

# 软件架构与中间件



涂志莹

[tzy\\_hit@hit.edu.cn](mailto:tzy_hit@hit.edu.cn)

哈尔滨工业大学

苏统华

[thsu@hit.edu.cn](mailto:thsu@hit.edu.cn)

# 软件架构与中间件

## Software Architecture and Middleware

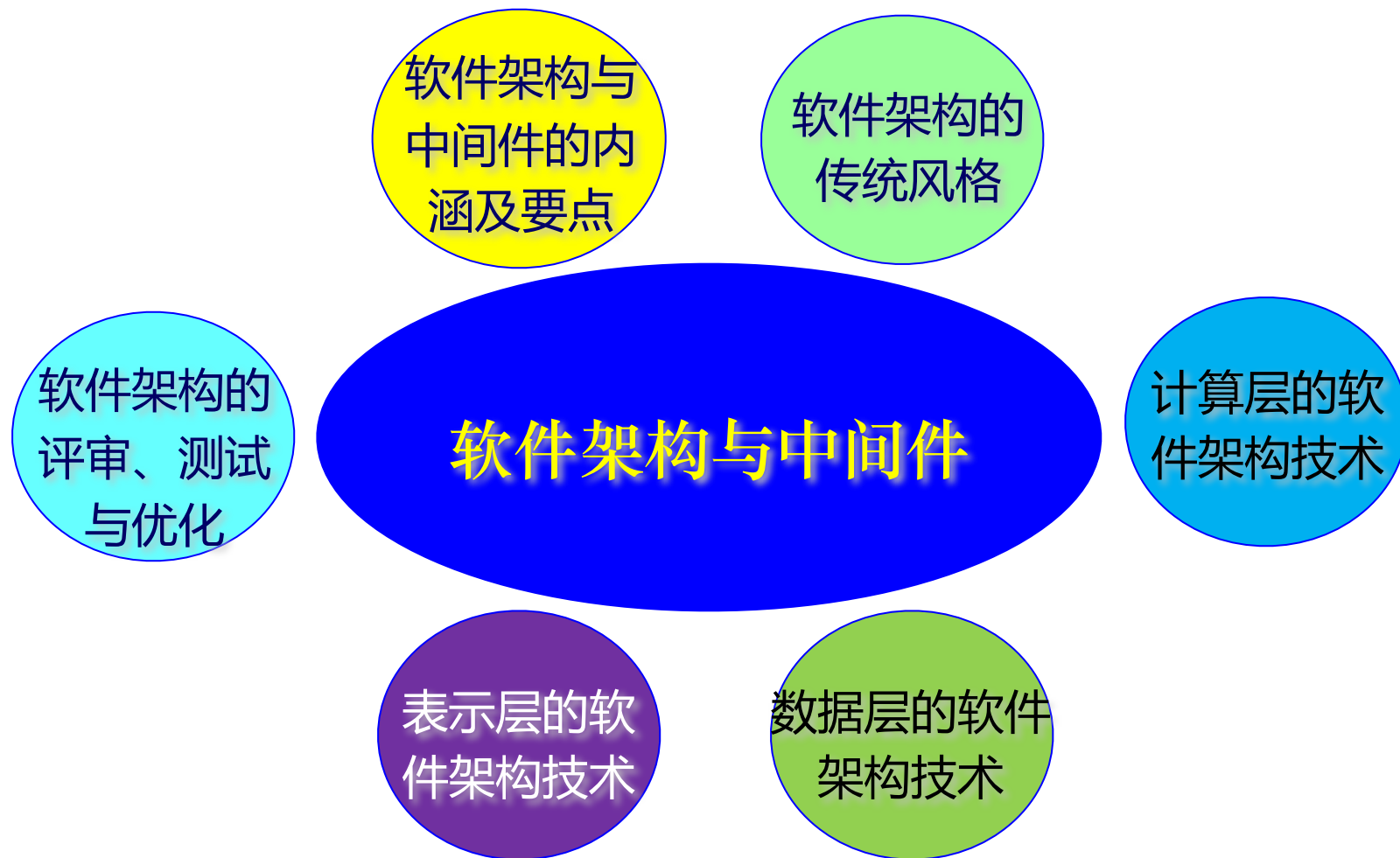


## 第2章

# 软件架构的传统风格



# 课程内容



# 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**

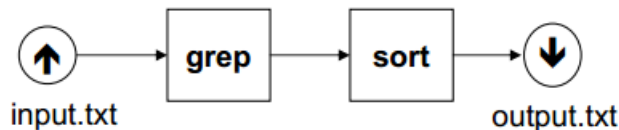
# 2.4

## 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

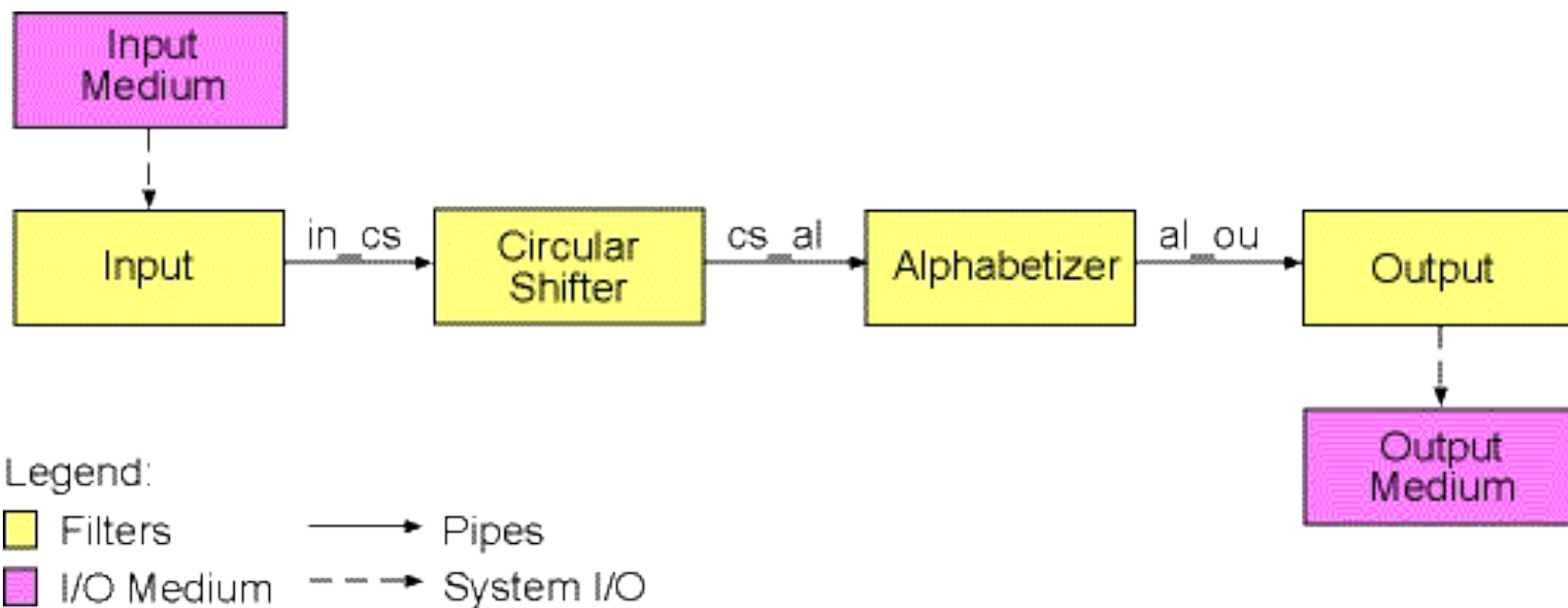
## 2.4.3 管道-过滤器风格应用

- 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`





- Uses a pipeline solution. (使用线性管道-过滤器风格)
  - There are four filters: input, shift, alphabetize, and output. (四个过滤器：输入、移位、排序、输出)
  - Each filter processes the data and sends it to the next filter. (每个过滤器处理数据，然后将结果送至下一个过滤器)
  - Control is distributed: each filter can run whenever it has data on which to compute. (控制机制是分布式的：只要有数据传入，过滤器即开始工作)
  - Data sharing between filters is strictly limited to that transmitted on pipes. (过滤器之间的数据共享被严格限制在管道传输)

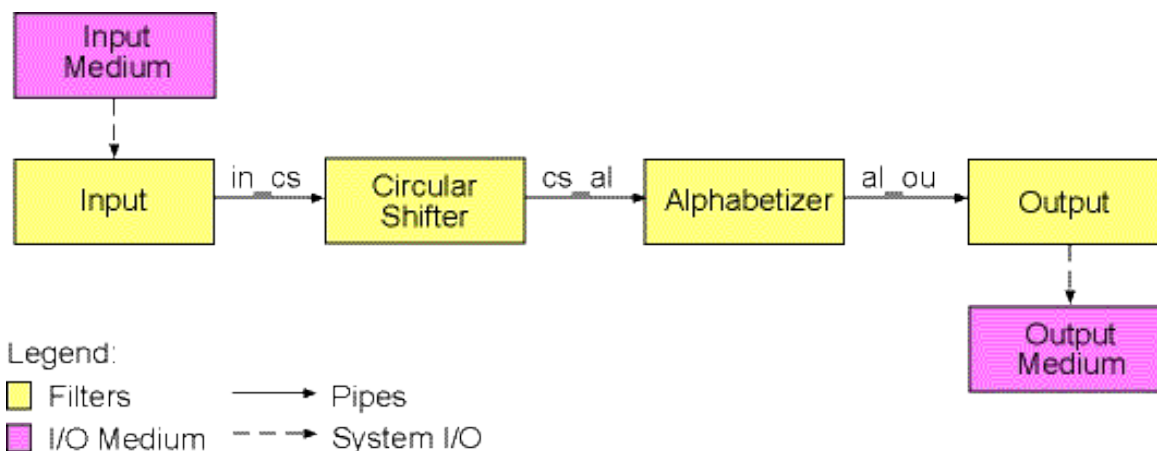


- Filters :

- Input filter ( “输入” 过滤器：从数据源读取输入文件，解析格式，将行写入输出管道)
- CircularShifter filter ( “循环移位” 过滤器)
- Alphabetizer filter ( “排序” 过滤器)
- Output filter ( “输出” 过滤器)

- Pipe :

- in\_cs pipe
- cs\_al pipe
- al\_ou pipe



## **Pipe**

**-reader\_: PipedReader**  
**-writer\_: PipedWriter**

**+read(out c:int)**  
**+write(c:int)**  
**+closeReader()**  
**+closeWriter()**

```
import java.io.PipedReader;  
import java.io.PipedWriter;  
import java.io.IOException;  
  
public class Pipe {  
    private PipedReader reader_;  
    private PipedWriter writer_  
  
    public Pipe() throws IOException {  
        writer_ = new PipedWriter();  
        reader_ = new PipedReader();  
        // let this piped writer to be connected to the piped reader  
        writer_.connect(reader_);  
    }  
}
```

```
// Writes char to the piped output stream
public void write(int c) throws IOException {
    writer_.write(c);
}

// Reads the next character of data from this piped stream
public int read() throws IOException {
    return reader_.read();
}

public void closeWriter() throws IOException {
    writer_.flush();
    writer_.close();
}

public void closeReader() throws IOException {
    reader_.close();
}
}
```

<i><b>Filter</b></i>
<b>input_:Pipe</b> <b>output_:Pipe</b>
<b>Filter()</b> <b>start()</b> <b>run()</b> <b>stop()</b> <i><b>transform()</b></i>

```
public abstract class Filter implements Runnable {  
    protected Pipe input_;  
    protected Pipe output_;  
    private boolean is_started_ = false;  
  
    public Filter(Pipe input, Pipe output) {  
        input_ = input;  
        output_ = output;  
    }  
    public void start() {  
        if (!is_started_) {  
            is_started_ = true;  
            Thread thread = new Thread(this);  
            thread.start();  
        }  
    }  
    public void stop() {  
        is_started_ = false;  
    }  
}
```



```
public void run() {  
    transform();  
}
```

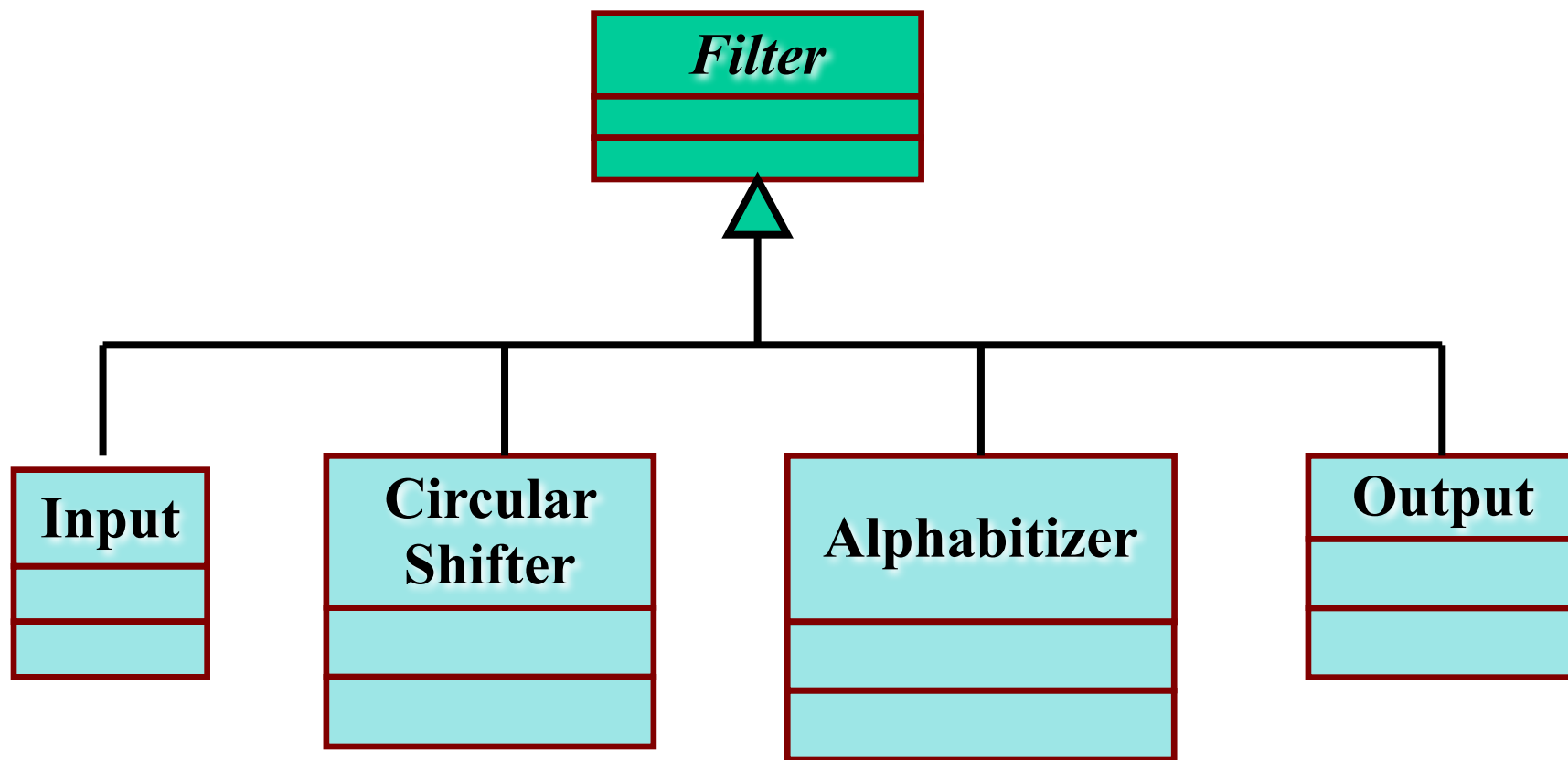
```
/*
```

```
 * This method transforms the data from the input pipe and writes the  
 * transformed data into output pipe.
```

```
*/
```

```
abstract protected void transform();
```

```
}
```



## KWIC Solution : Pipes and Filters

...

```
FileInputStream in = new FileInputStream(file);
```

```
Pipe in_cs = new Pipe(); // create three objects of Pipe
```

```
Pipe cs_al = new Pipe(); // and one object of type
```

```
Pipe al_ou = new Pipe(); // FileInputStream
```

```
Input input = new Input(in, in_cs);
```

```
CircularShifter shifter = new CircularShifter(in_cs, cs_al);
```

```
Alphabetizer alpha = new Alphabetizer(cs_al, al_ou);
```

```
Output output = new Output(al_ou); // output to screen
```

```
input.start();
```

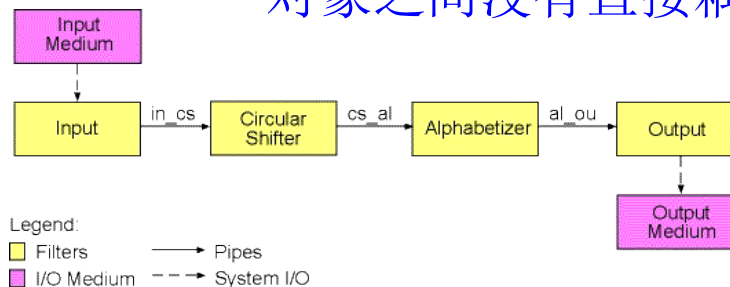
```
shifter.start();
```

```
alpha.start();
```

```
output.start();
```

...

对象之间没有直接耦合



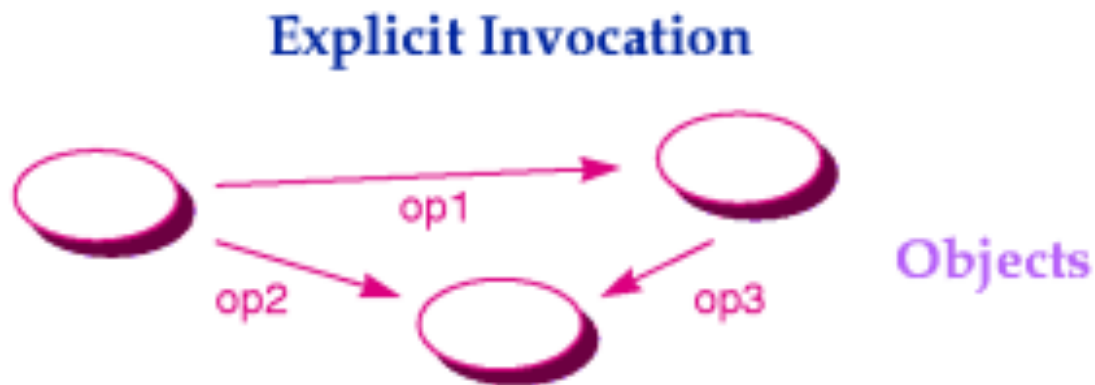
# 2.5 事件系统风格

- 1、事件系统风格概述
- 2、事件调度策略
- 3、事件系统风格应用

## 2.5.1 事件系统风格概述

# Explicit Invocation (显式调用)

- Traditionally, when the components provide a collection of routines and functions, such as an object-oriented system, components typically interact with each other by explicitly invoking those routines.
  - 各个构件之间的互动是由显性调用函数或过程完成的。
  - 调用的过程与次序是固定的、预先设定的。



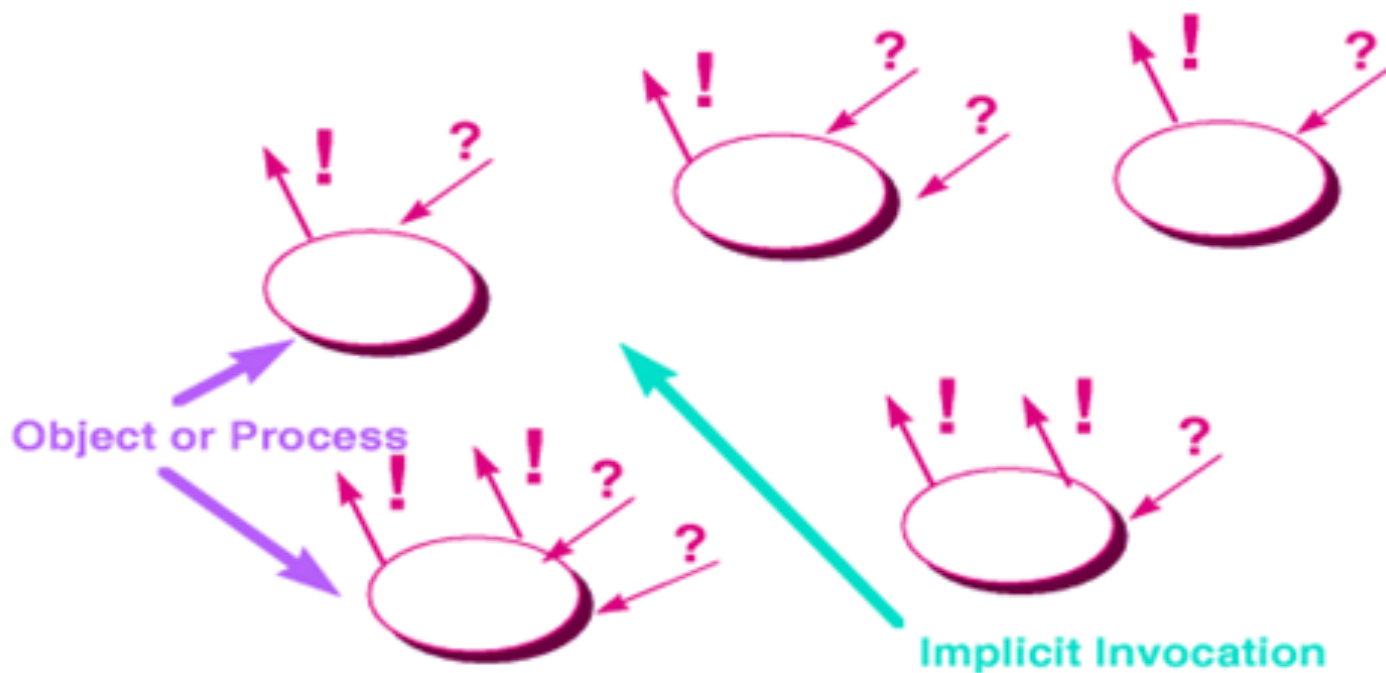
# Implicit Invocation(隐式调用)

- Garland and Shaw: The idea behind **implicit invocation** is that instead of invoking a procedure directly.
  - A component(**Event Source**) can announce (or broadcast) one or more events一个组件可以广播一些事件。
  - Other components(**Event Handlers**) in the system can register an interest in an event by associating a procedure with it 系统中的其它构件可以注册自己感兴趣的事件，并将自己的某个过程与相应的事件进行关联。
  - When the event is announced the system (**Event Manager**) itself invokes all of the procedures that have been registered for the event当一个事件被发布，系统自动调用在该事件中注册的所有过程。
- Thus an event 'implicitly' causes the invocation of procedures in other modules.



# 事件系统风格特点

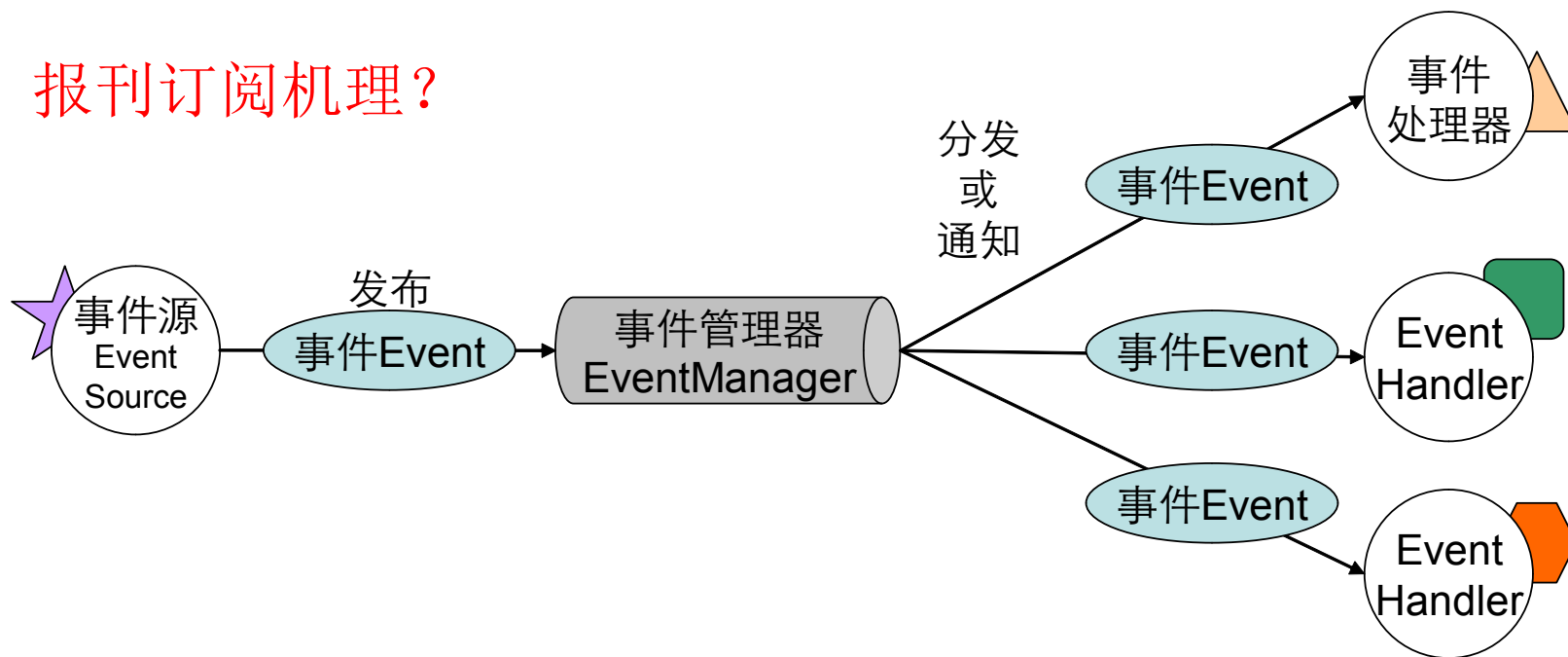
- 这种风格的主要特点是：事件的触发者并不知道哪些构件会被这些事件影响，相互保持独立
  - 这样不能假定构件的处理顺序，甚至不知道哪些过程会被调用
  - 各个构件之间彼此之间无连接关系，各自独立存在，通过对事件的发布和注册实现关联






特点	描述
分离的交互	事件发布者并不会意识到事件订阅者的存在。
一对多通信	采用发布/订阅消息传递，一个特定事件可以影响多个订阅者。
基于事件的触发器	由事件触发过程调用。
异步	支持异步操作。

## 报刊订阅机理？



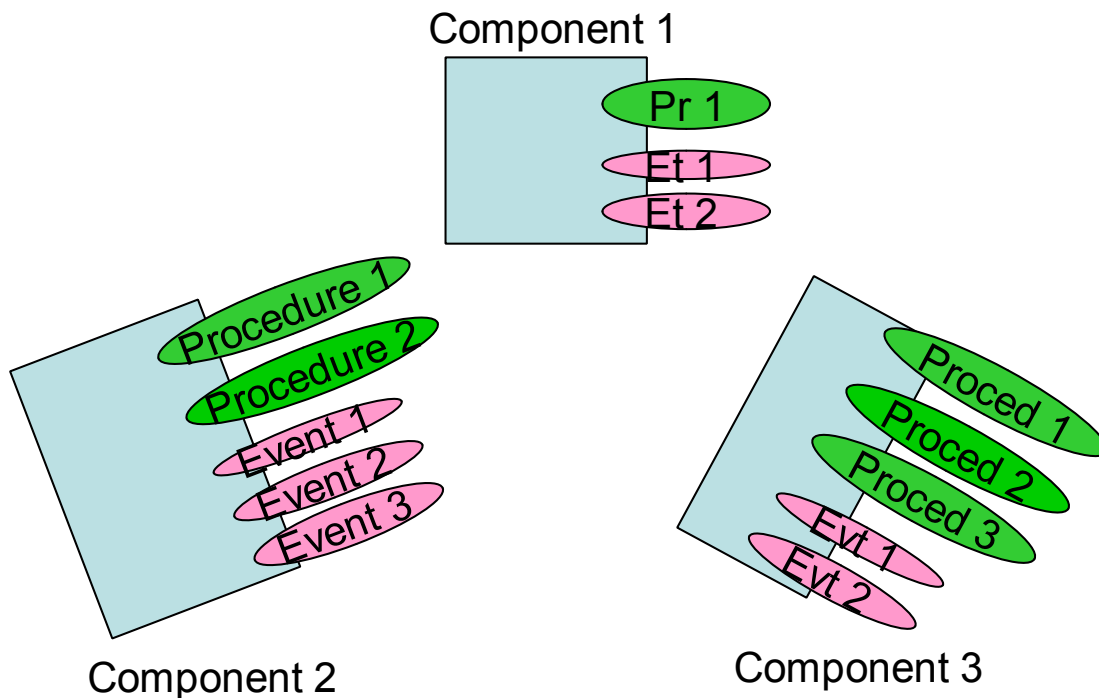
# 事件系统的基本构成与工作原理

<b>Class: Event</b> 	
<b>Responsibility</b>	<b>Collaborators:</b>
<ul style="list-style-type: none"> <li>• Encapsulates details of an event (reason, time, target, location, context)</li> <li>• Classifies event</li> </ul>	
<b>Class: EventManager</b>	
<b>Responsibility</b>	<b>Collaborators:</b>
<ul style="list-style-type: none"> <li>• Decouples EventSource from EventHandler</li> <li>• Synchronizes Events</li> <li>• Coordinates EventHandlers</li> </ul>	<ul style="list-style-type: none"> <li>• EventSource</li> <li>• EventHandler</li> <li>• Event</li> </ul>
<b>Class: EventSource</b>	
<b>Responsibility</b>	<b>Collaborators:</b>
<ul style="list-style-type: none"> <li>• Collects Event details</li> <li>• Delivers Events</li> </ul>	<ul style="list-style-type: none"> <li>• EventManager</li> <li>• Event</li> </ul>
<b>Class: EventHandler</b>	
<b>Responsibility</b>	<b>Collaborators:</b>
<ul style="list-style-type: none"> <li>• Expresses interest in (a class of) Events</li> <li>• Accepts Events or rejects Events</li> </ul>	<ul style="list-style-type: none"> <li>• EventManager</li> <li>• Event</li> </ul>

*sometimes called  
EventTarget*

# 事件系统的基本构件

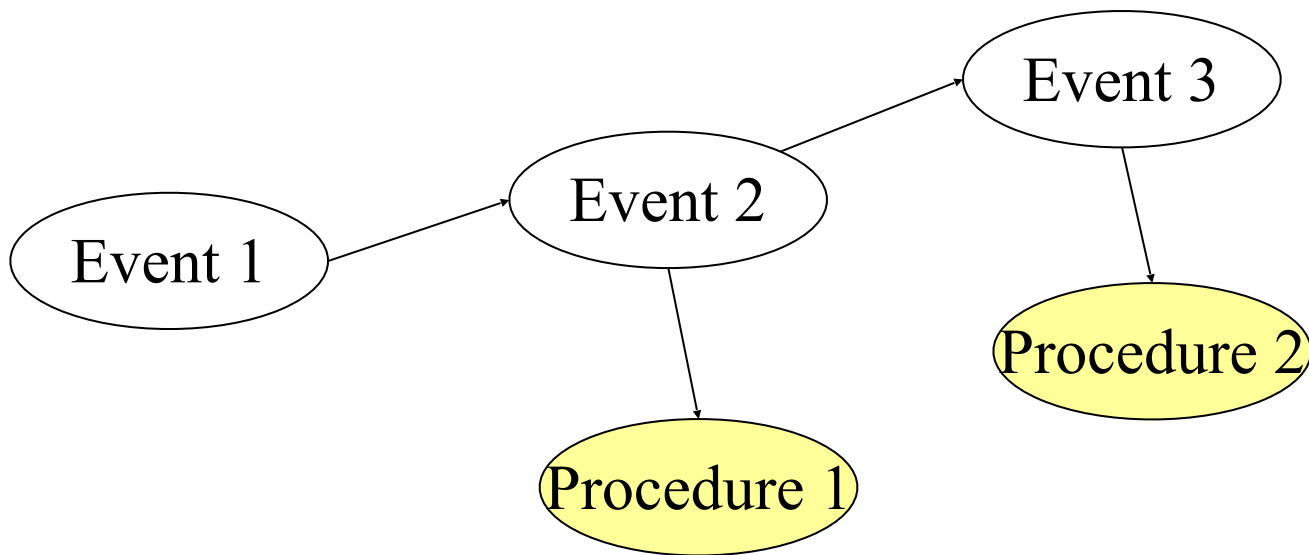
- Components: objects or processes whose interfaces provide both (构件：对象或过程，并提供如下两种接口)
  - a collection of procedures (过程或函数，充当事件源或事件处理器的角色)
  - a set of events (事件).



- Connectors: event-procedure bindings (连接器: 事件-过程绑定)
  - Procedures are registered with events (事件处理器 (事件的接收和处理方) 的过程向特定的事件进行注册)
  - Components communicate by announcing events at “appropriate” times (事件源构件发布事件)
  - when an event is announced the associated procedures are (implicitly) invoked (当某些事件被发布时, 向其注册的过程被隐式调用)
  - Order of invocation is non-deterministic (调用的次序是不确定的)

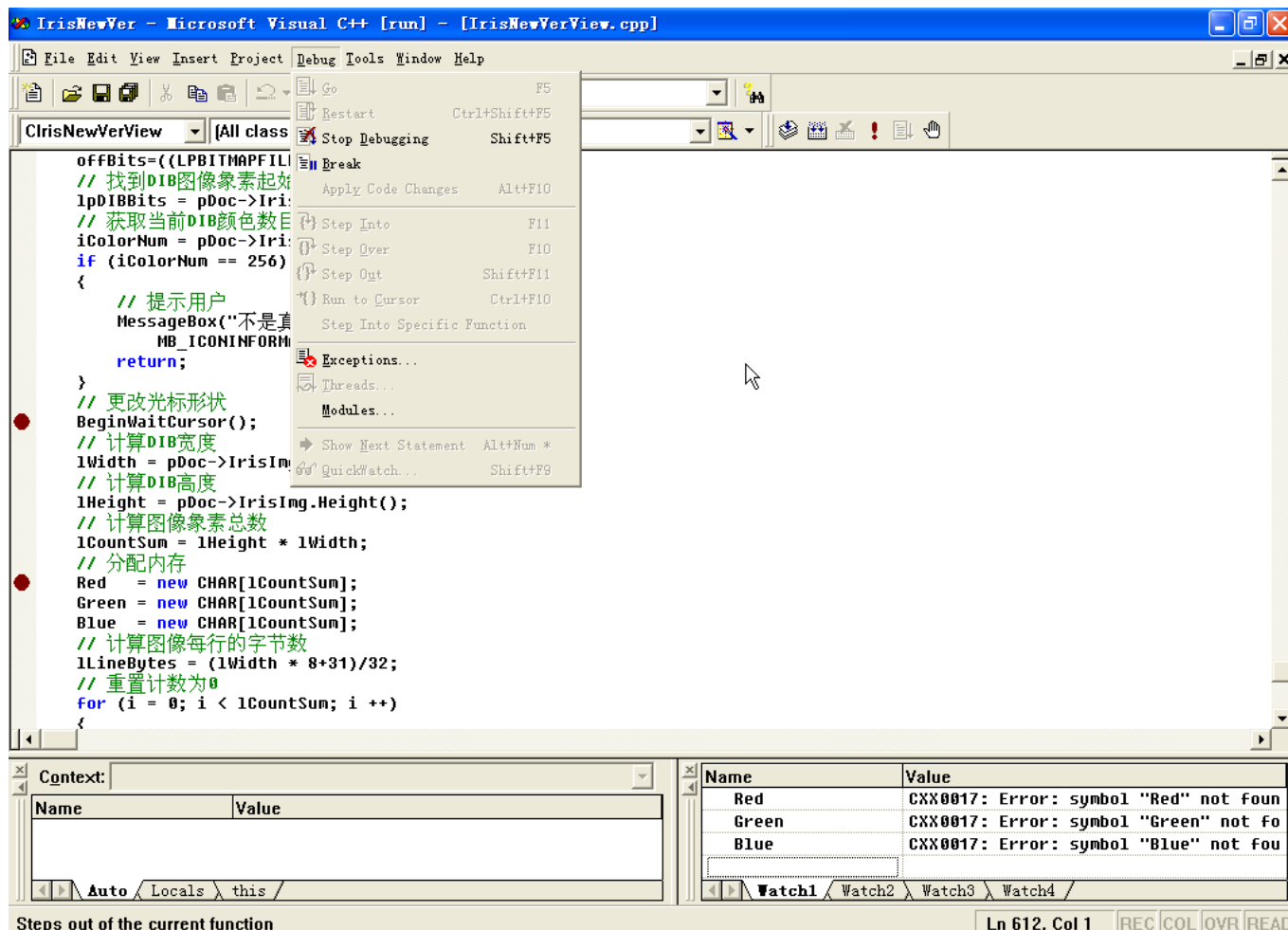
# 事件系统的连接机制

- In some treatments, connectors are event-event bindings (在某些情况下, 连接件可以是事件-事件的绑定).



一个事件也可能触发其他事件, 形成事件链

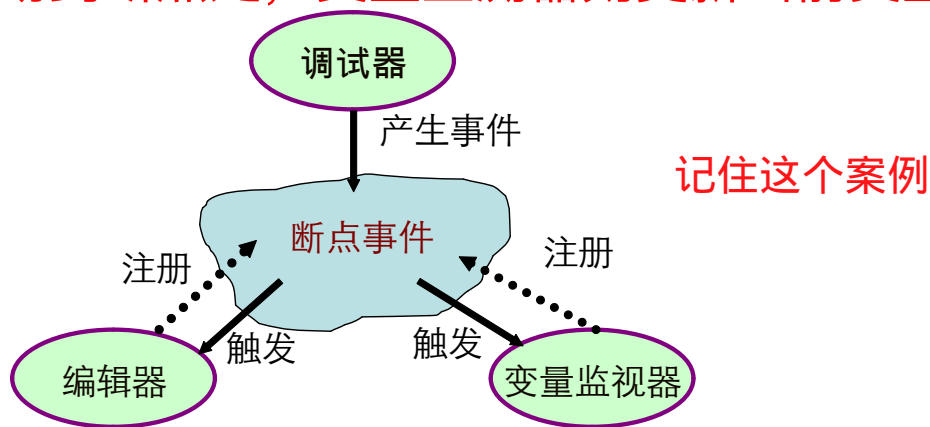
## 思考：调试器中的断点处理？



遇到断点，编辑器将源代码滚动到断点处，变量监测器则更新当前变量值并显示出来。怎么做到的呢？

# 调试器的例子

- EventSource: debugger (调试器)
- EventHandler: editor and variable monitor (编辑器与变量监视器)
- EventManager: IDE (集成开发环境)
- 编辑器与变量监视器向调试器注册, 接收“断点事件”;
- 一旦遇到断点, 调试器发布事件, 从而触发“编辑器”与“变量监测器”;
- 编辑器将源代码滚动到断点处, 变量监测器则更新当前变量值并显示出来。



## 2.5.2 事件调度策略

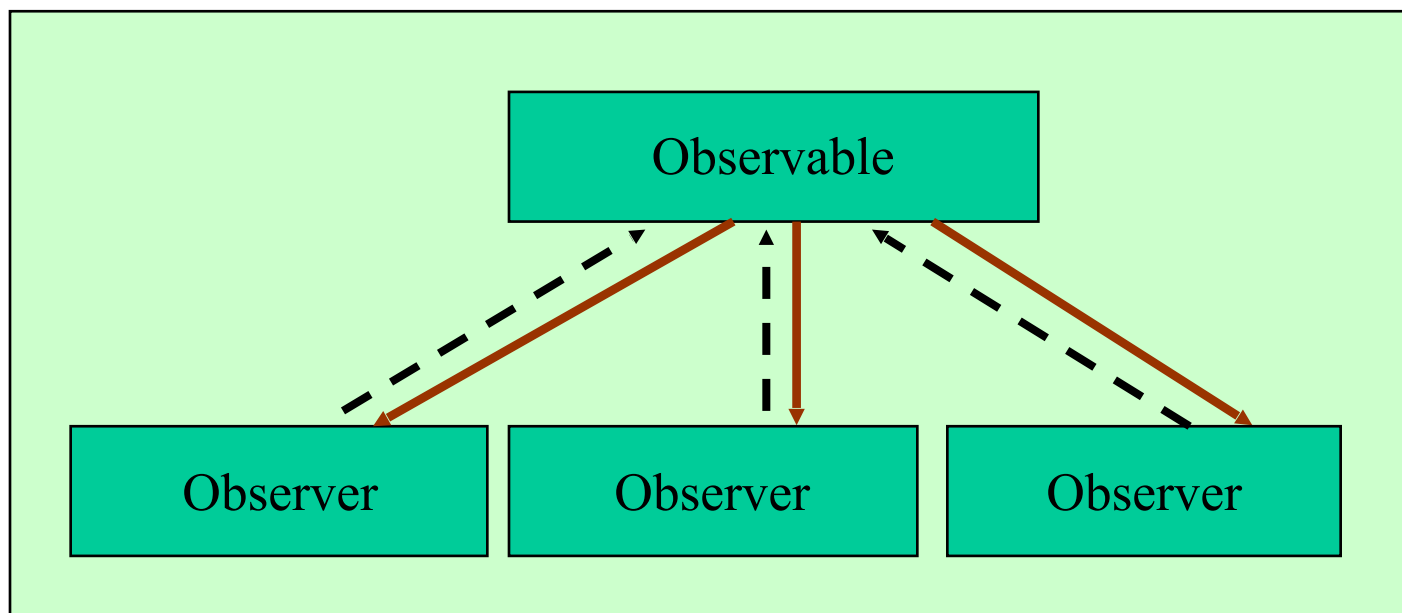


- When an event is announced, the system itself automatically invokes all of the procedures that have been registered for that event. (当事件发生时，已向此事件注册过的所有过程被激发并执行)
- How to make events dispatched to registered components in the system? (问题：事件如何被分发到已注册的模块？)
- 两种调度思想
  - EventManager without a central dispatcher module (无独立（非集中式）调度模块的事件管理器)
  - EventManager with separated dispatcher module (带有独立调度模块的事件管理器)

# 无独立调度模块的事件系统

- This module is usually called Observable/Observer (称为 “被观察者/观察者” ).
- Each module allows other modules to declare interest in events that they are sending. (每一个模块都允许其他模块向自己所能发送的某些事件表明兴趣)
- Whenever a module sends an event it sends that event to exactly those modules that registered interest in that event. (当某一模块发出某一事件时，它自动将这些事件发布给那些曾经向自己注册过此事件的模块)

# 无独立调度模块的系统



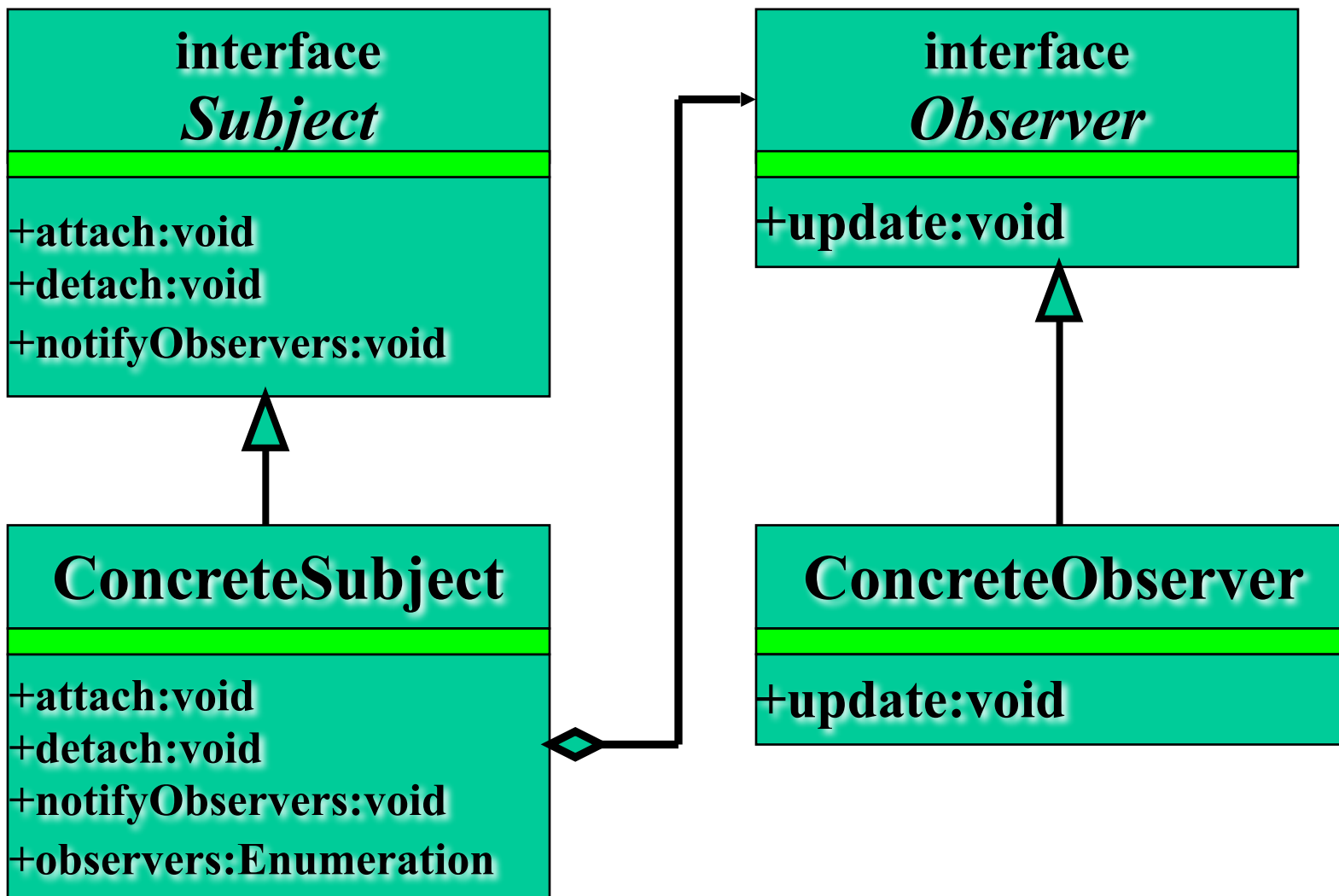
Observable/Observer module

Legend:

.....► Register Event

————► Send event

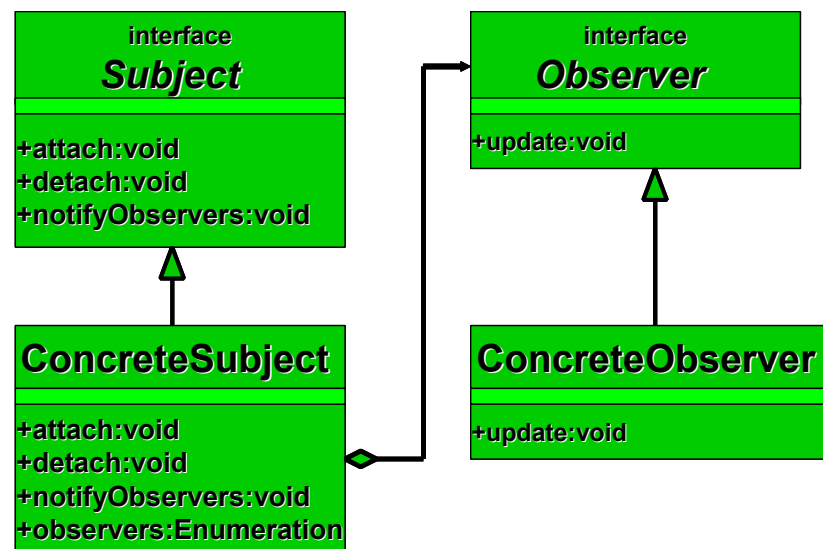
# Observer Pattern(观察者模式)



# Observer Pattern(观察者模式)

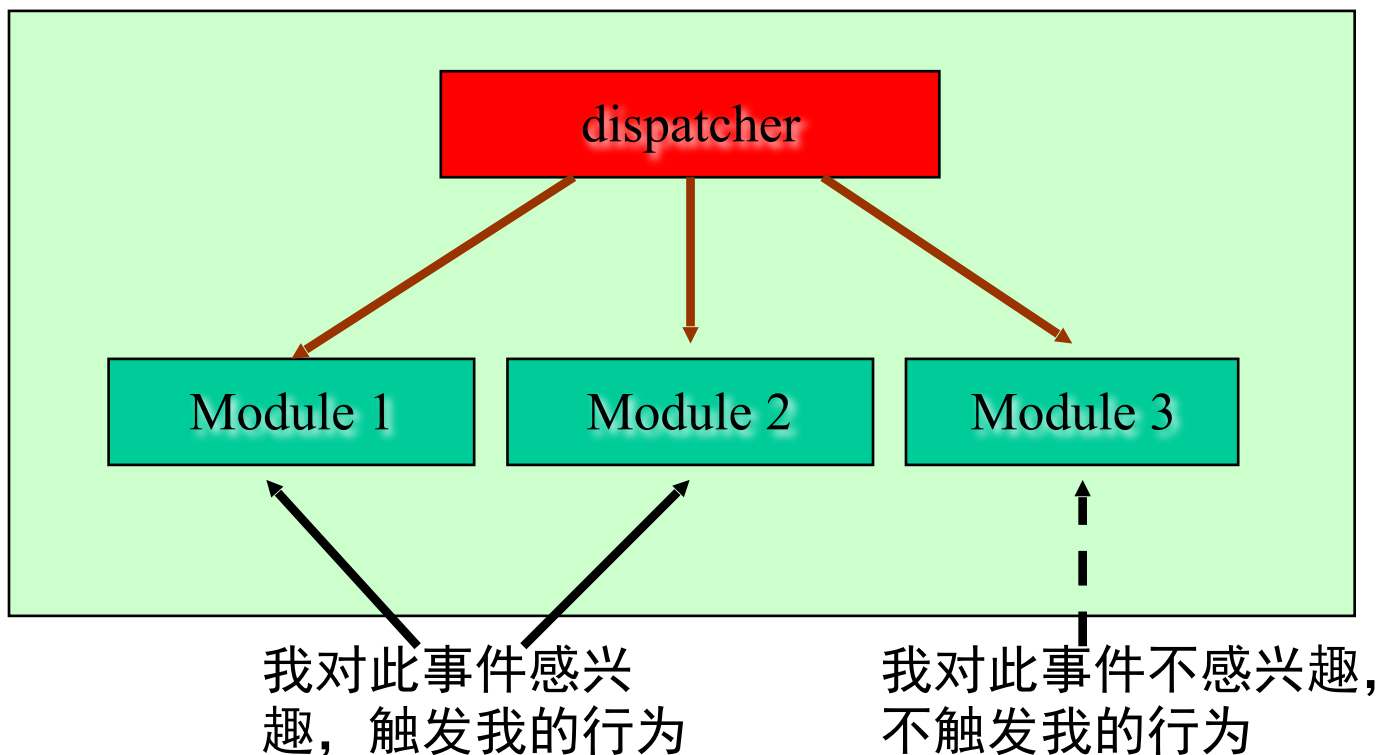
## • Mechanism

- A Concrete subject object attaches observers
- If there is an event, the Concrete subject object will notify all observers
- Every Observer subclasses has a method called update
- Once notified, the update will do something



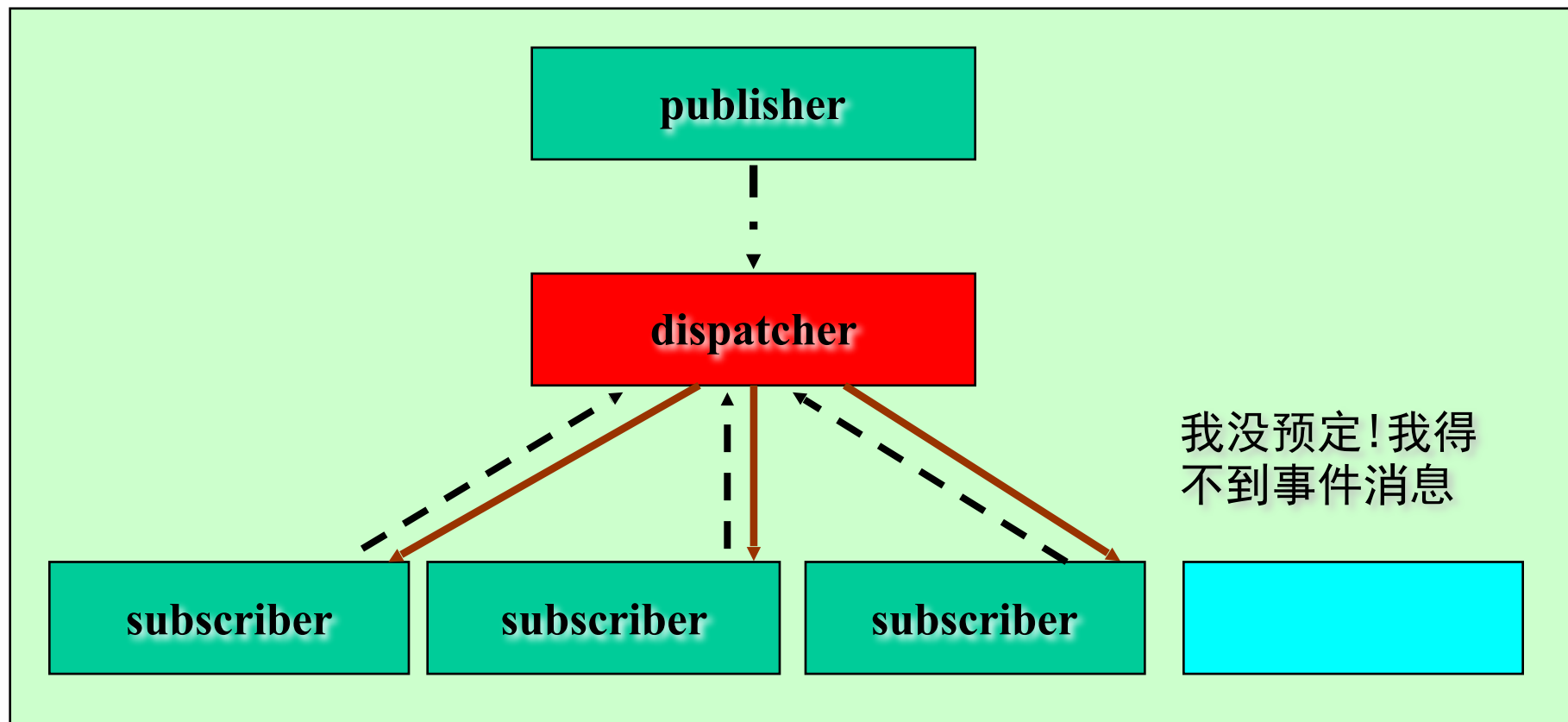
- 事件调度模块(Event dispatcher module)的功能：
  - The dispatcher module is responsible for receiving all incoming events and dispatching them to other modules in the system. (负责接收到来的事件并分发它们到其它模块)
  - The dispatcher should decide how events are sent to other modules with two different strategies (调度器要决定怎样分发事件：存在两种策略)
    - 全广播式(All broadcasting)：调度模块将事件广播到所有的模块，但只有感兴趣的模块才去取事件并触发自身的行为；
    - 选择广播式(Selected broadcasting)：调度模块将事件送到那些已经注册了的模块中。

无目的广播，靠接受者自行决定是否加以处理或者简单抛弃



我们都得到了  
事件

# 选择广播式



也称 发布/订阅策略.

**Publish/Subscribe strategy**

— · — · — · ▶ **Publish event**

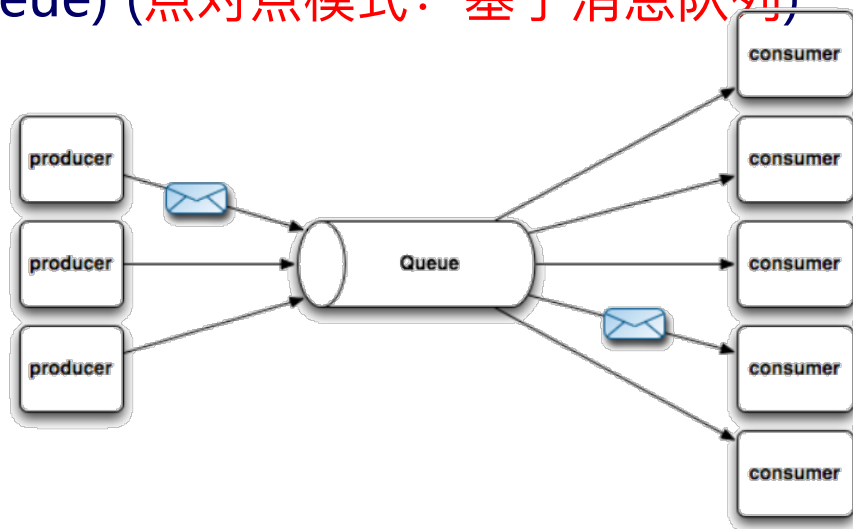
— — — — —▶ **Subscribe**

—————▶ **Send event**

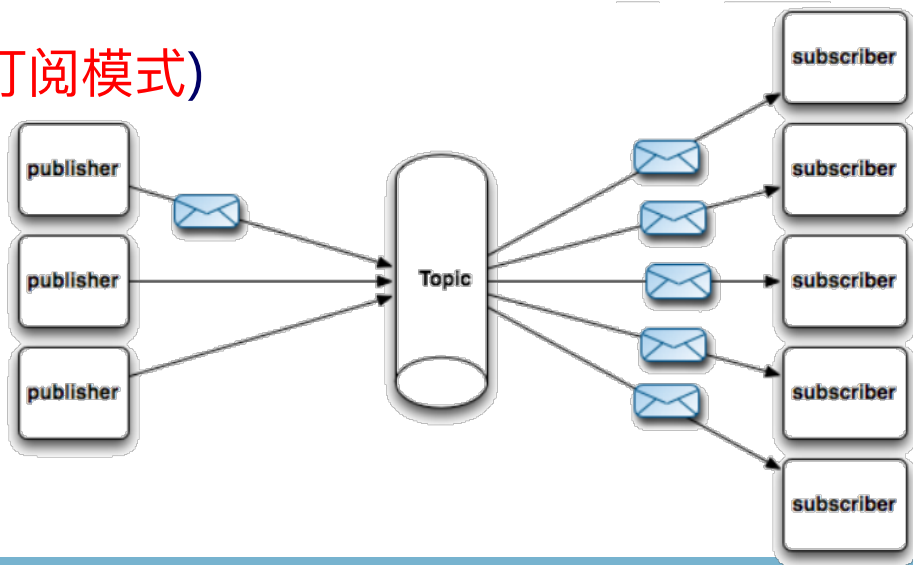


# 选择广播式的两种策略

- Point-to-Point (message queue) (点对点模式：基于消息队列)



- Publish-Subscribe (发布-订阅模式)

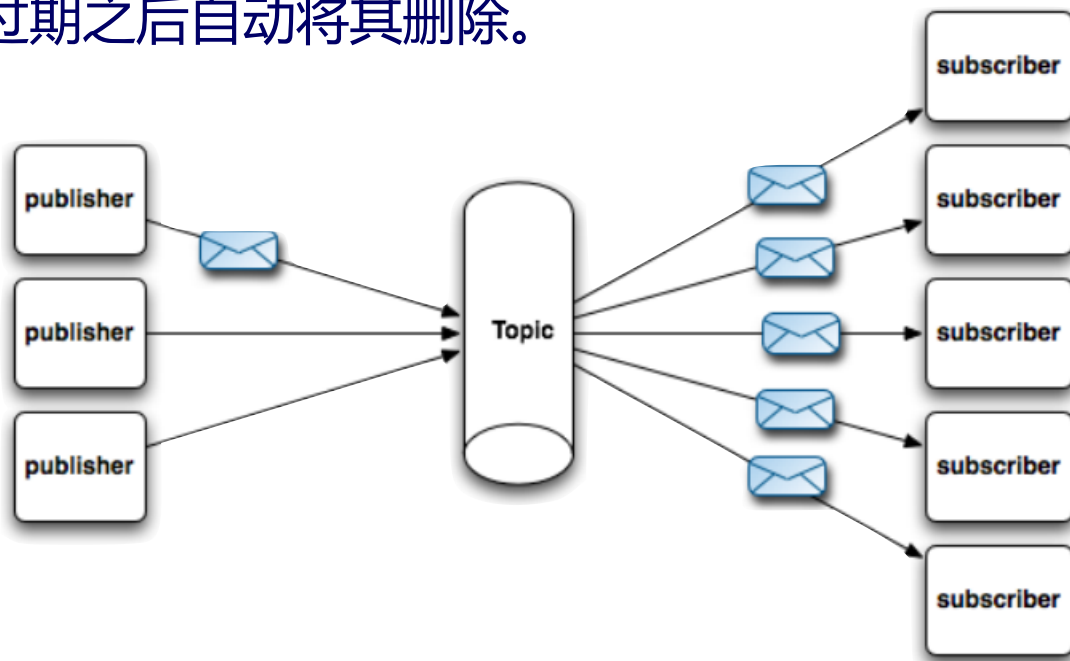


## 点对点的选择广播式：消息队列

- System installs and configures a queue manager and defines a named message queue. (系统安装并配置一个队列管理器，并定义一个命名的消息队列)
- An application then registers a software routine that “listens” for messages placed onto the queue. (某个应用向消息队列注册过程，以监听并处理被放置在队列里的事件)
- Second and subsequent applications may connect to the queue and transfer a message onto it. (其他的应用连接到该队列并向其中发布事件)
- The queue manager stores the messages until a receiving application connects and then calls the registered software routine. (队列管理器存储这些消息，直到接收端的应用连接到队列，取回这些消息并加以处理)
- Message can be consumed by only one client. (消息只能够被唯一的消费者所消费，消费之后即从队列中删除)

# 发布-订阅的选择广播式

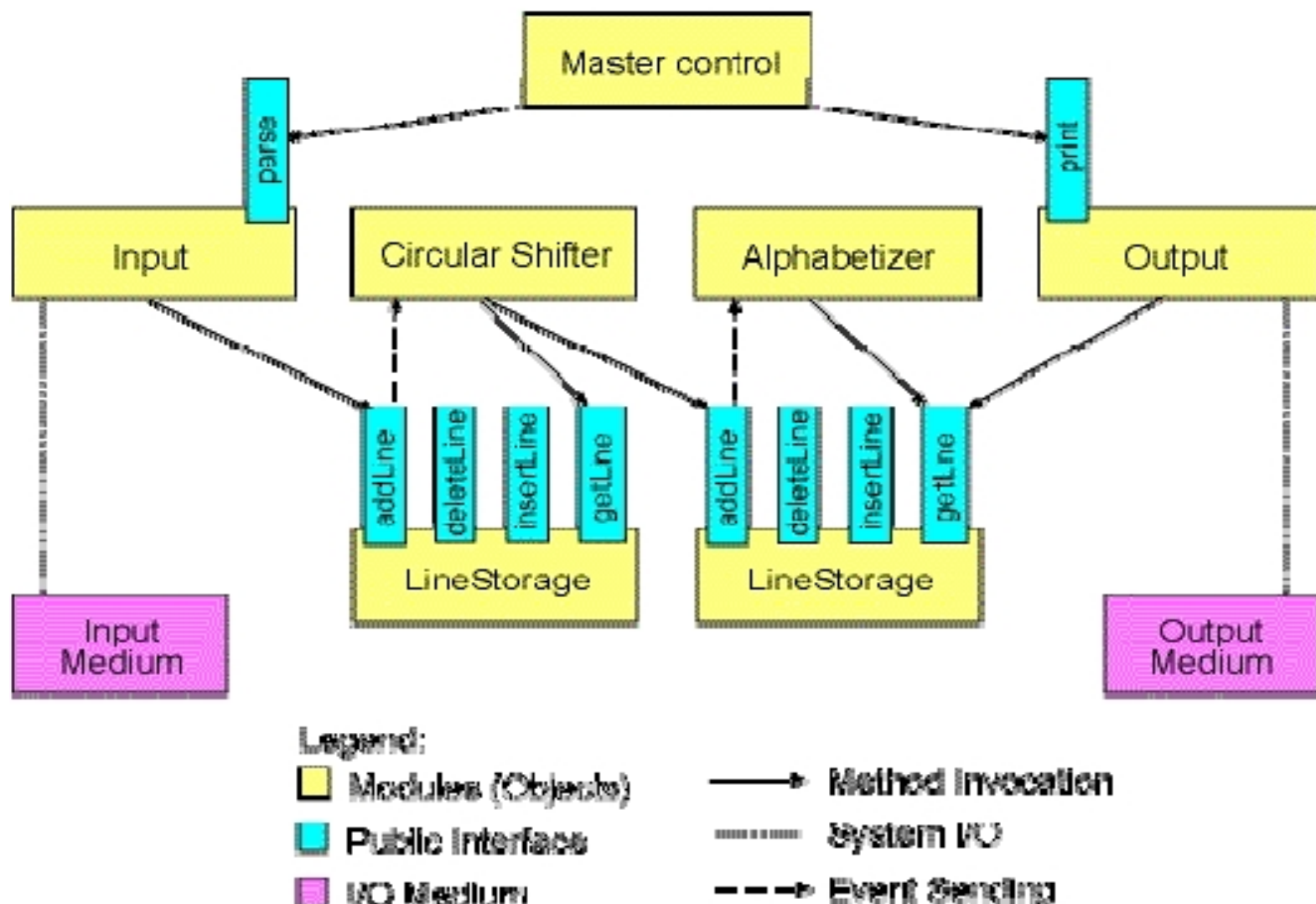
- Publishers post messages to an intermediary Broker and subscribers register subscriptions with that broker. (事件发布者向“主题”发布事件，订阅者向“主题”订阅事件)
  - 一个事件可以被多个订阅者消费；
  - 事件在发送给订阅者之后，并不会马上从topic中删除，topic会在事件过期之后自动将其删除。



## 2.5.3 事件系统风格应用

- 与主程序-子过程风格方案相似的地方
  - 四个功能模块
  - 共享数据
- 不同之处
  - 共享数据并不直接暴露其格式，而是进行封装（借鉴面向对象方案）
  - 模块的调用发生在数据发生改变时，不是主程序控制

# KWIC案例：事件驱动风格



- Two LineStorage modules
  - 第一个LineStorage模块负责保存最初输入的lines
  - 第二个LineStorage模块负责保存所有经过循环移位/排序后得到的
- Input:负责从输入文件读取信息并保存在第一个LineStorage模块中
- CircularShifter: 负责移位并将结果存储在第二个LineStorage模块中
- Alphabetizer: 负责对移位结果进行排序
- Output: 负责产生输出结果
- Master control: 负责系统的全局控制

# KWIC案例：设计模式

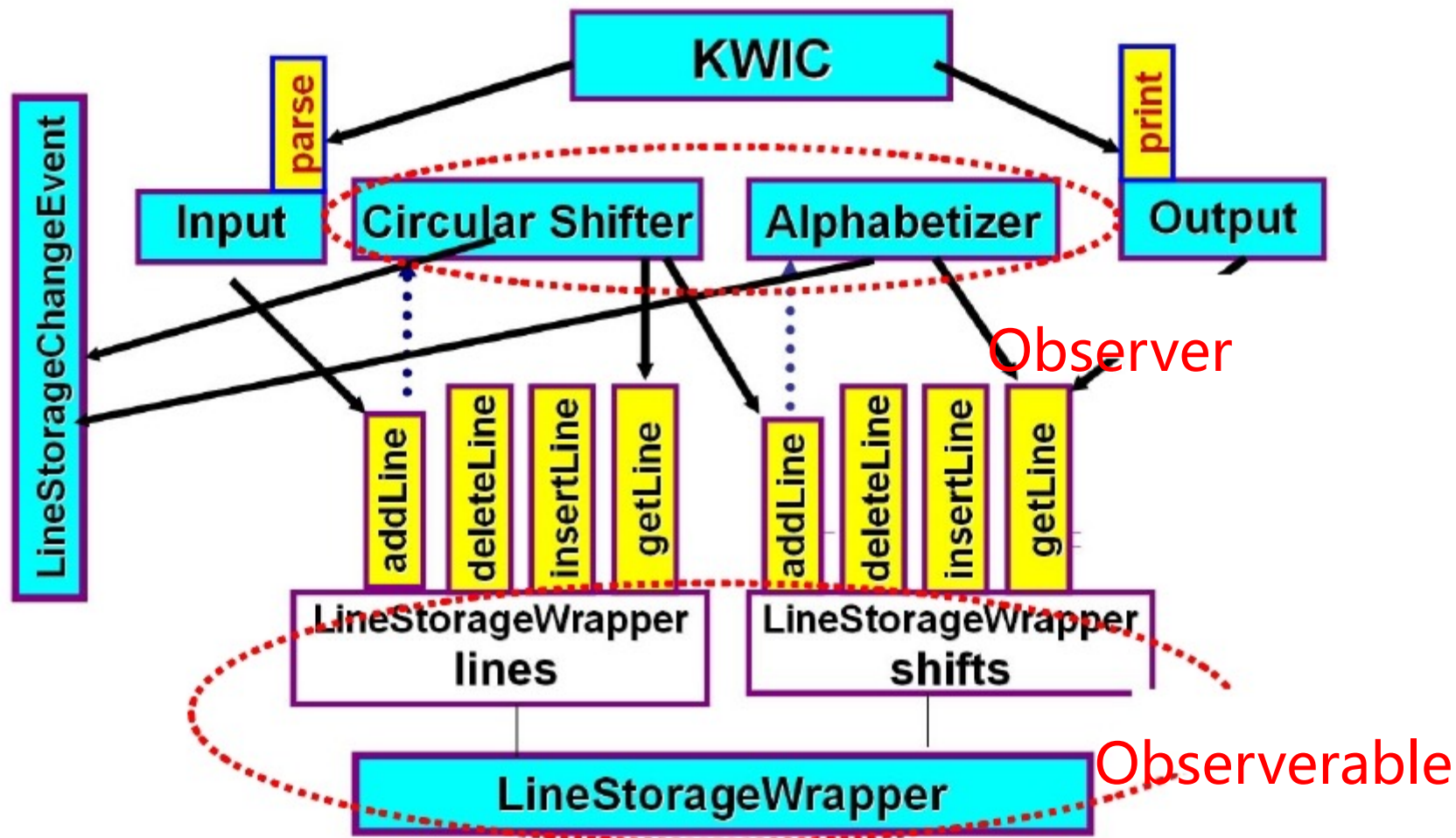
- 该案例中，使用观察者模式来处理事件
  - 两个LineStorage模块被实现为“被观察者”
  - CircularShifter和Alphabetizer模块被实现为“观察者”
- CircularShifter 是第一个 LineStorage 模块的观察者,
- 而 Alphabetizer 是第二个 LineStorage模块的观察者。



# KWIC案例：设计模式

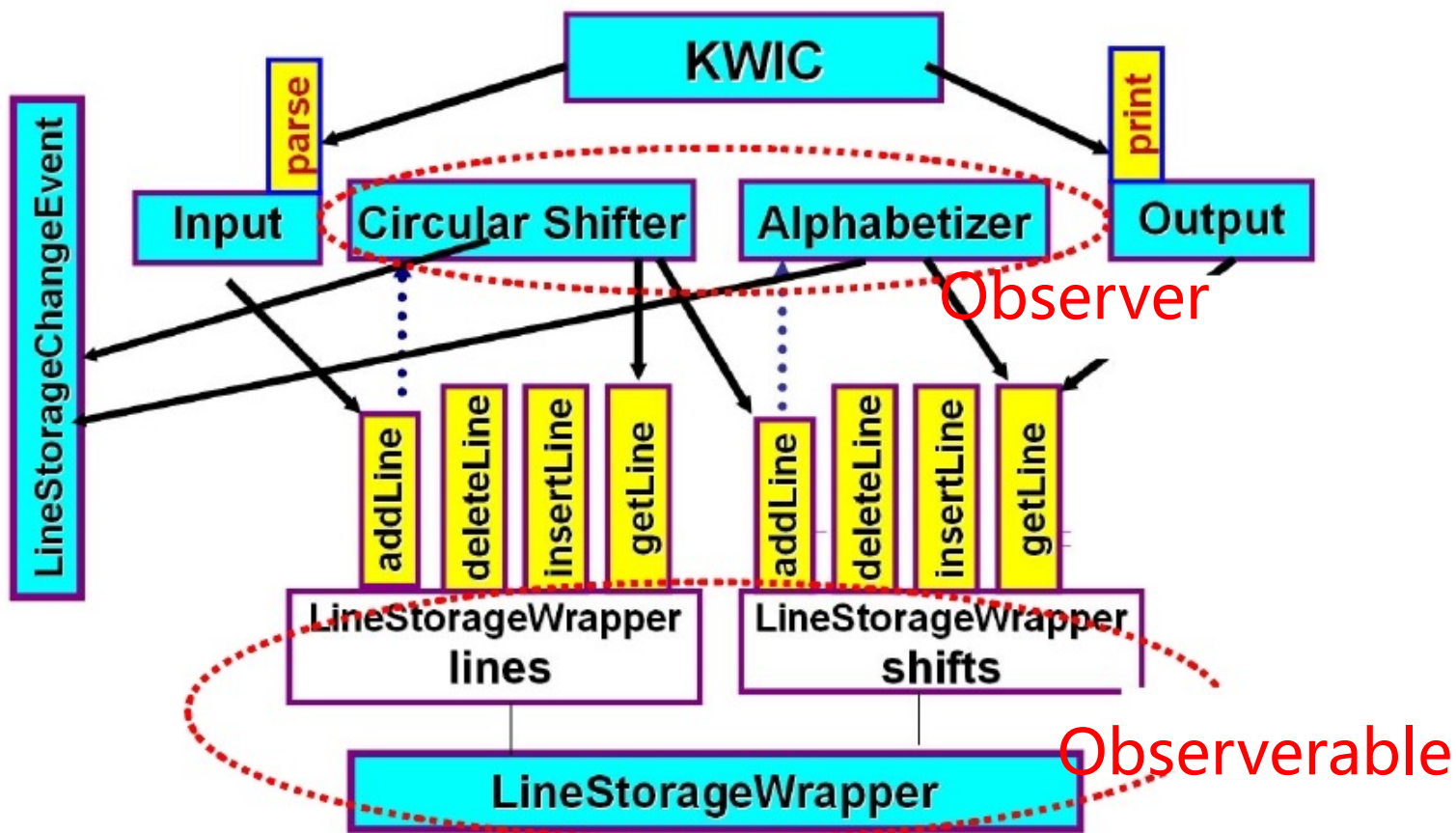
- CircularShifter向第一个LineStorage模块注册，以表明兴趣
  - 当有新行被加入到第一个LineStorage模块时，它将发出一个事件
  - CircularShifter模块接收到该事件
  - 作为对该事件的响应，CircularShifter模块对新加入的行进行循环移位，并将产生的结果写入第二个LineStorage模块中
- Alphabetizer模块向第二个LineStorage模块注册，表明兴趣
  - 当有新的循环移位行加入到第二个LineStorage模块时，它发出事件
  - Alphabetizer模块接收到该事件
  - 作为对该事件的响应，Alphabetizer对这些循环移位行进行排序

# KWIC案例：设计模式



# Quiz：显式调用与隐式调用的协作

- Alphabetizer是被事件驱动的，Output是通过显式调用激发的，两者怎么确定执行次序？

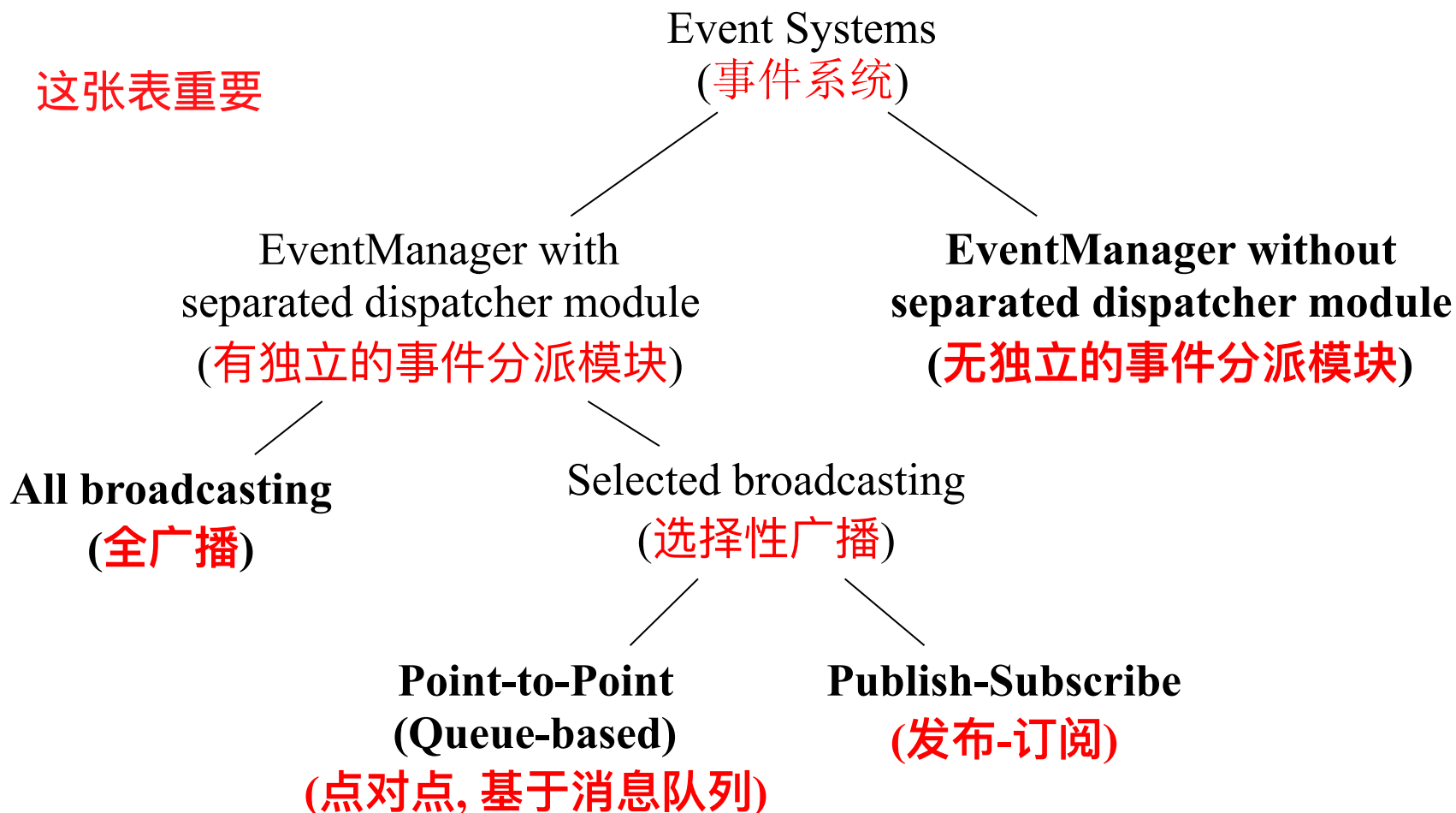


- 支持实现交互式系统（用户输入/网络通讯）
- 异步执行，不必同步等待执行结果；  
异步特点尤为明显
- 为软件复用提供了强大的支持。
  - 当需要将一个构件加入现存系统中时，只需将它注册到系统的事件中；
- 为系统动态演化带来了方便。
  - 构件独立存在，当用一个构件代替另一个构件时，不会影响到其它构件的接口；
- 对事件的并发处理将提高系统性能；
- 健壮性：一个构件出错将不会影响其他构件。

- 分布式的控制方式使得系统的同步、验证和调试变得异常困难：
  - 构件放弃了对系统计算的控制。一个构件触发一个事件时，不能确定其它构件是否会响应它。而且即使它知道事件注册了哪些构件的构成，它也不能保证这些过程被调用的顺序。
  - 既然过程的语义必须依赖于被触发事件的上下文约束，关于正确性的推理则难以保证。
  - 传统的基于先验和后验条件的验证变得不可能。
- 数据交换的问题：
  - 数据可通过事件直接在系统间传递(无调度模块时)，但在具有独立调度模块的事件系统中，数据则需要经过调度模块的传递。在这些情况下，全局性能和资源管理成为了系统的瓶颈。

# 小结: 事件系统的分类

这张表重要



请仔细辨析这些不同类型的事件系统

## 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