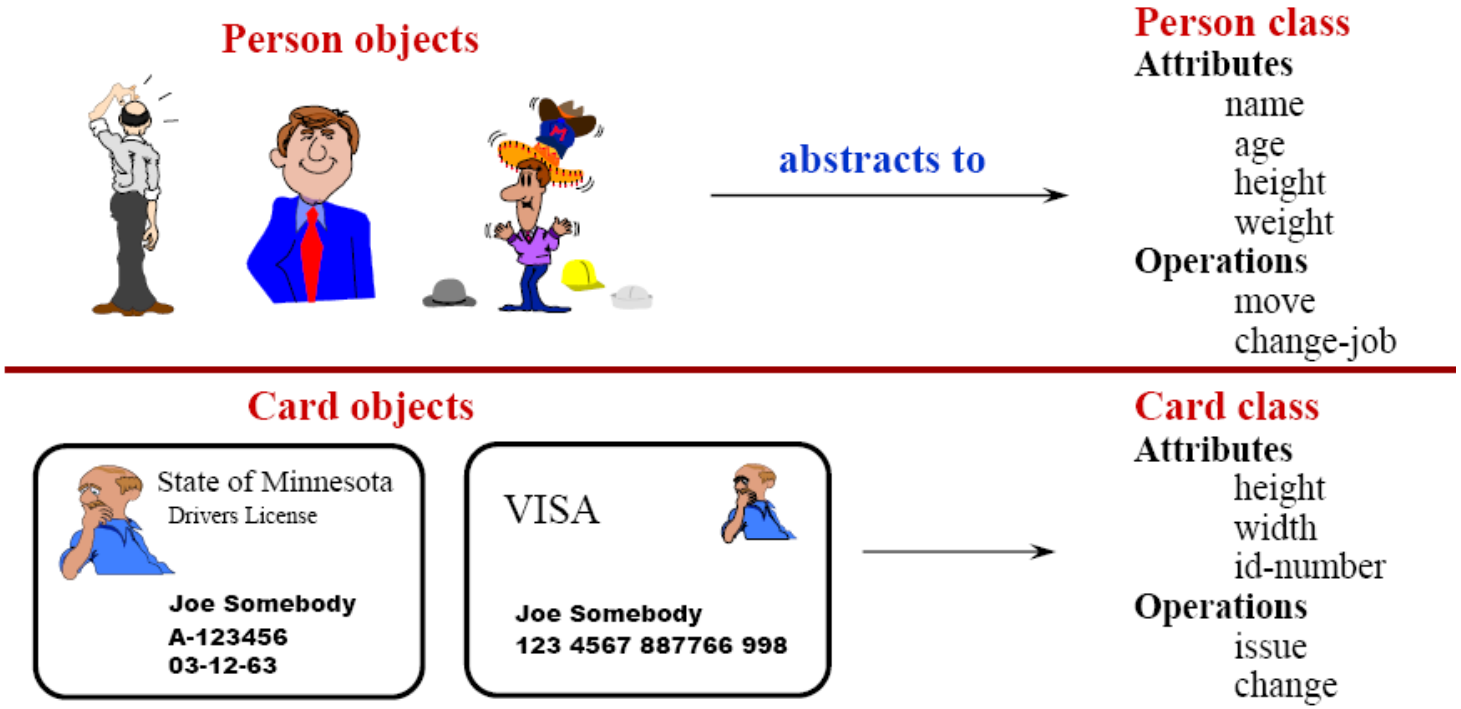
面向对象风格基本构成

- Component: classes and objects (OO风格的构件是：类和对象)
- Connectors: objects interact through function and procedure invocations. (连接件：对象之间通过函数与过程调用实现交互)

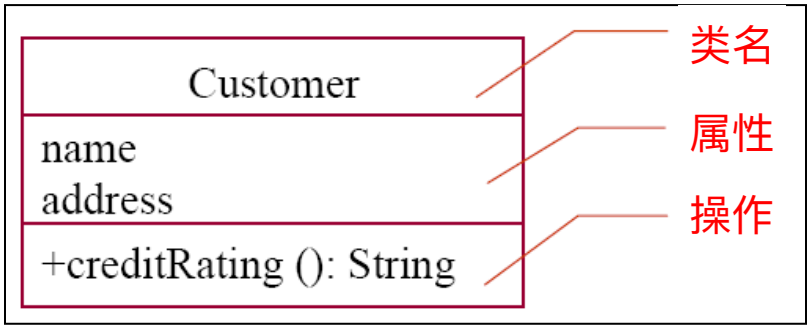


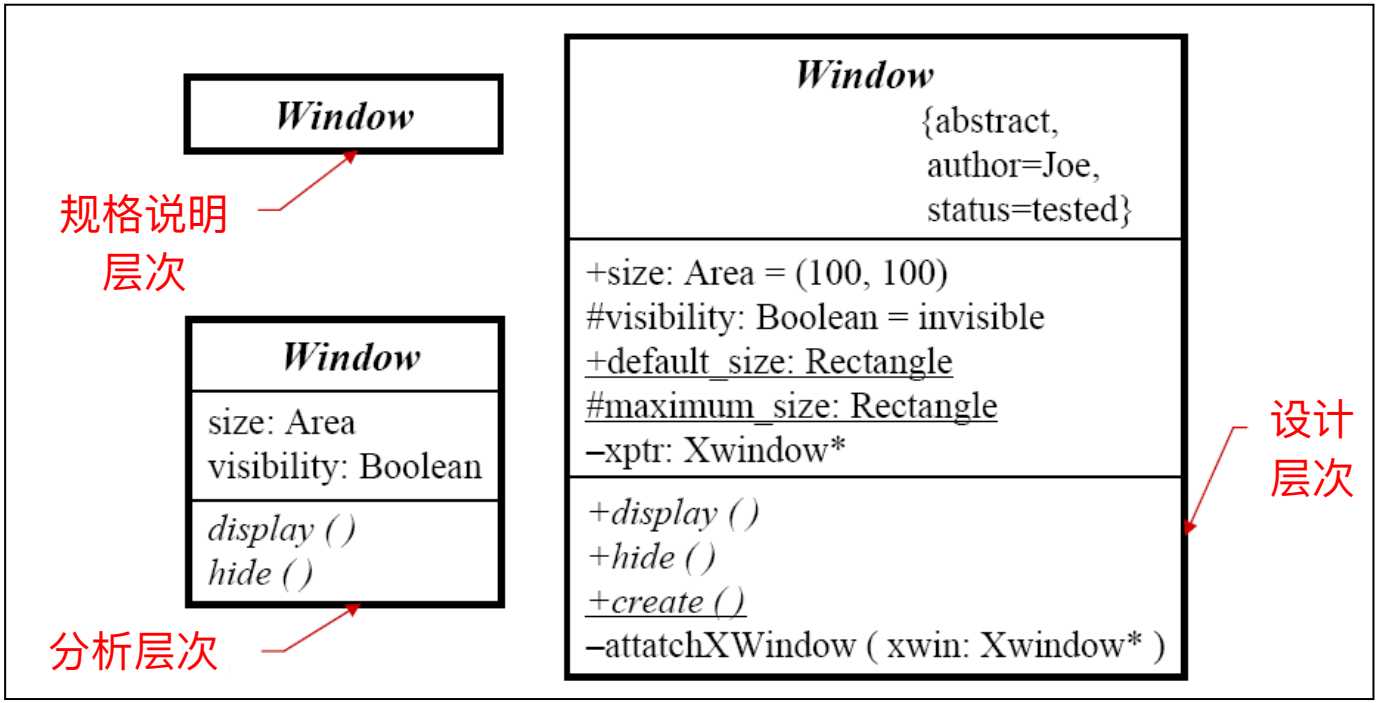
类的内部结构：属性和操作

- 属性(Attribute)：描述对象静态特性的数据项；
- 操作(Operation)：描述对象动态特性的一个动作；

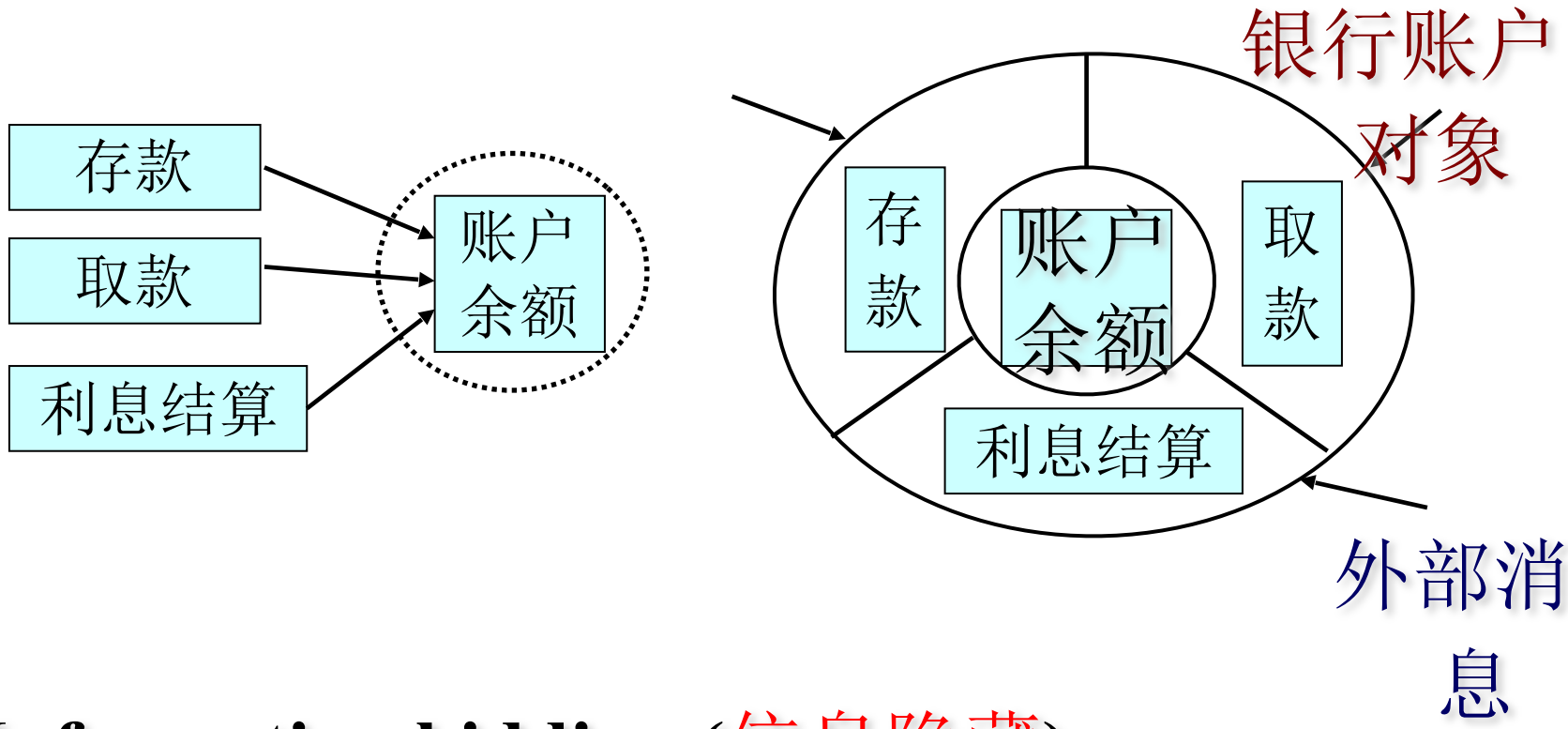


类的内部结构：属性和操作





Main Program/Subroutine VS. OO



Information hidding (信息隐藏)

Encapsulation based on hiding of design "secrets" (内部的设计决策则被封装起来)

Characteristics of OO

这部分要记在A4上

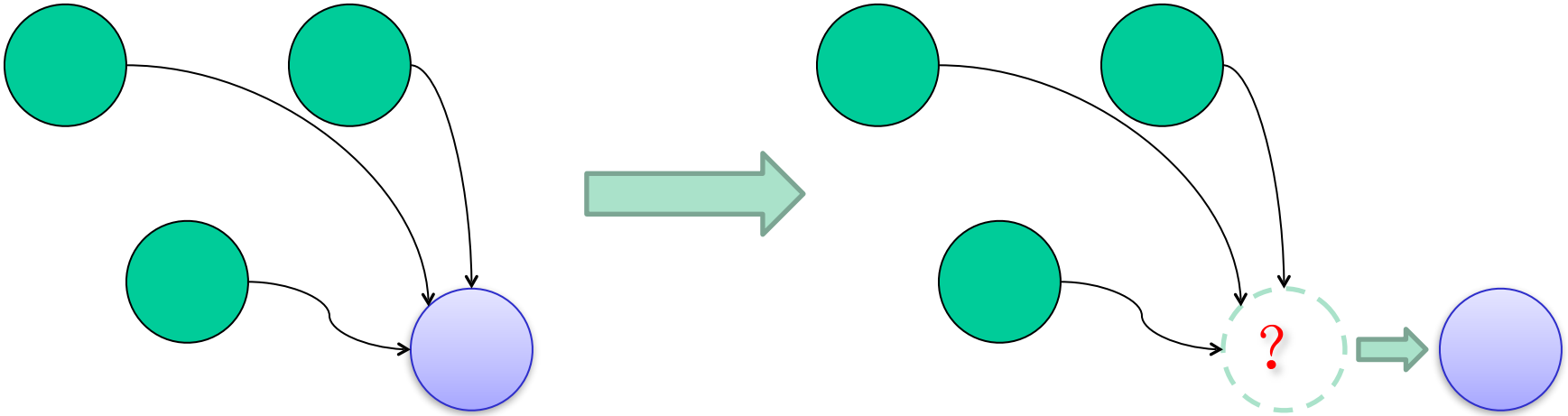
- Encapsulation: Restrict access to certain information (封装：限制对某些信息的访问)
- Interaction: Via procedure calls or similar protocol (交互：通过过程调用或类似的协议)
- Polymorphism: Choose the method at run-time (多态：在运行时选择具体的操作)
- Inheritance: Keep one definition of shared functionality (继承：对共享的功能保持唯一的接口)
- Dynamic binding: Determine actual operation to call at runtime (动态绑定：运行时决定实际调用的操作)
- Reuse and maintenance: Exploit capsulation and locality (复用和维护)

Advantages of OO

- Reuse and maintenance: Exploit encapsulation and locality to increase productivity (复用和维护：利用封装和聚合提高生产力)
  - ➢ It is possible to change the implementation without affecting its clients.
- Real world mapping: For some systems, there may be an obvious mapping from real world entities to system objects (反映现实世界)
- Easy decomposition of a system: the bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents. (容易分解一个系统)

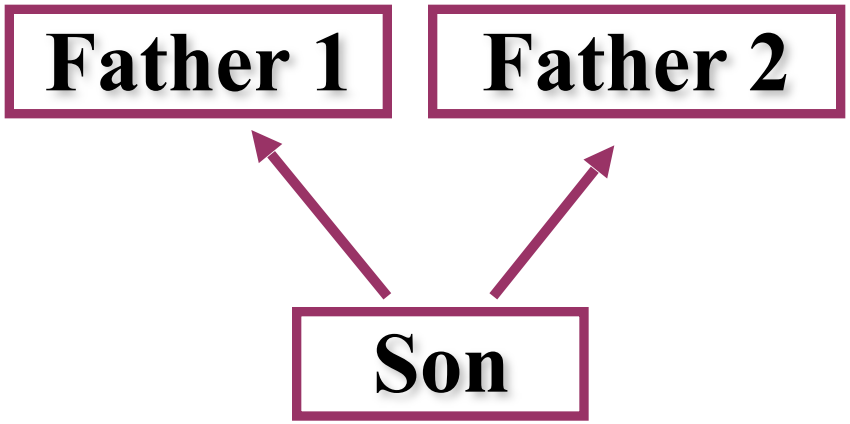
Disadvantages of OO

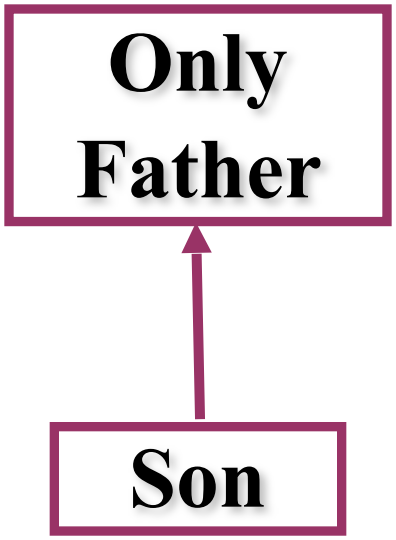
- Management of many objects: Need structure on large set of definitions (管理大量的对象：怎样确立大量对象的结构)
- In order for one object to interact with another (via procedure call) it must know the identity of that object. (必须知道对象的身份标识)



- Inheritance introduces complexity and is undesirable in critical systems (继承引起复杂度，关键系统中慎用)

Inheritance in C++ and Java



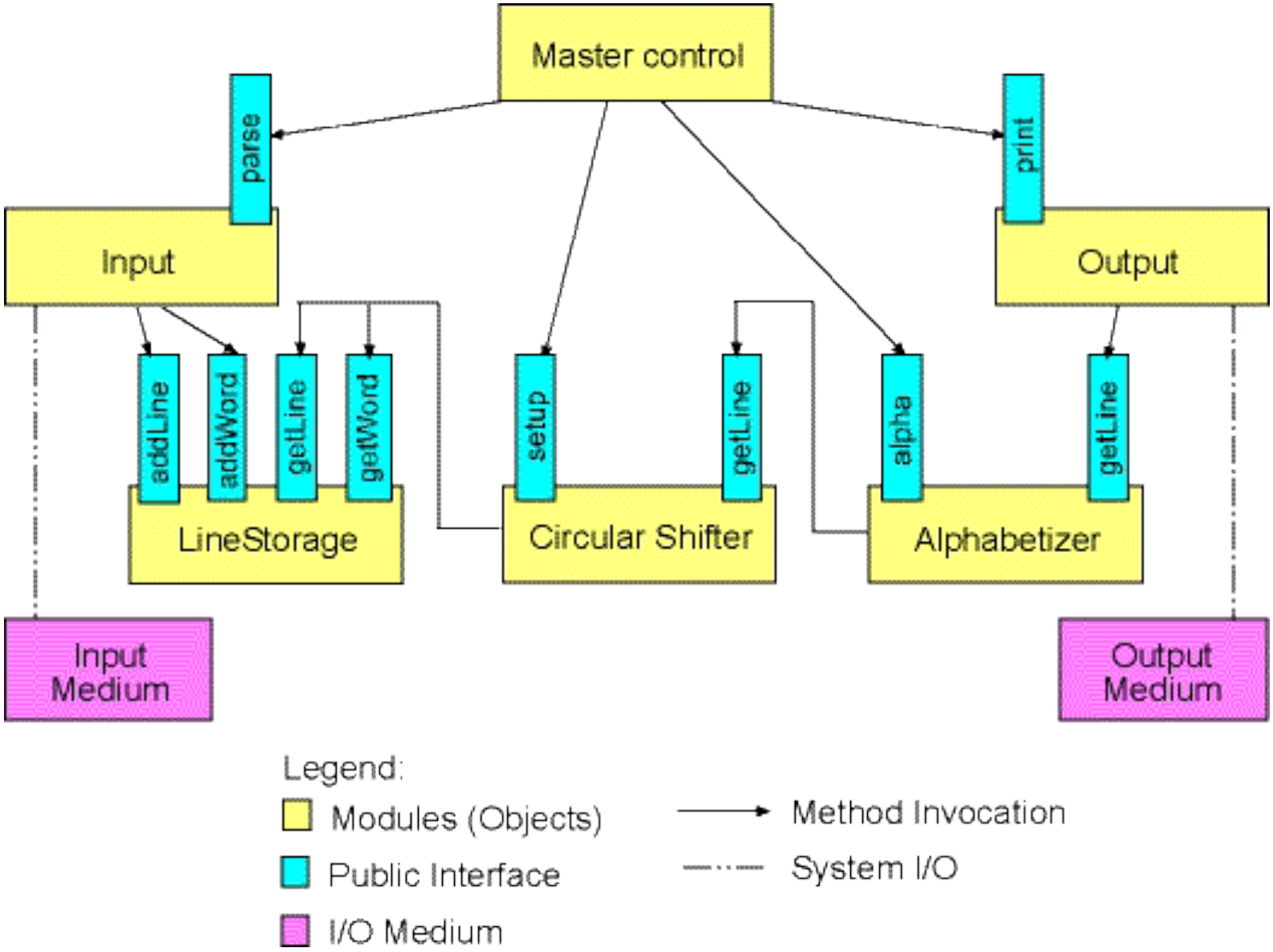


In C++, one class can inherit from two classes

In Java, one class can only inherit from one class

2.3.2 面向对象风格应用

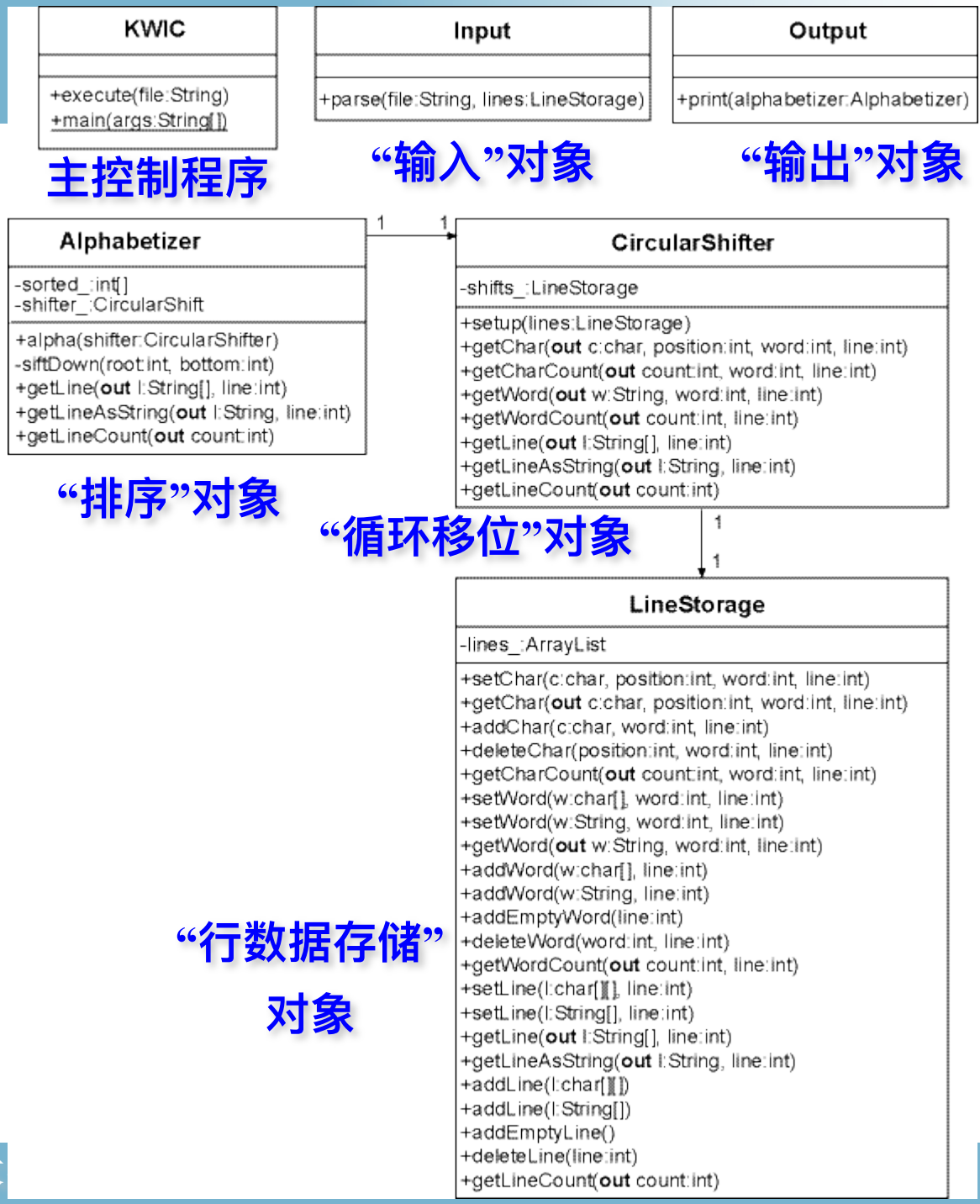
KWIC Solution: Abstract Data Types



KWIC Solution: Abstract Data Types

- 采用OO的思想，数据和作用在数据上的读写操作被封装为object，主程序调用这些object，形成控制流程；
  - Data is no longer directly shared by the computational components. (数据不再被构件直接共享，而是被封装在了Object中)
  - Each module provides an interface that permits other components to access data only by invoking procedures in that interface. (每个对象提供了一个接口，允许其他对象通过该接口调用对该对象内封装的数据的操作)

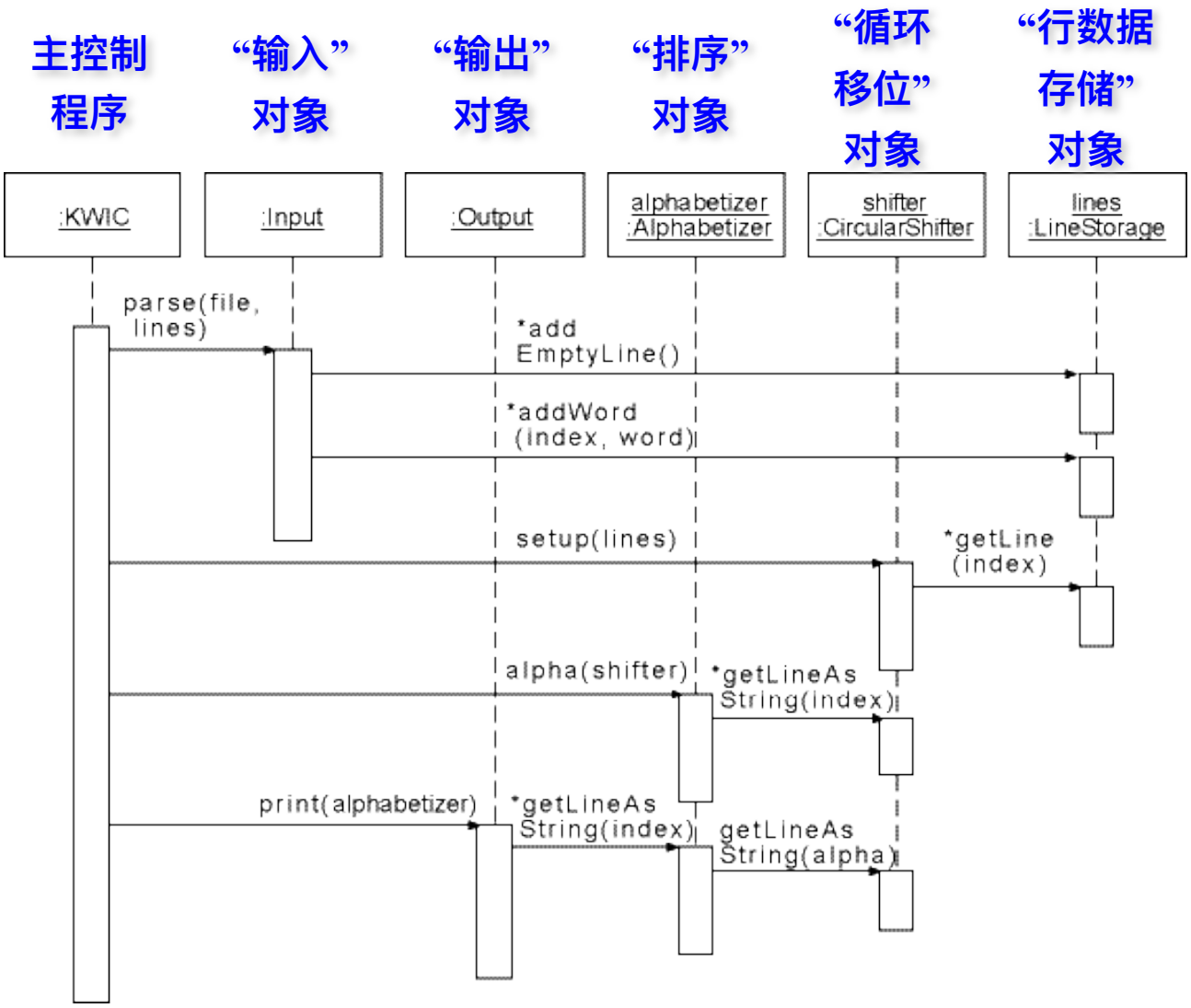
Class Diagram: System Statics



KWIC Solution : Abstract Data Types

- Master control object (主控制对象：负责控制其他各对象中方法的调用次序)
- Input object (负责从输入文件中读取/解析数据并将其存储在LineStorage对象中)
- LineStorage object (存储和管理字符、单词、行)
- CircularShifter object (负责对LineStorage对象中存储的数据进行循环移位)
- Alphabetizer object (负责对循环移位后得到的数据进行排序)
- Output object (负责打印输出排序后的数据)

Sequence Diagram: System Dynamics



Class KWIC

```
//主对象KWIC构造五个对象实例
LineStorage lines = new LineStorage();
Input input = new Input();
CircularShifter shifter = new CircularShifter();
Alphabetizer alphabetizer = new Alphabetizer();
Output output = new Output();
//然后分别调用这五个对象实例的某些方法
input.parse(file, lines);
shifter.setup(lines);
alphabetizer.alpha(shifter);
output.print(alphabetizer);
```

KWIC Solution : Abstract Data Types

- Advantages:
  - Both algorithms and data representations can be changed in individual modules without affecting others. (某一构件的算法与数据结构的修改不会影响其他构件)
  - Reuse is better supported than in the first solution because modules make fewer assumptions about the others with which they interact. (构件之间依赖性降低，提高了复用度)
- Disadvantages:
  - The solution is not particularly well-suited to enhancements. (不适合功能的扩展)
  - For adding new functions to the system, the implementor must either modify the existing modules or add new modules that lead to performance penalties. (为了增加新功能，要么修改已有的模块，要么就加入新的模块，性能会受到影响)

2.4
数据流风格

1、数据流风格概述

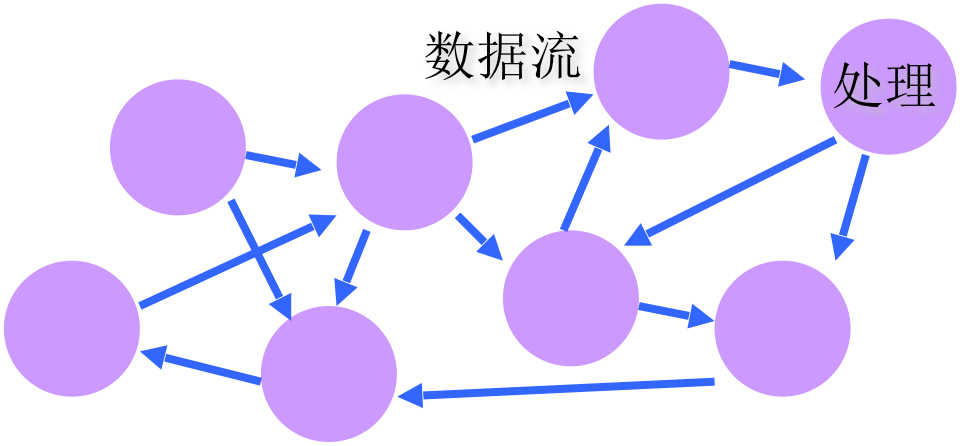
2、管道-过滤器风格

3、风格应用

2.4.1 数据流风格概述

数据流风格的直观理解

处理操作：数据到达即被激活，无数据时不工作



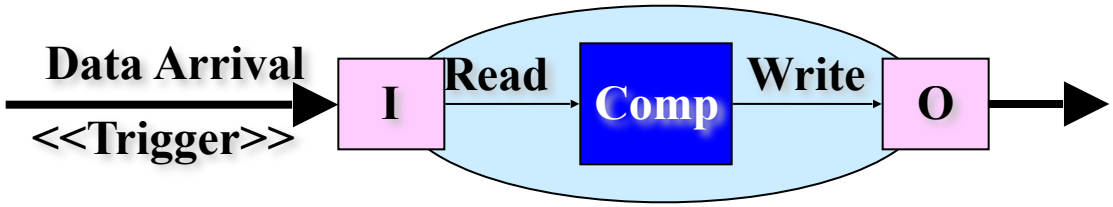
一个直观实例：在MS Excel中，改变某个单元格的值，则依赖于该单元格的其他单元格的值也会随之改变

数据流风格的特征

- A data flow system is one in which
  - the availability of data controls the computation (数据的可用性决定着处理<计算单元>是否执行)
  - the structure of the design is dominated by orderly motion of data from process to process (系统结构：数据在各处理之间的有序移动)
  - in a pure data flow system, there is no other interaction between processes (在纯数据流系统中，处理操作之间除了数据交换，没有任何其他的交互)

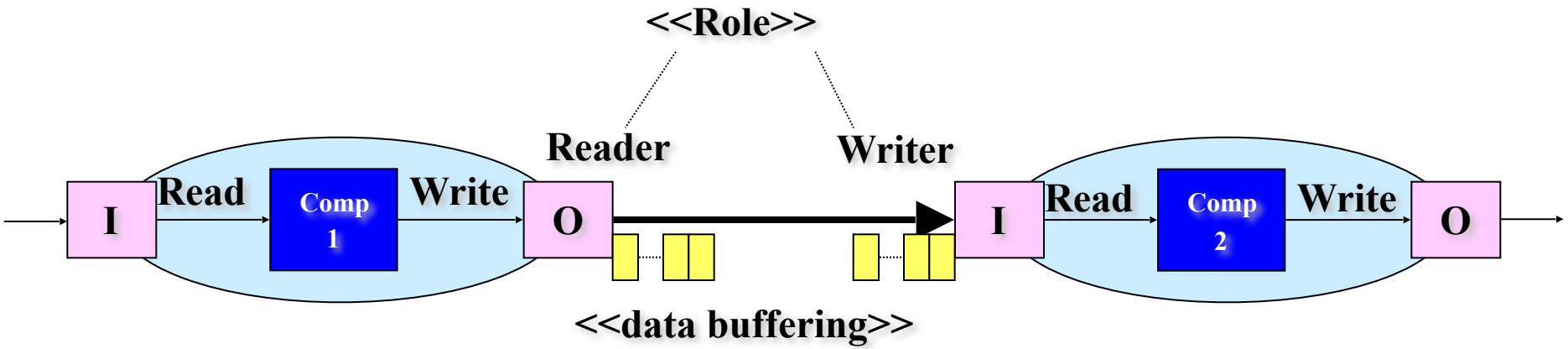
数据流风格的基本构件(Component)

- Components: data processing components(基本构件：数据处理)
  - Interfaces are input ports and output ports (构件接口：输入端口和输出端口)
  - Input ports read data; output ports write data (从输入端口读取数据，向输出端口写入数据)
  - Computational model: read data from input ports, compute, write data to output ports (计算模型：从输入端口读数，经过计算/处理，然后写到输出端口)



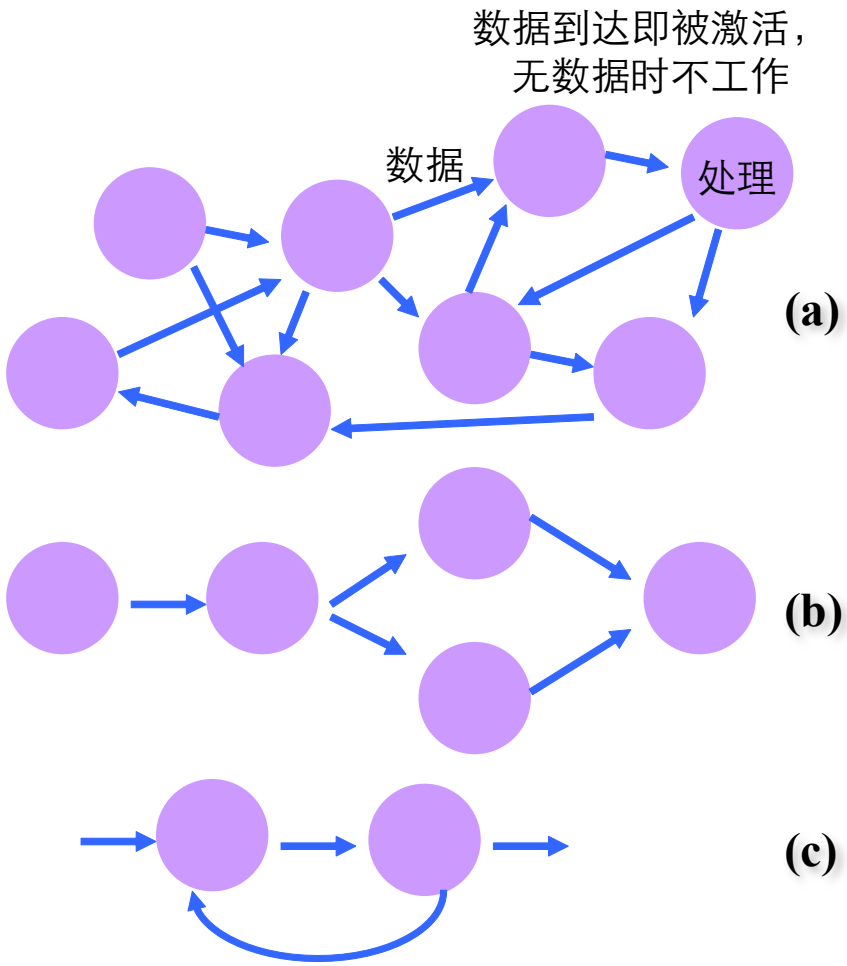
数据流风格的连接件(Connector)

- Connectors: data flow (data stream) (连接件：数据流)
  - Uni-directional, usually asynchronous, buffered (单向、通常是异步、有缓冲)
  - Interfaces are reader and writer roles (接口角色：reader和writer)
  - Computational model (计算模型: 把数据从一个处理的输出端口传送到另一个处理的输入端口)



数据流风格的拓扑结构(Topology)

- Arbitrary graphs (任意拓扑结构的图)



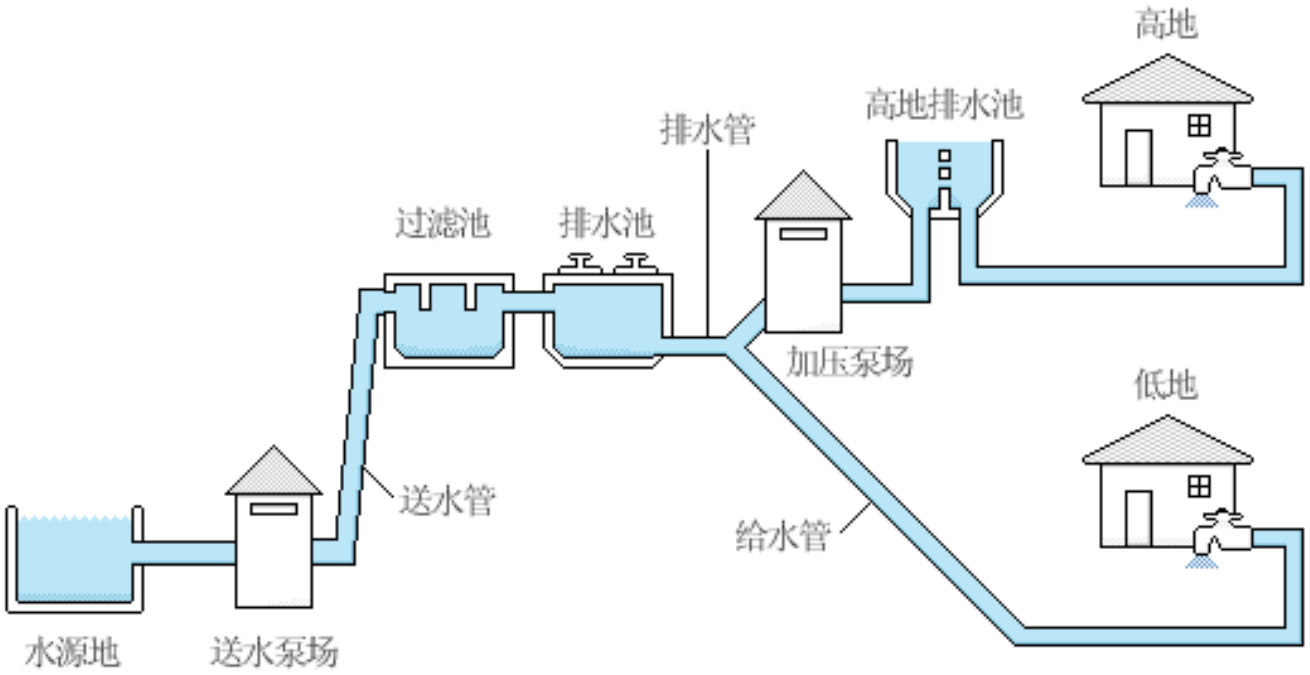
In general, data can flow in Arbitrary patterns (一般来说，数据的流向是无序的)

Often we are primarily Interested in nearly linear Data flow systems (我们主要关注近似线性的数据流)

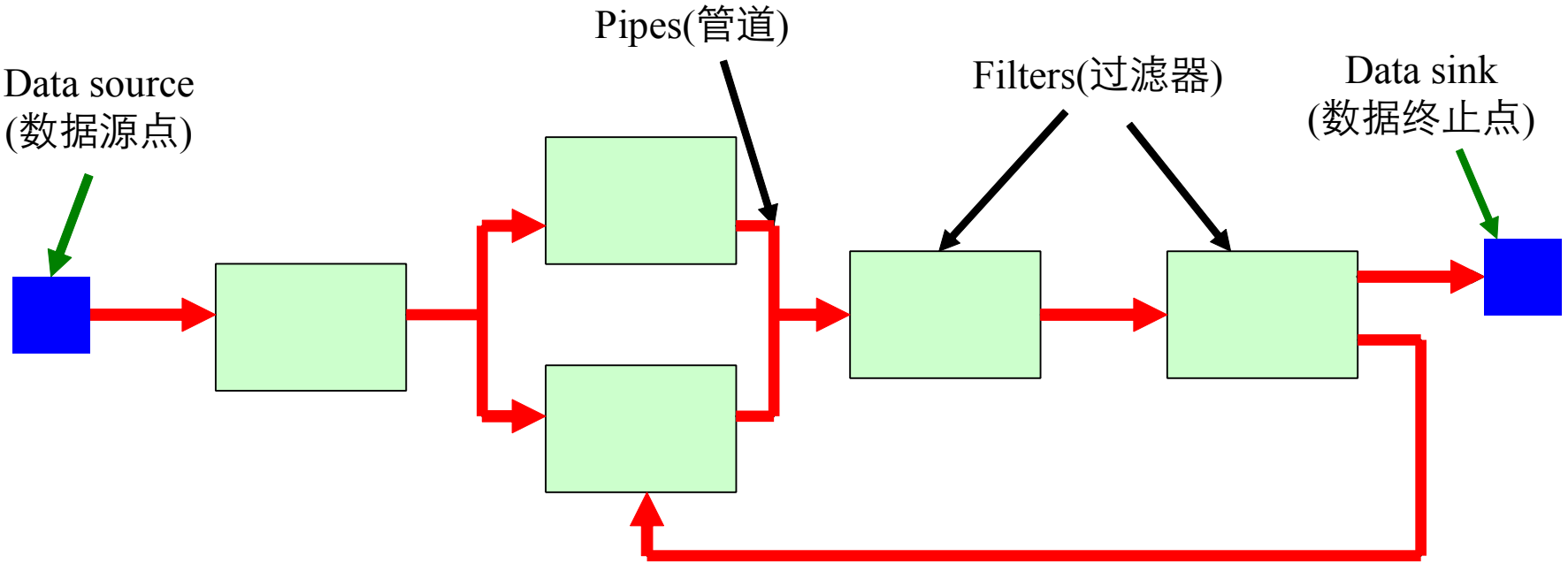
Or in very simple, highly constrained cyclic structures (或在限度内的循环数据流)

2.4.2 管道-过滤器风格

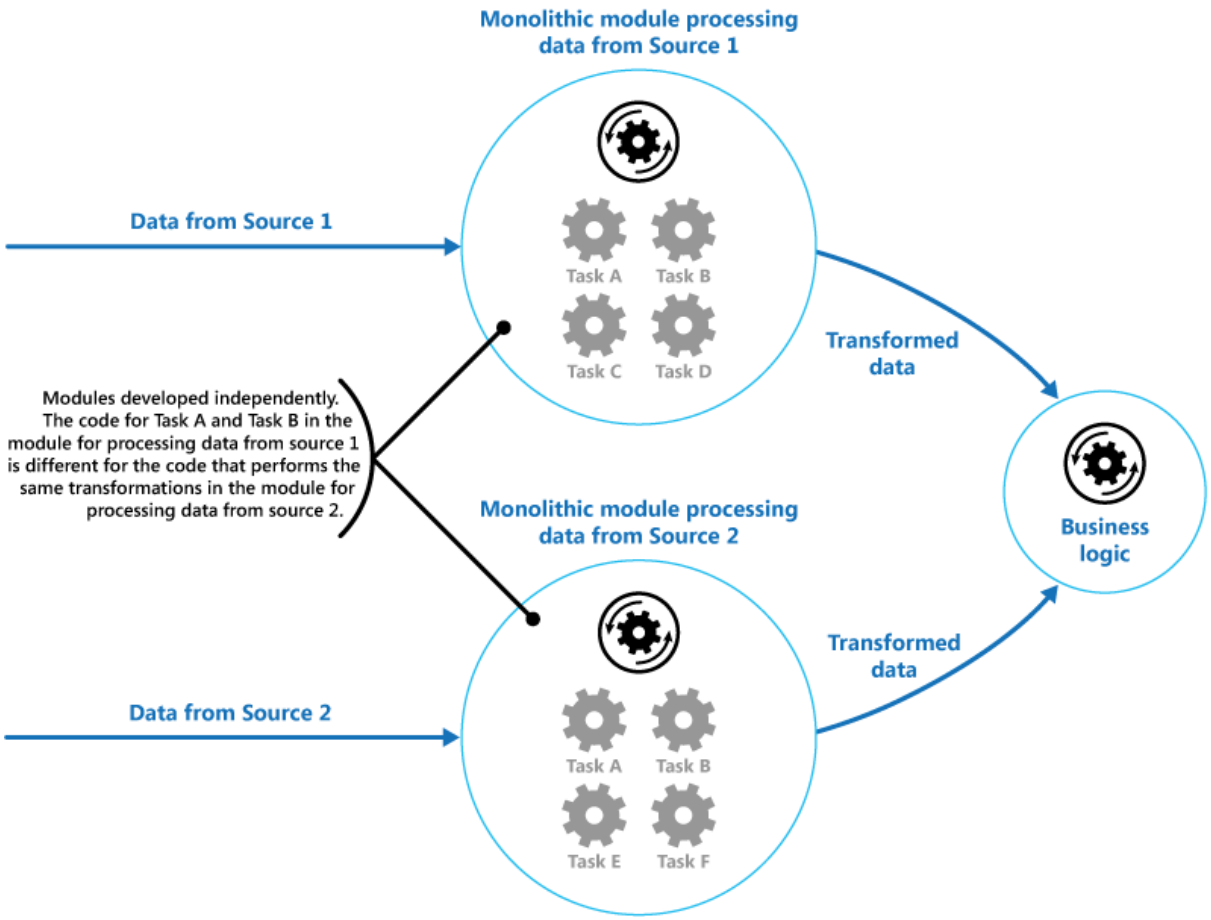
自来水处理中的“Pipe-and-Filter”结构



Pipe-and-Filter风格的直观结构

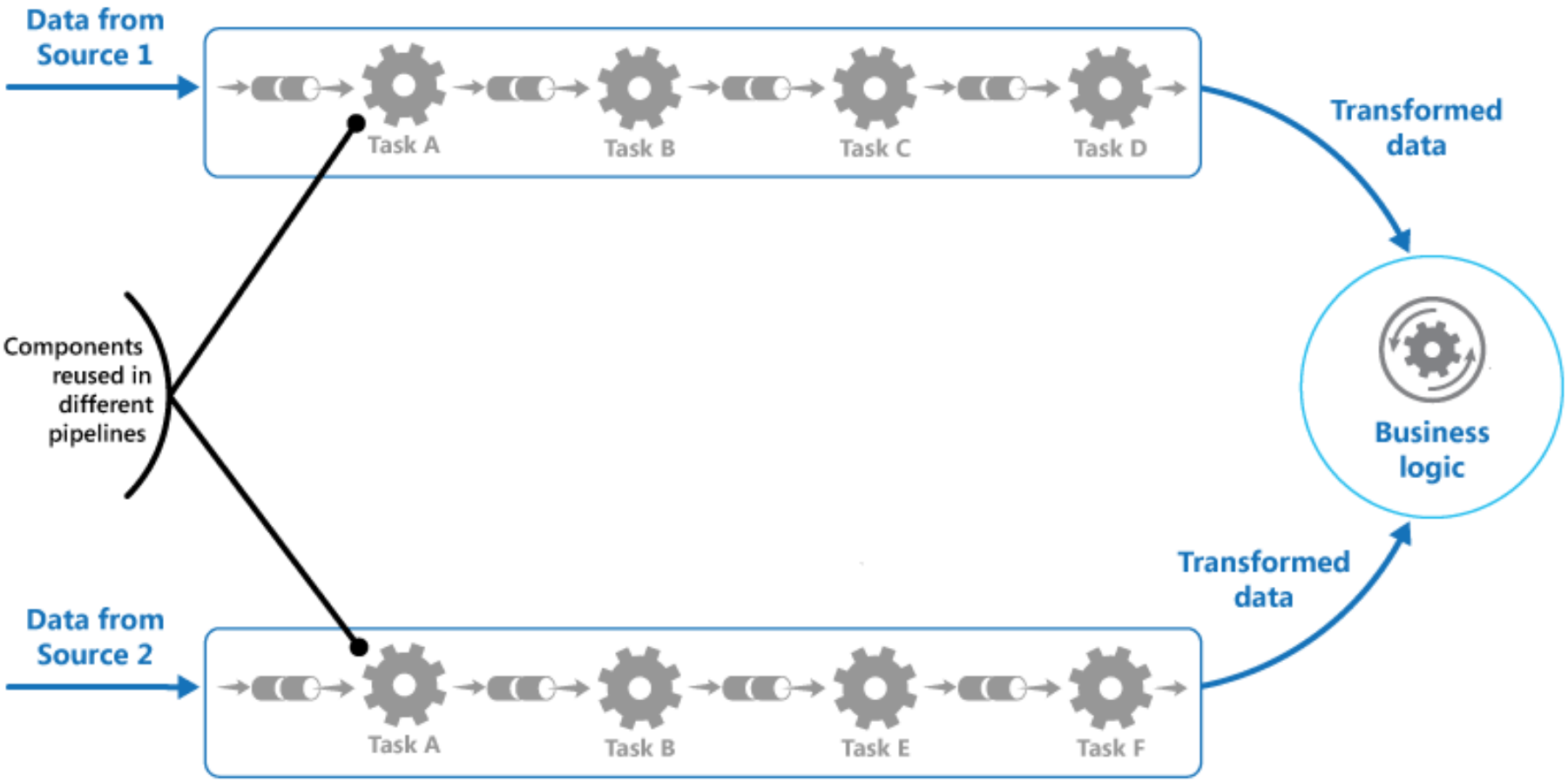


Scenario



https://docs.microsoft.com/zh-cn/azure/architecture/patterns/pipes-and-filters

Scenario



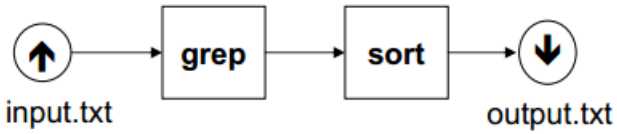
https://docs.microsoft.com/zh-cn/azure/architecture/patterns/pipes-and-filters

基本概念

- 适用场景：数据源源不断的产生，系统需要对这些数据进行若干处理(分析、计算、转换等)。
- 解决方案：
  - ➢把系统分解为几个顺序的处理步骤，这些步骤之间通过数据流连接，一个步骤的输出是另一个步骤的输入；
  - ➢每个处理步骤由一个过滤器构件(Filter)实现；
  - ➢处理步骤之间的数据传输由管道(Pipe)负责。
- 每个处理步骤(过滤器)都有一组输入和输出，过滤器从管道中读取输入的数据流，经过内部处理，然后产生输出数据流并写入管道中。

Pipe-and-Filter风格的基本构成

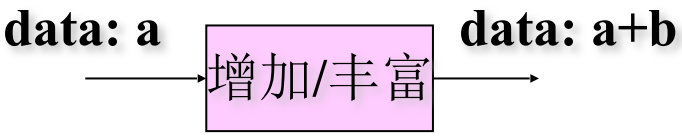
- Components: Filters — process data streams (构件：过滤器，处理数据流)
  - A filter encapsulates a processing step (algorithm or computation) (一个过滤器封装了一个处理步骤)
  - Data source and data sink are particular filters (数据源点和数据终止点可以看作是特殊的过滤器)
- Connectors: A pipe connects a source and a sink filter (连接件：管道，连接一个源和一个目的过滤器)
  - Pipes move data from a filter output to another filter input (转发数据流)
  - Data may be a stream of ASCII characters (数据可能是ASCII字符形成的流)
- Topology: Connectors define data flow graph (连接器定义了数据流的图，形成拓扑结构)
- Example
  - Unix shell: cat input.txt | grep “test” | sort > output.txt

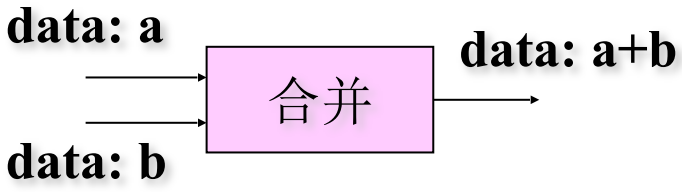


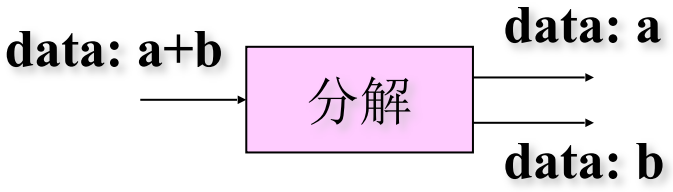
1 过滤器(Filter)

- Incrementally transform some of the source data into sink data (目标：将源数据递增的变换成目标数据)
- Stream to stream transformation (从“数据流”→“数据流”的变换)
  - enrich data by computation and adding information (通过计算和增加信息来丰富数据)
  - refine by distilling data or removing irrelevant data (通过浓缩和删减来精炼数据)
  - transform data by changing its representation (通过改变数据表现方式来转化数据)
  - decompose data to multiple streams (将数据分解为多个流)
  - merge multiple streams into one stream (将多个数据流合并为一个)

过滤器对数据流的五种变换类型

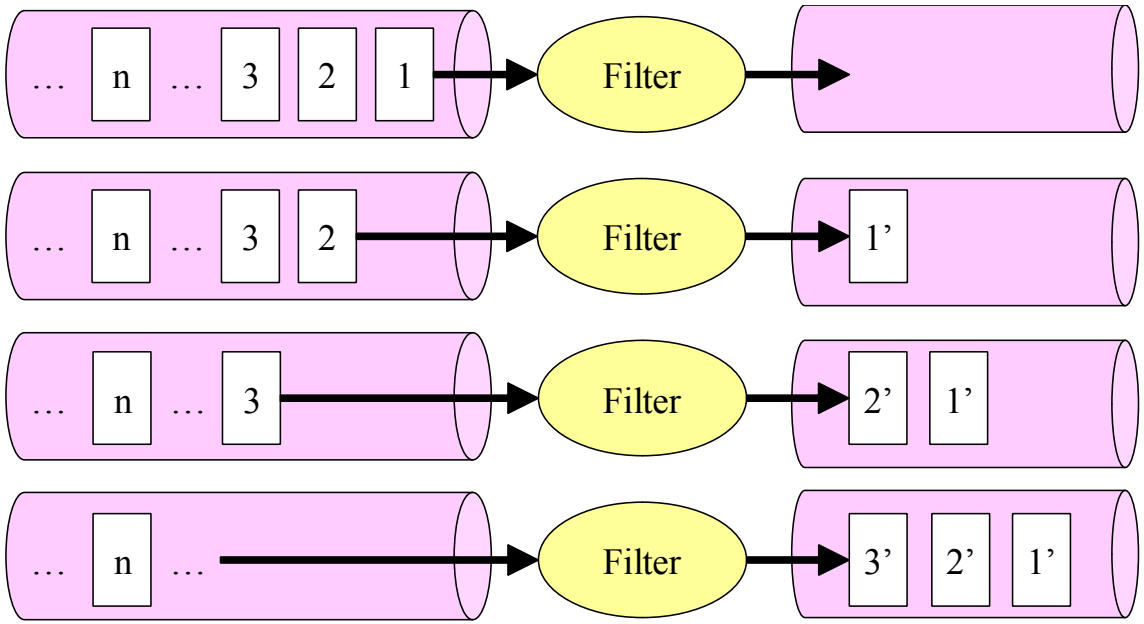






过滤器读取与处理数据流的方式

- Incrementally transform data from the source to the sink (递增的读取和消费数据)
  - data is processed as it arrives, not gathered then processed (数据到来时便被处理，不是收集完了然后处理，即在输入被完全消费之前，输出便产生了)



过滤器的其他特征

- Filters are independent entities, i.e.,
  - no context in processing streams (无上下文信息)
  - no state preservation between instantiations (不保留状态)
  - no knowledge of upstream/downstream filters (对上下游的其他过滤器无任何了解)

过滤器的状态

- 停止状态：表示过滤器处于待启动状态，当外部启动过滤器后，过滤器处于处理状态。
- 处理状态：表示过滤器正处理输入数据队列中的数据。
- 等待状态：表示过滤器的输入数据队列为空，此时过滤器等待，当有新的数据输入时，过滤器处于处理状态。

2 管道(Pipe)

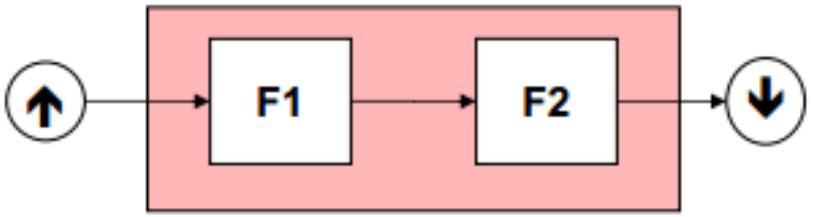
- Move data from a filter's output to another filter's input (or to a device or file) (作用：在过滤器之间传送数据)
  - One way flow from one data source to one data sink (单向流)
  - A pipe may implement a buffer (可能具有缓冲区)
  - collections can be used to buffer the data passed through pipes: files, arrays, dictionaries, trees, etc. (数据缓冲区可以是文件、数组、字典、树等集合类型)
  - Pipes form data transmission graph (管道形成数据传输图)
  - the correctness of the output should not depend upon the order in which the pipes are connected in a pipe-and-filter network(管道的先后顺序不影响输出的结果)
- 不同的管道中流动的数据流，可能具有不同的数据格式(Data format)。
  - 原因：数据在流过每一个过滤器时，被过滤器进行了丰富、精练、转换、合并、分解等操作，因而发生了变化。

管道的实现机制：以Linux为例

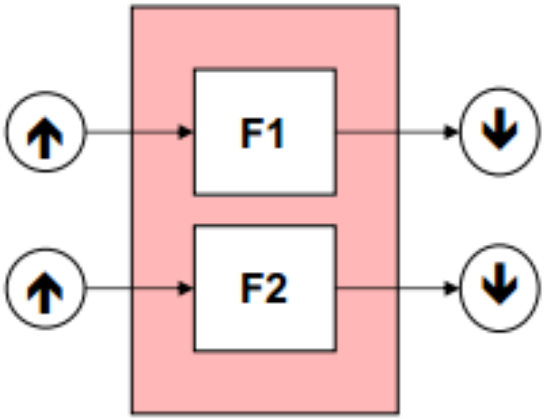
- 在Linux中，管道本质上也是一种特殊的文件，但它又和一般的文件有所不同，具体表现为：
  - 限制管道的大小。实际上，管道是一个固定大小的缓冲区。比如在Linux中，该缓冲区的大小为1页，即4K字节，使得它的大小不象文件那样无限制地增长。
  - 使用单个固定缓冲区也会带来问题，比如在写管道时可能变满，当这种情况发生时，随后对管道的write()调用将默认地被阻塞，等待某些数据被读取，以便腾出足够的空间进行write()调用。
  - 读取进程也可能工作得比写进程快。当所有当前进程数据已被读取时，管道变空。当这种情况发生时，一个随后的read()调用将默认地被阻塞，等待某些数据被写入，这解决了read()调用返回文件结束的问题。
- 注意：从管道读数据是一次性操作，数据一旦被读，它就从管道中被抛弃，释放空间以便写更多的数据。

管道连接过滤器的方式

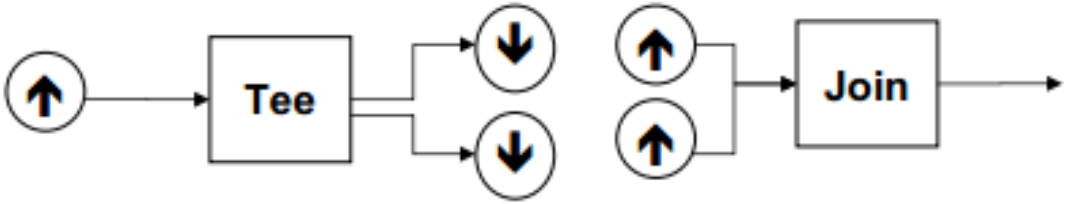
- Sequential Composition
  Unix: F1 | F2



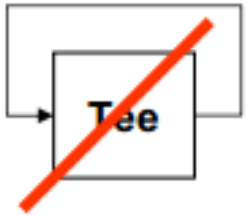
- Parallel Composition
  Unix: F1 & F2



- Tee & Join



- Restriction to **Linear Composition**



管道-过滤器风格的优点

- 使得系统中的构件具有良好的隐蔽性和高内聚、低耦合的特点；
  - ➢支持软件复用：
    - 允许设计者将整个系统的输入/输出行为看成是多个过滤器的行为的简单合成；
    - 只要提供适合在两个过滤器之间传送的数据，任何两个过滤器都可被连接起来；
  - ➢系统维护和增强系统性能简单：
    - 新的过滤器可以添加到现有系统中来，旧的可以被改进的过滤器替换掉；

维护更加简单！

- 允许对一些如吞吐量、死锁等属性的分析；
- 支持并行执行：
  - ➢每个过滤器是作为一个单独的任务完成，因此可与其它任务并行执行。

管道-过滤器风格的缺点

- 通常导致进程成为批处理的结构
  - 这是因为虽然过滤器可增量式地处理数据，但它们是独立的，所以设计者必须将每个过滤器看成一个完整的从输入到输出的转换；
- 不适合处理交互的应用
  - 当需要增量地显示改变时，这个问题尤为严重；
- 因为在数据传输上没有通用的标准，每个过滤器都增加了解析和合成数据的工作，这样就导致了系统性能下降，并增加了编写过滤器的复杂性。
  - 绝大部分处理时间消耗在格式转换上

第2章

软件架构的传统风格

Thanks for listening

涂志莹、苏统华
哈尔滨工业大学计算机学院
企业与服务计算研究中心