2. Software Architecture Style
（软件体系结构风格）

Agenda

- Taxonomy of styles
  - Data Flow
  - Call/Return
  - Data-centered
  - Virtual Machine
  - Independent Component
- Other styles

软件体系结构的定义

SA = {Components, Connectors, Constraints}

- 构件（Components）是功能单元，执行预定义的服务并且与其他构件交互。
- 连接器（ Connectors ）定义交互协议与策略。
- 约束（Constraints） 定义了系统必须服从的规则。

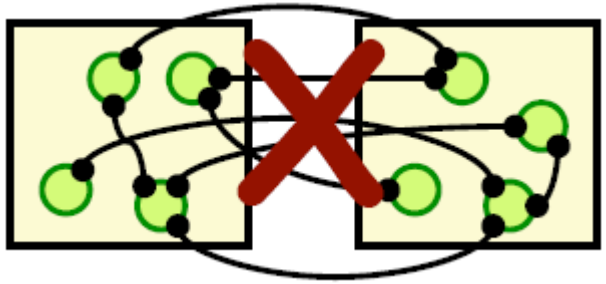
---------- Garlan and Shaw

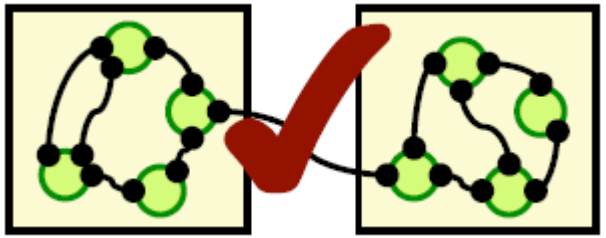
Coupling (耦合) and Cohesion (内聚)

- **Architectural Building blocks:**

Coupling (耦合) and Cohesion (内聚)

- **A good architecture:**
  - **Minimizes coupling between modules**
    - **Goal: modules don't to know much about one another to interact**
    - **Low coupling makes future change easier**
  - **Maximizes the cohesion of each module**
    - **Goal: the contents of each module are strongly inter-related**
    - **High cohesion makes a module easier to understand**





Architectural Styles

- **A style**
  - **describes a class of architectures**（*描述一类体系结构*）
  - **is independent on the problems**（*独立于实际问题，强调了软件系统中通用的组织结构*）
  - **is found repeatedly in practice**（*在实践中被多次设计、应用*）
  - **is a package of design decisions**（*是若干设计思想的综合*）
  - **has known properties that permit reuse**（*具有已经被熟知的特性，并且可以复用*）

软件体系结构风格的概念

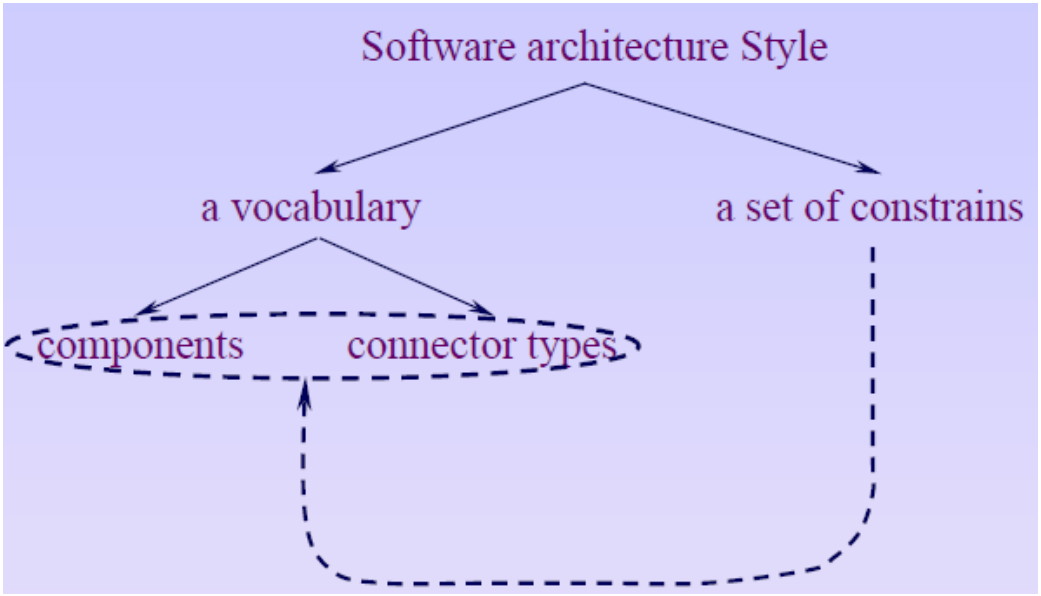
- 软件体系结构风格（Architectural Style）
  - 一种体系结构风格以结构组织模式定义了一个系统家族
  - 关于构件和连接件类型的术语；一组约束对它们组合方式的规定；一个或多个语义模型，规定了如何从各成分的特性决定系统整体特性
  - 概括地说，一种软件体系结构风格刻画一个具有共享结构和语义的系统家族

Benefits of Using Styles

- **Design reuse**
  - Well-understood solutions applied to new problems
- **Code reuse**
  - Shared implementations of invariant aspects of a style
- **Understandability of system organization**
  - A phrase such as “client-server” conveys a lot of information
- **Interoperability (*互操作*)**
  - Supported by style standardization
- **Style-specific analyses**
  - Enabled by the constrained design space
- **Visualizations**
  - Style-specific depictions matching engineers’ mental models

Software Architectural Style

- 软件体系结构风格（Software architecture style）是描述某一特定应用领域中系统组织方式的惯用模式（idiomatic paradigm）。



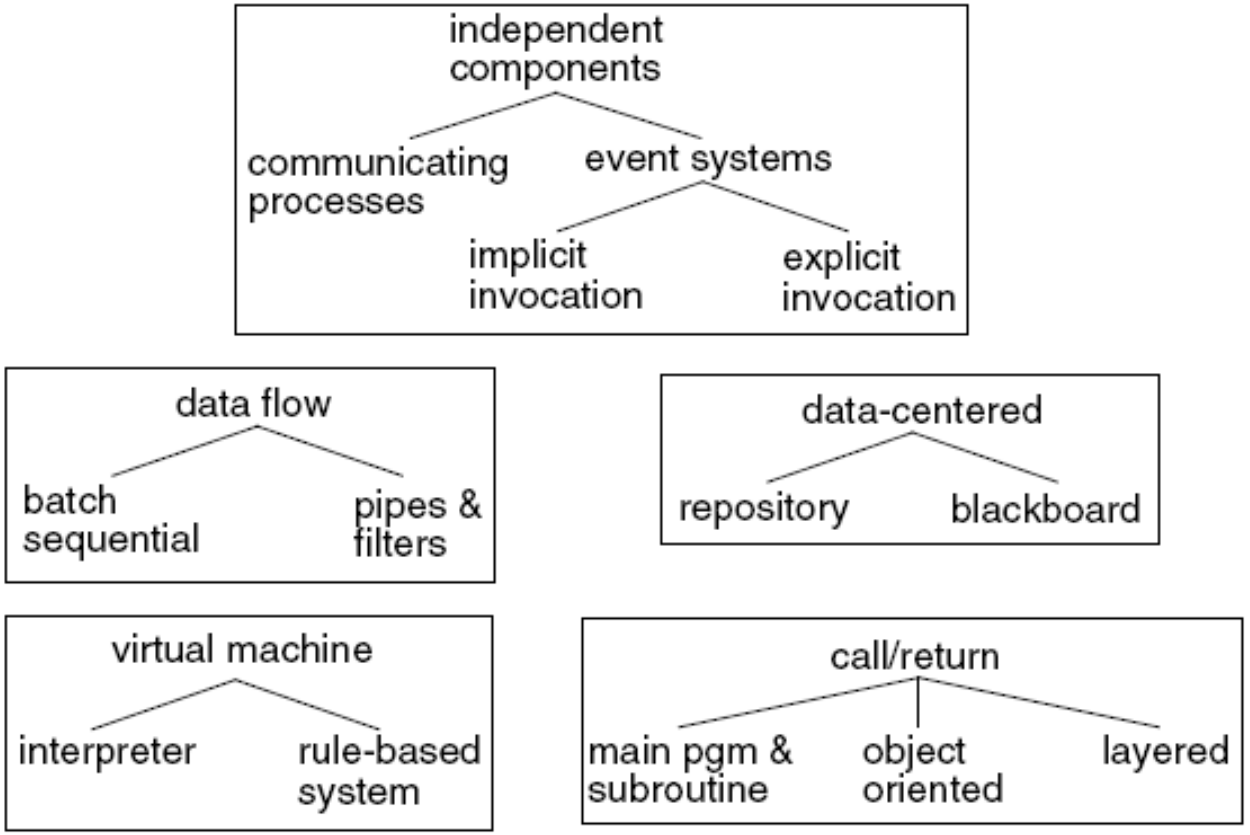
- 软件体系结构风格定义了用于描述系统的术语表和一组指导构建系统的规则。

Style Analysis Dimensions

- What is the design vocabulary *(设计词汇表)*?
  - Component and connector types
- What are the allowable structural pattern?
- What is the underlying computational model?
- What are the essential invariants *(基本不变性)* of the style?
- What are some common examples of its use?
- What are the advantages and disadvantages of using that style?
- What are some of the common specializations?

Taxonomy of Styles

- **Garlan和Shaw给出了通用体系结构风格的分类（a list of common architectural styles）**

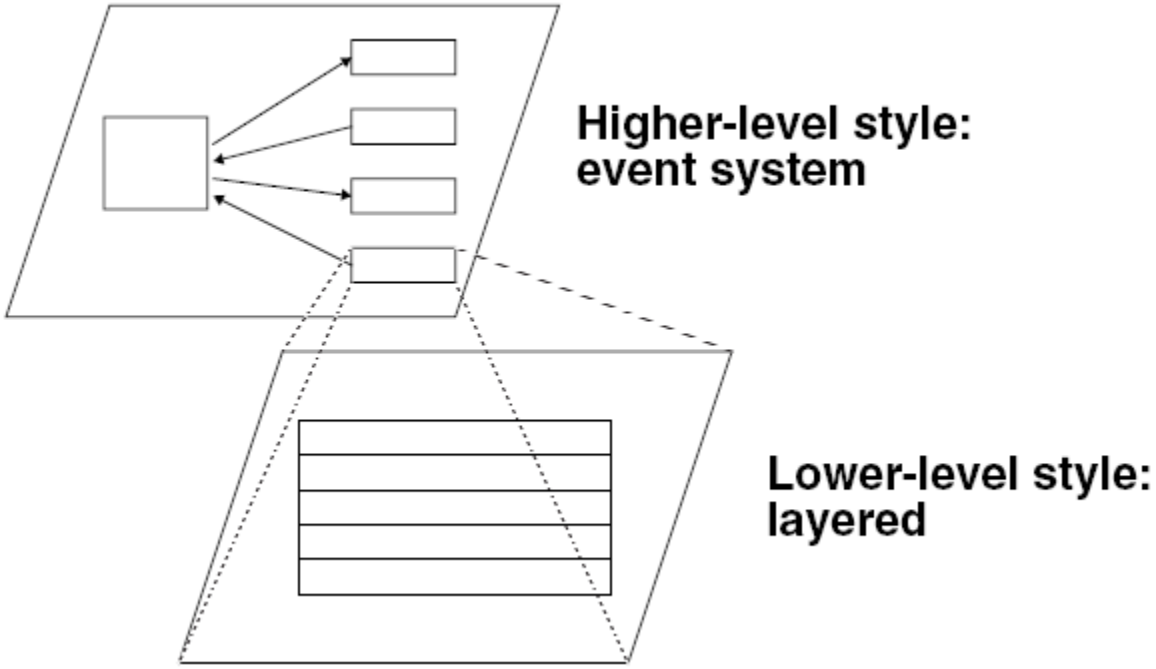


Notes about Architecture Styles

- 特别注意：体系结构风格不是堆软件进行分类的标准，它仅仅是描述软件的不同角度而已
  - 例如，一个系统采用了分层风格，但这并不妨碍它用面向对象的方法来实现。
- **There is no complete list**（*没有完备的列表*）
- **Styles overlap**（*风格是彼此重叠的*）
- **Systems exhibit multiple styles at once**（*一个系统通常表现出多种风格*）
  - *同一个系统采用多种风格形成了所谓体系结构风格的异构组合。*

Heterogeneous(异构) Styles

- A system need not be comprised of only one style



- A single (part of a) system may be viewed as several style *concurrently.*

Agenda

- Taxonomy of style
  - Data Flow
  - Call/Return
  - Data-centered
  - Virtual Machine
  - Independent Component
- Other styles

Data Flow Style

- A data flow system is one in which
  - the availability of data controls the computation（由数据控制计算）
  - the structure of the design is dominated by orderly motion of data from process to process（系统结构由数据在处理之间的有序移动决定）
  - the pattern of data flow is explicit（数据流系统的结构是显而易见的）
- In a pure data flow system, there is no other interaction between processes（在纯数据流系统中，处理之间除了数据交换，没有任何其他的交互）

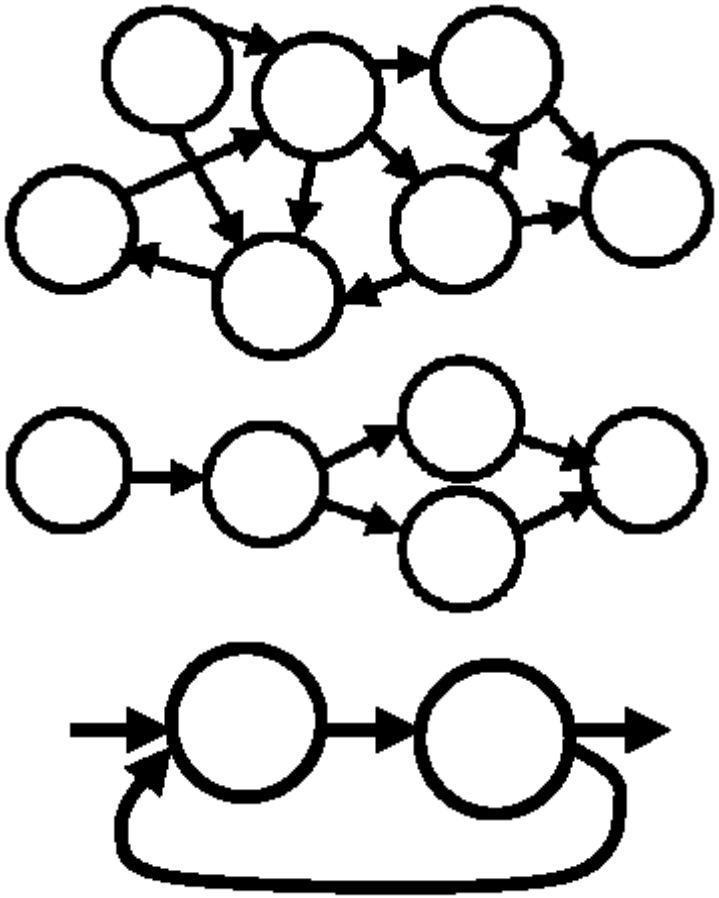
Data Flow Style

- There are variety of variations on this genera theme:
  - how control is exerted (e.g., *push* versus *pull*)（如何施加控制（比如：推还是拉））
  - degree of concurrency between processes（处理之间并行的程度）
  - topology（拓扑结构）

Data Flow Styles

- **Components: Data Flow Components**
  - **Interfaces are input ports and output ports** *（组件接口是输入端口和输出端口）*
  - **Computational model: read data from input ports, compute, write data to output ports** *（计算模型：从输入端读数据，计算，将计算结果写到输出端）*
- **Connectors: Data Streams**
  - **Uni-directional** *（单向）*
    - **usually asynchronous, buffered** *（通常是异步的，有缓冲）*
  - **Interfaces are reader and writer roles**
- **Systems**
  - **Arbitrary graphs** *（任意拓扑结构）*
  - **Computational model: functional composition**

Patterns of Data Flow in Systems



- In general, data can flow in arbitrary patterns（一般情况，数据的流向无序）
- Often we are primarily interested in nearly linear data flow systems（我们主要研究近似线性数据流）
- or in very simple, highly constrained cyclic structures（或者是在限度内的循环数据流）

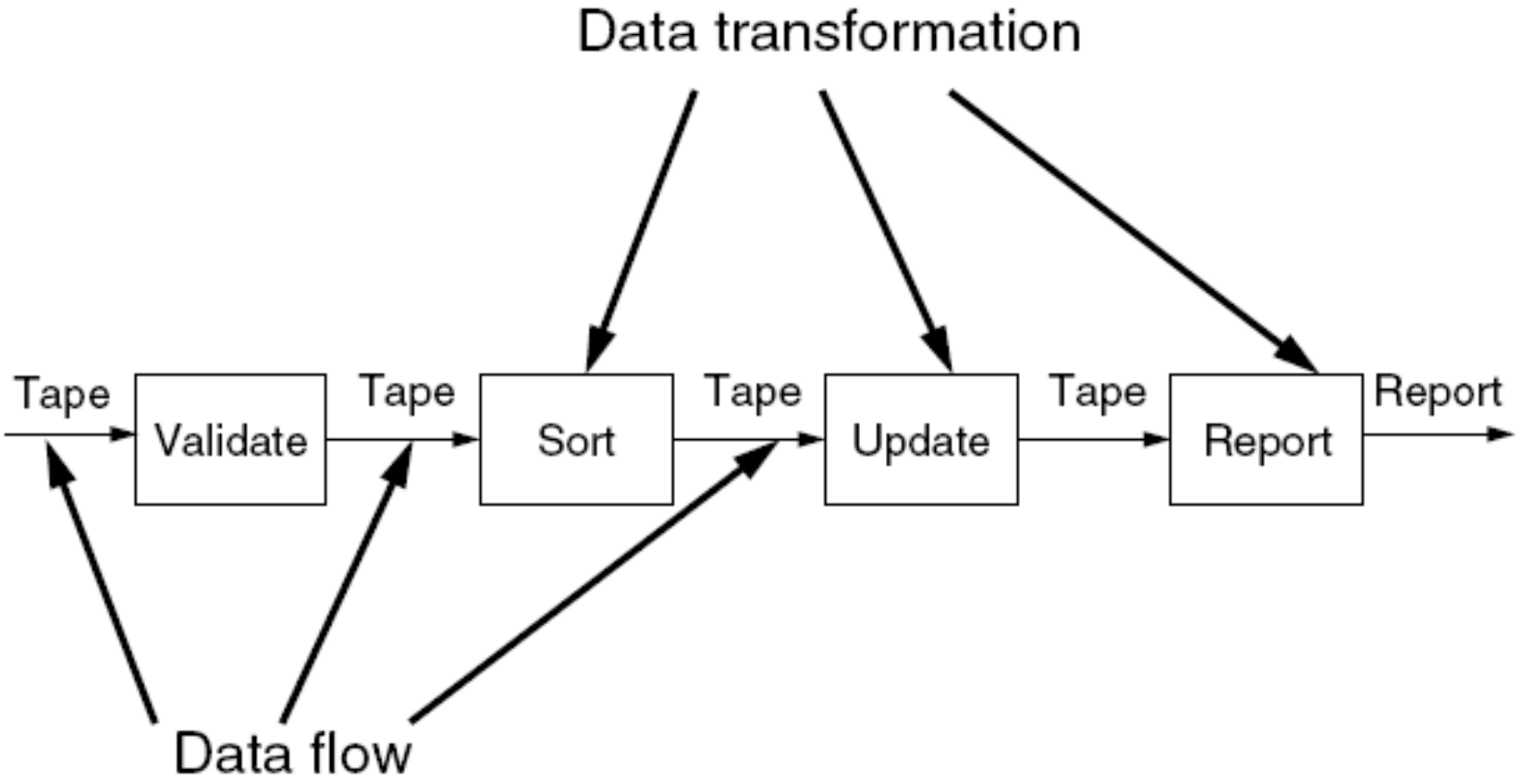
Control Flow vs. Data Flow

- **Control Flow (typical case in procedural systems)**
  - **Dominant question is how locus of control moves through the program** *（主要问题是控制点怎样在程序或系统之间移动）*
  - **Data may accompany the control but is not the driving force** *（数据可能跟着控制走，但是并不起推动系统运转的作用）*
  - **Primary reasoning is about order of computation** *（关注的核心是计算顺序）*
- **Data Flow**
  - **Dominant question is how data moves through a collection of (atomic) computations** *（主要问题是数据怎样在运算单元之间流动）*
  - **As data moves, control is "activated"** *（数据到了，控制（计算）单元便开始工作）*
  - **We reason about data availability, transformations, latency, …** *（我们关心数据是否可用，转换，延时……）*

Three Examples of Data Flow

- **Batch Sequential**（批处理）
- **Pipe-and-Filter**（管道–过滤器）
- **Process Control**（过程控制）

Batch Sequential



Batch Sequential: Model

- Processing steps are independent programs（每个处理步骤是一个独立的程序）
- Each step runs to completion before next step starts（每一步必须在前一步结束后才能开始）
- Data transmitted as a whole between steps（数据必须是完整的，以整体的方式传递）
- Typical applications:
  - classical data processing（传统的数据处理）
  - program compilation（程序编译）/computer aided software engineering

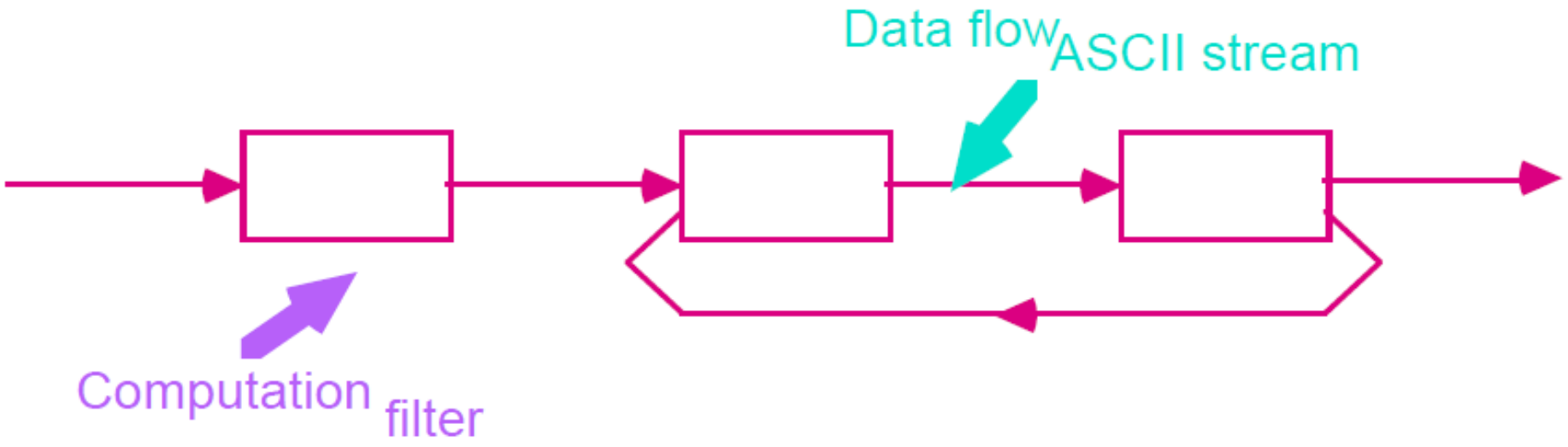
Pipe-and-Filter

- 在管道-过滤器风格下，每个功能模块都有一组输入和输出。功能模块称作过滤器（filter）；功能模块间的连接可以看作输入、输出数据流之间的通路，称作管道（pipe）。
- 管道-过滤器风格的特性之一在于过滤器的相对独立性，即过滤器独立完成自身功能，相互之间无需进行状态交互。

Pipe-and-Filter

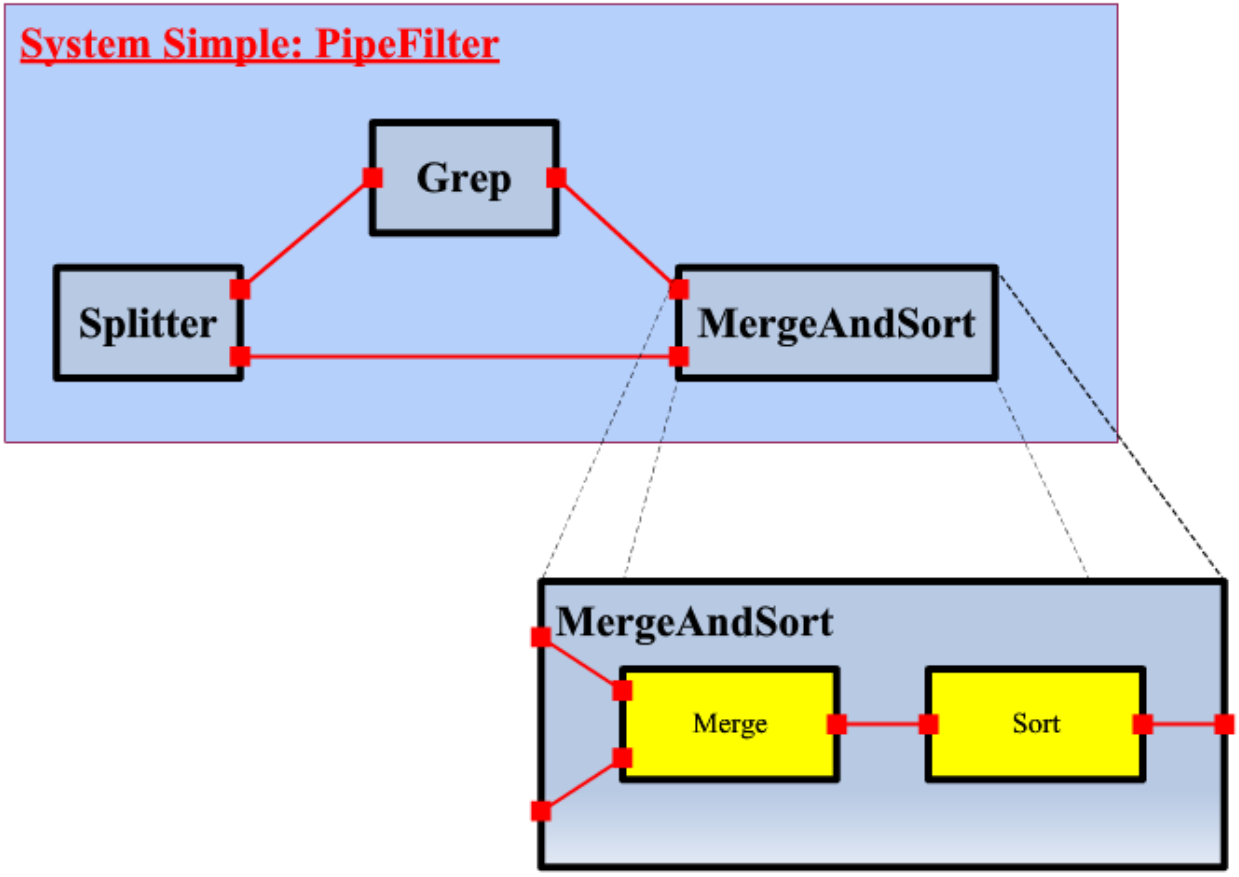
- 过滤器是独立运行的构件
  - 非邻近的过滤器之间不共享状态
  - 过滤器自身无状态
- 过滤器对其处理上下连接的过滤器“无知”
  - 对相邻的过滤器不施加任何限制
- 结果的正确性不依赖于各个过滤器运行的先后次序
  - 各过滤器在输入具备后完成自己的计算。
  - 完整的计算过程包含在过滤器之间的拓扑结构中。

Pipe-and-Filter



Pipe-and-Filter

- 一个采用了嵌套的管道过滤器的系统示例：



Pipe-and-Filter: Examples

- Unix系统中的管道过滤器结构
  - ls -al | grep my
- DOS中的管道命令
  - DOS允许在命令中出现用“|”分开的多个命令，将“|”之前的命令的输出，作为“|”之后命令的输入，这就是“管道功能”，“|”是管道操作符。

Pipe-and-Filter: Examples

- **dir | more**

```
C:\>dir | more
 驱动器 C 中的卷是 System
 卷的序列号是 0006-F8CA

 C:\ 的目录

2012/11/09  11:02             1,024 .rnd
2014/02/20  09:05    <DIR>          alipay
2009/06/11  05:42                24 autoexec.bat
2013/09/16  16:10               644 bar.emf
2009/06/11  05:42                10 config.sys
2013/12/16  14:43                 9 error.txt
2014/02/21  08:30               159 hwsig.log
2014/02/28  13:53    <DIR>          iDownload
2013/03/14  08:25    <DIR>          iNodeLog
2012/10/09  10:32    <DIR>          Intel
2014/03/04  08:19    <DIR>          ksDownloads
2009/07/14  10:37    <DIR>          PerfLogs
2014/02/28  13:53    <DIR>          Program Files
2013/11/12  13:59        49,633,792 setup_force_up.exe
2012/10/09  10:33    <DIR>          SWSETUP
2012/11/08  15:26    <DIR>          tmp
2012/10/09  10:30    <DIR>          Users
2014/03/04  10:20    <DIR>          Windows
               7 个文件     49,635,662 字节
-- More  --
```

Pipe-and-Filter: Advantages

- 使得软件构件具有良好的隐蔽性和高内聚、低耦合的特点
- 设计者可以将整个系统的输入输出特性理解为各个过滤器功能的简单合成
- 支持功能模块的重用：任意两个过滤器只要相互间所传输的数据格式上达成一致，就可以连接在一起
- 系统容易维护和扩展：新的过滤器容易加入到系统中，旧的过滤器也可被改进的过滤器替换
- 支持某些特定属性的分析，如吞吐量和死锁检测
- 支持并发执行，每个过滤器既可以独立运行，也可以与其他过滤器并发执行（并行？并发？）

Pipe-and-Filter: Disadvantages

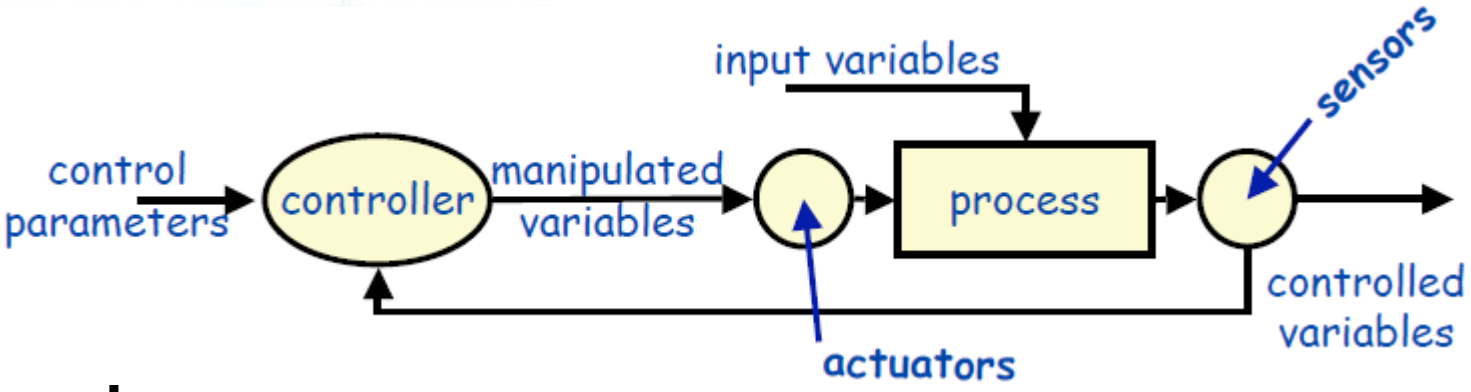
- 由于过滤器的传输特性，这种模式通常不适合于交互性很强的应用。
- 因为在数据传输上没有通用的标准，每个过滤器都增加了解析和合成数据的工作，这会导致系统性能的下降，也会增加编写过滤器的复杂性（数据格式转换与映射）。

Batch Sequential vs Pipe & Filter

- **Both decompose task into fixed sequence of computations, interact only through data passed from one to another.** *（把任务分解成为一系列固定顺序的计算单元(组件)，组件间只通过数据传递交互。）*

Batch Sequential	Pipe/Filter
• total	• incremental
• High latency (real-time is hard)	• Results start immediately
• Random access to input ok	• Processing localized in input
• No concurrency	• Feedback loops possible
• Non-interactive	• Often interactive, awkwardly

Process Control



- **Examples**
  - **Aircraft/spacecraft flight control systems**
  - **Controllers for industrial production lines, power stations**
  - **Chemical engineering**
- **Interesting properties**
  - **Separates control policy from the controlled process**
  - **Handles real-time, reactive computations**
- **Disadvantages**
  - **Difficult to specify the timing characteristics and response to disturbances**

Agenda

- Taxonomy of styles
  - Data Flow
  - Call/Return
  - Data-centered
  - Virtual Machine
  - Independent Component
- Other styles

Call/Return

- Main program and subroutines
  - Classical programming paradigm-functional decomposition
- Object-Oriented/Abstract Data Types
  - Information (representation, access method) hiding
- Layered hierarchies
  - Each level only communicates with its immediate neighbors
- Other
  - Client-server
  - ……

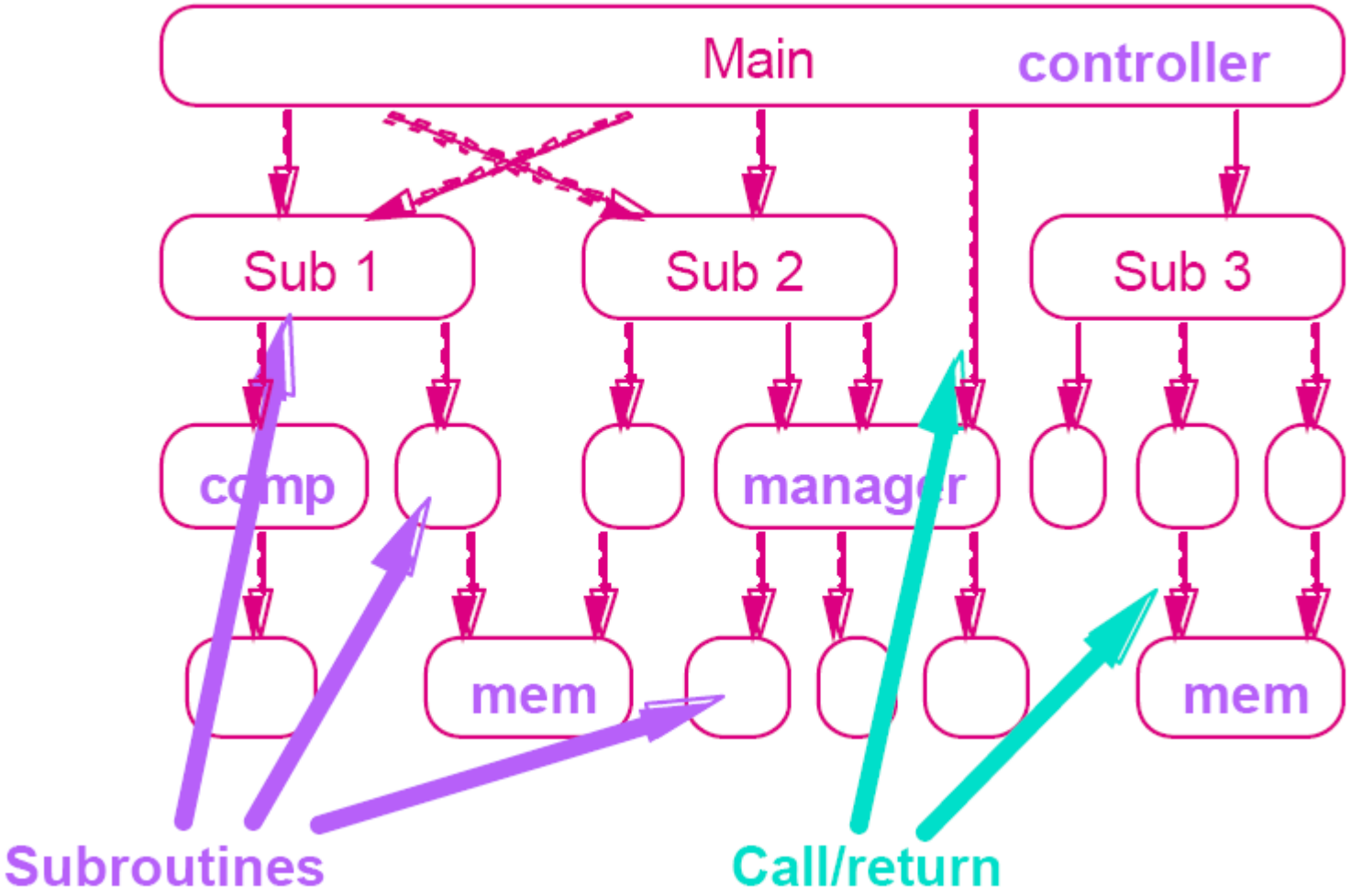
History

- **Main program and subroutines**
  - **Decomposition into processing steps with single-threaded control（*单线程控制，划分为若干处理步骤*）**
- **Functional modules**
  - **Aggregation of processing steps into modules（*把处理步骤集成到模块内*）**
- **Abstract Data Types**
  - **Bundle operations and data, hide representations and other secrets（*操作和数据捆绑在一起，隐藏实现和其他秘密*）**

History

- **Objects**
  - **Methods (bound dynamically), polymorphism (subtypes), reuse (through inheritance)（方法（动态绑定），多态（子类），重用（继承））**
- **OO Architectures**
  - **Objects as separate processes/threads（对象作为不同的进程/线程）**
  - **Client-server, tiered styles**
- **Components**
  - **Multiple interfaces, binary compatibility, advanced middleware （多个接口，二进制兼容，高级中间件）**

Main Program and Subroutine



Main Program and Subroutine

- **Problem:**
  - **This pattern is suitable for applications in which the computation can appropriately be defined via a hierarchy of procedure definitions.**
- **Context:**
  - **Many programming languages provide natural support for defining nested collections of procedures and for calling them hierarchically. These languages often allow collections of procedures to be grouped into modules, thereby introducing *name-space* locality. The execution environment usually provides a single thread of control in a single name space.**

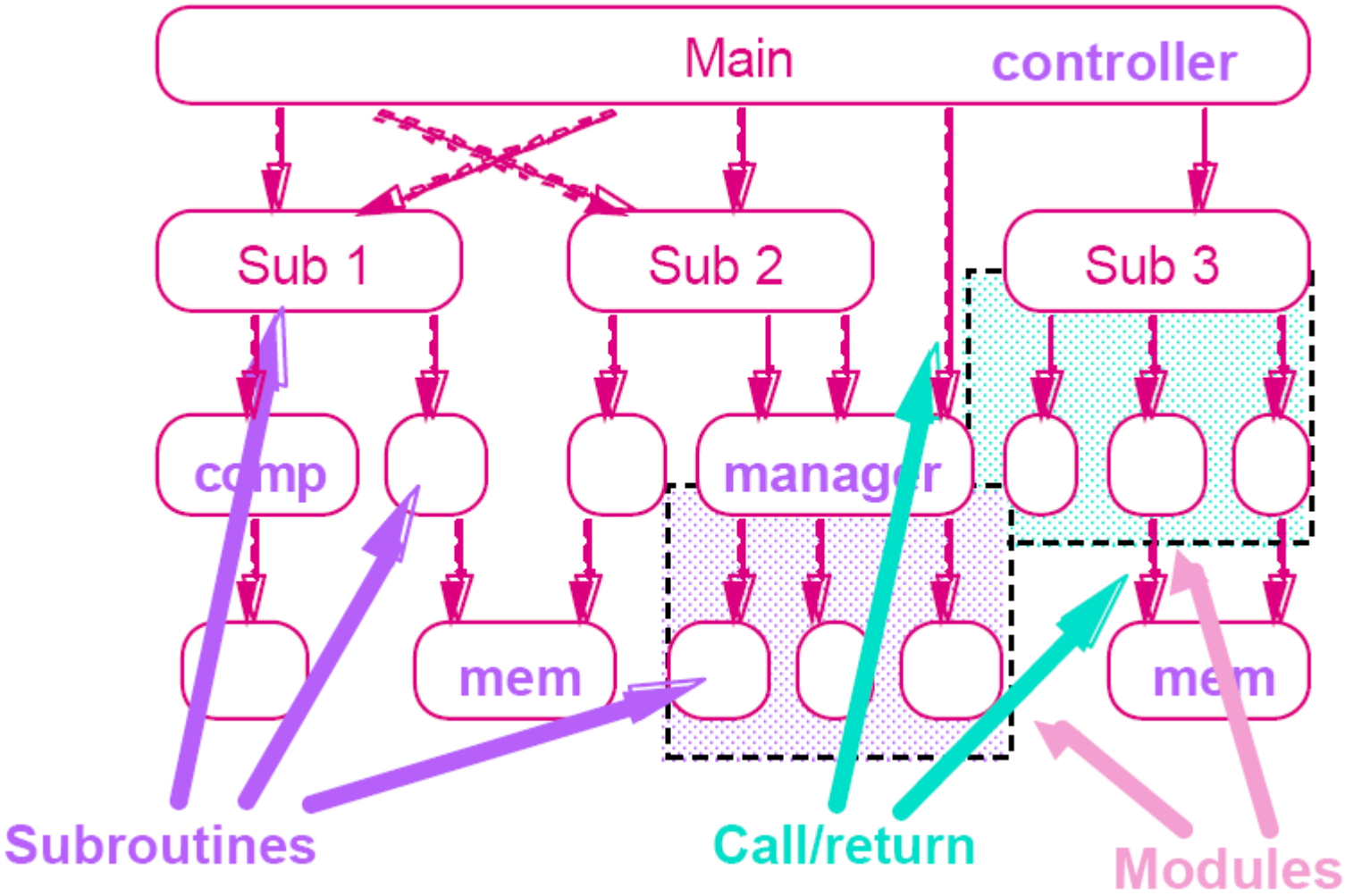
Main Program and Subroutine

- **Solution:**
  - ***System model:*** **call and definition hierarchy, subsystems often defined via modularity**
  - ***Components:*** **procedures and explicitly visible data**
  - ***Connectors:*** **procedure calls and explicit data sharing**
  - ***Control structure:*** **single thread**

Main Program and Subroutine: Model

- **Hierarchical decomposition:**
  - **Based on definition-use relationship**
  - **Uses procedure call as interaction mechanism**
- **Single thread of control:**
  - **Supported directly by programming languages**
- **Hierarchical reasoning:**
  - **Correctness of a subroutine depends on the correctness of the subroutines it calls**
- **Subsystem structure implicit:**
  - **Subroutines typically aggregated into modules**

Main Program and Subroutine



Pipes vs Procedures

	Pipes	Procedures
Control	Asynchronous, data-driven	Synchronous, blocking
Semantics	Functional	Hierarchical
Data	Streamed	Parameter / return value
Variations	Buffering, end-of-file behavior	Binding time, exception handling, polymorphism

Criteria for Modularization

- What is a module?
  - Common view: a piece of code. But too limited.
  - Compilation unit, including related declarations and interface
  - Parnas: a unit of work.
- Why modularize a system, anyway?
  - Management: Partition the overall development effort
    - divide and conquer （分而治之）
  - Evolution: Decouple parts of a system so that changes to one part are isolated from changes to other parts
  - Understanding: Permit system to be understood as composition of mind-sized chunks
- Key issue: what criteria to use for modularization

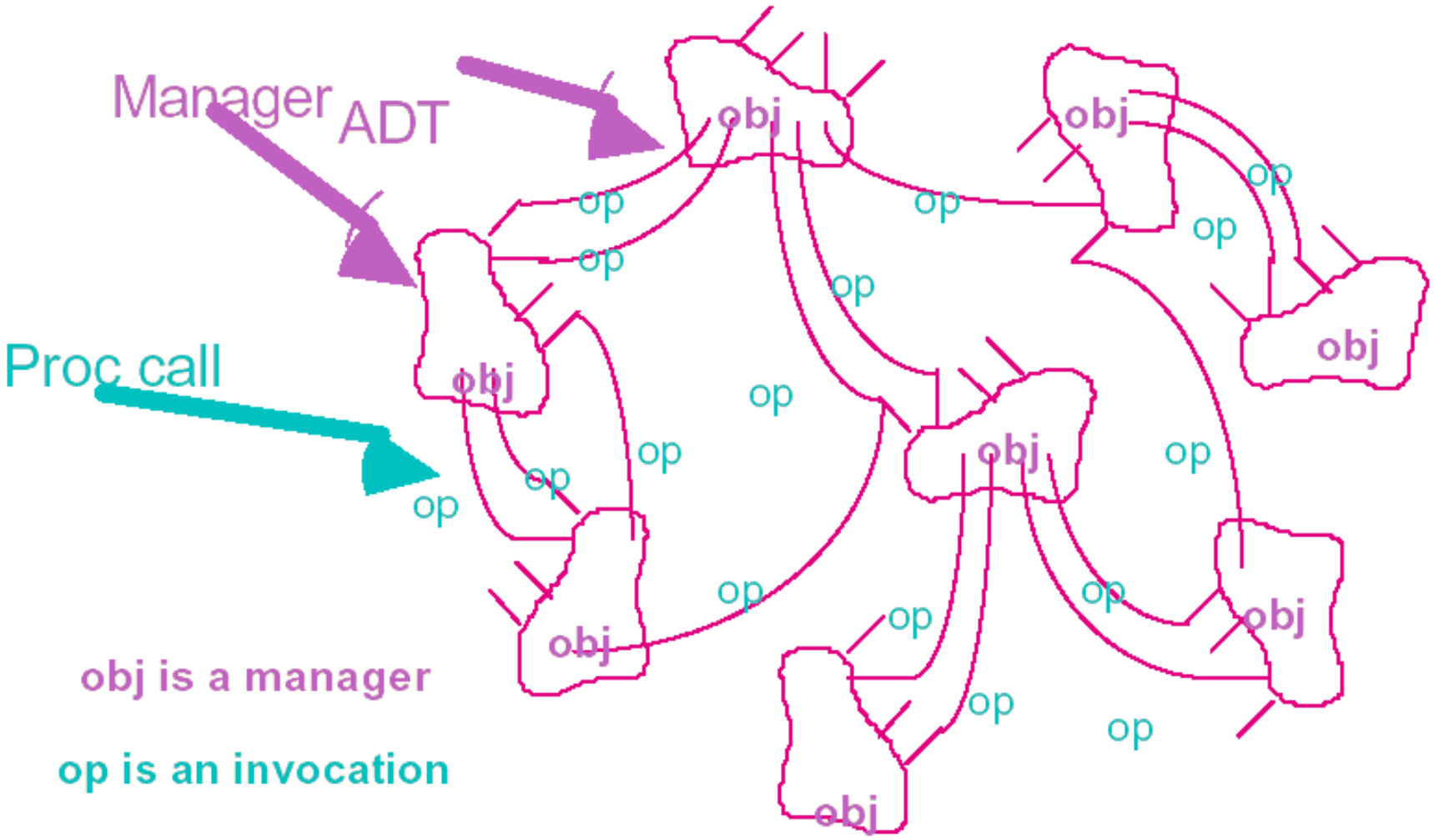
Module Decomposition

- Parnas
  - Hide secrets. OK, what's a "secret"?
    - Representation of data
    - Properties of a device, other than required properties
    - Mechanisms that support policies
  - Try to localize future change
    - Hide system details likely to change independently
    - Expose in interfaces assumptions unlikely to change
  - Use functions to allow for change
    - They're easier to change than visible representation

Encapsulation/Information Hiding

- Parnas: Hide secrets (not just representations)
- Booch: Object's behavior is characterized by actions that it suffers and that it requires
- Practically speaking:
  - Object has state and operations, but also has responsibility for the integrity of its state
  - Object is known by its interface
  - Object is probably instantiated from a template
  - Object has operations to access and alter state and perhaps generator
  - There are different kinds of objects (e.g., actor, agent, server)

Data Abstraction or Object-Oriented



Data Abstraction or Object-Oriented

- **Problem: This pattern is suitable for applications in which a central issue is identifying and protecting related bodies of information, especially representation information.**
- **Context: Numerous design methods provide strategies for identifying natural objects. Newer programming languages support various variations on the theme, so if the language choice or the methodology is fixed, that will strongly influence the flavor of the decomposition.**
- **Solution:**
  - ***System model:*** **localized state maintenance**
  - ***Components:*** **managers (e.g., servers, objects, abstract data types)**
  - ***Connectors:*** **procedure call**
  - ***Control structure:*** **decentralized, usually single thread**

Elements of Object Architectures

- ***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*: Shared definitions of functionality** *（继承：对共享的功能保持唯一的接口）*
- ***Reuse and maintenance*: Exploit encapsulation and locality to increase productivity** *（重用和维护：利用封装和聚合提高生产力）*

Problems with Object Approaches

- **Managing many objects**
  - **vast sea of objects requires additional structuring** *（对象的海洋需要额外的结构来容纳）*
  - **hierarchical design suggested by Booch and Parnas**
- **Managing many interactions**
  - **single interface can be limiting & unwieldy (hence, “friends”)** *（单一的接口能力有限并且笨拙（于是，“友元”））*
  - **some languages/systems permit multiple interfaces (inner class, interface, multiple inheritance)**
- **Distributed responsibility for behavior**
  - **makes system hard to understand**
  - **interaction diagrams now used in design**
- **Capturing families of related designs**
  - **types/classes are often not enough**
  - **design patterns as an emerging off-shoot**

Managing Large Object Sets

- Pure O-O design leads to large flat systems with many objects
  - Same old problems can reappear
  - Hundreds of modules => hard to find things
  - Need a way to impose structure
- Need additional structure and discipline
- Structuring options
  - Layers (which are not necessarily objects)
  - Supplemental index
  - Hierarchical decomposition: big objects and little objects

Next Class

- **Architecture Style**
  - **Call/Return**
  - **Data centered**
  - **Virtual machine**
  - **Independent component**