



第3章 经典软件体系结构风格

3.1 调用-返回风格 Call / Return style



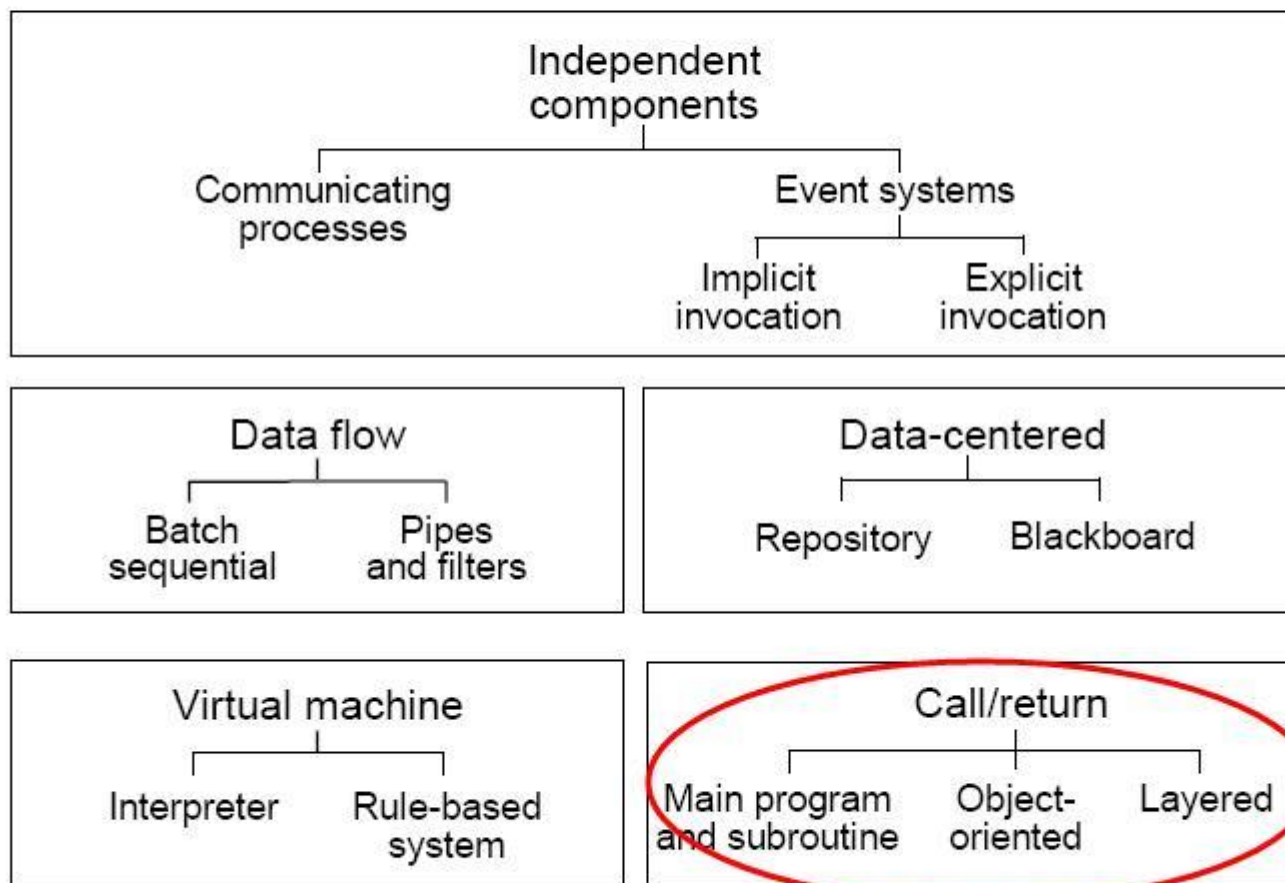
刘其成

计算机与控制工程学院

ytliuqc@163.com

2018-9

主要内容



主要内容

- 3.1.1 主程序-子过程风格
(Main program and subroutine)
- 3.1.2 面向对象风格(Object oriented System)
- 层次风格 3.4

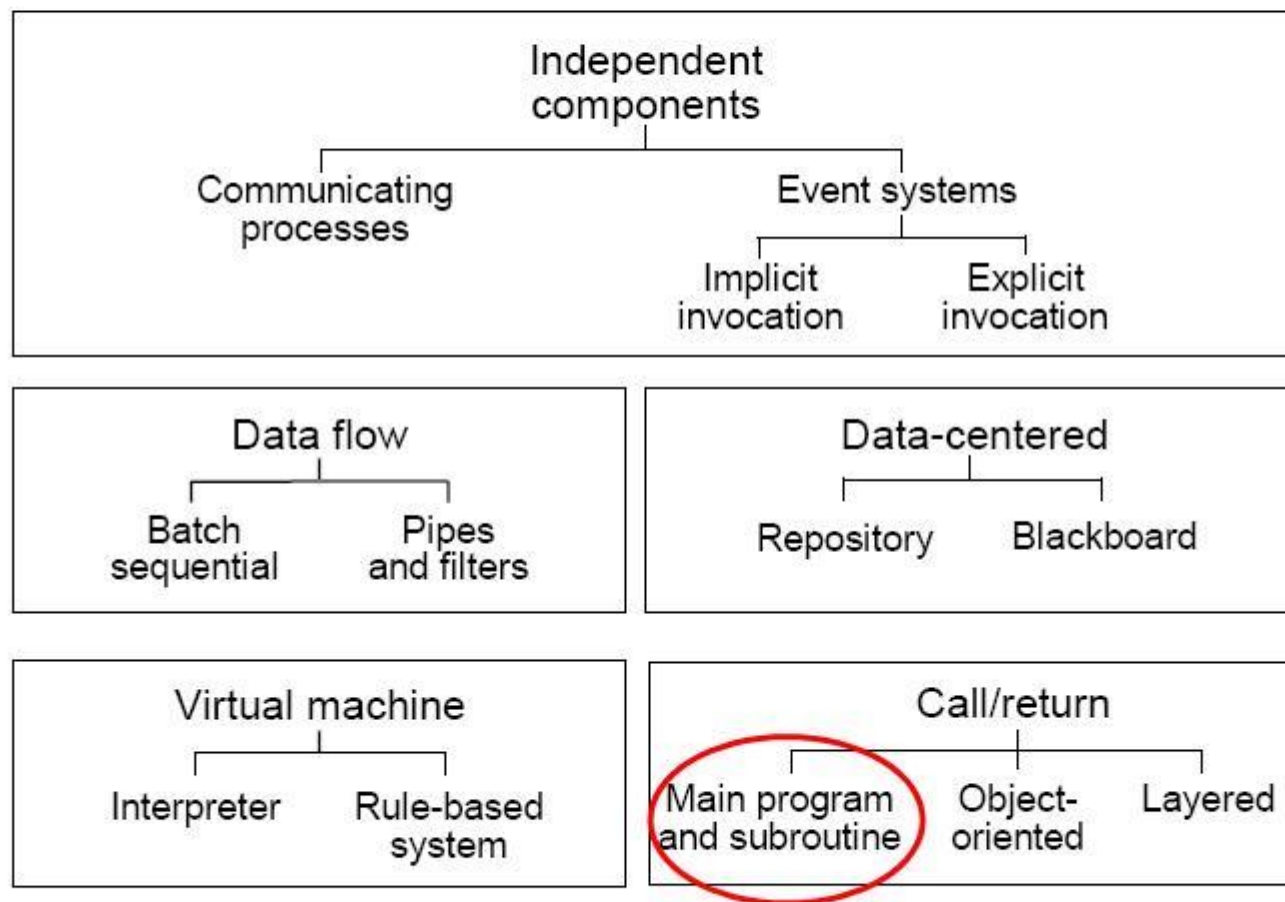


3.1.1 主程序-子过程风格

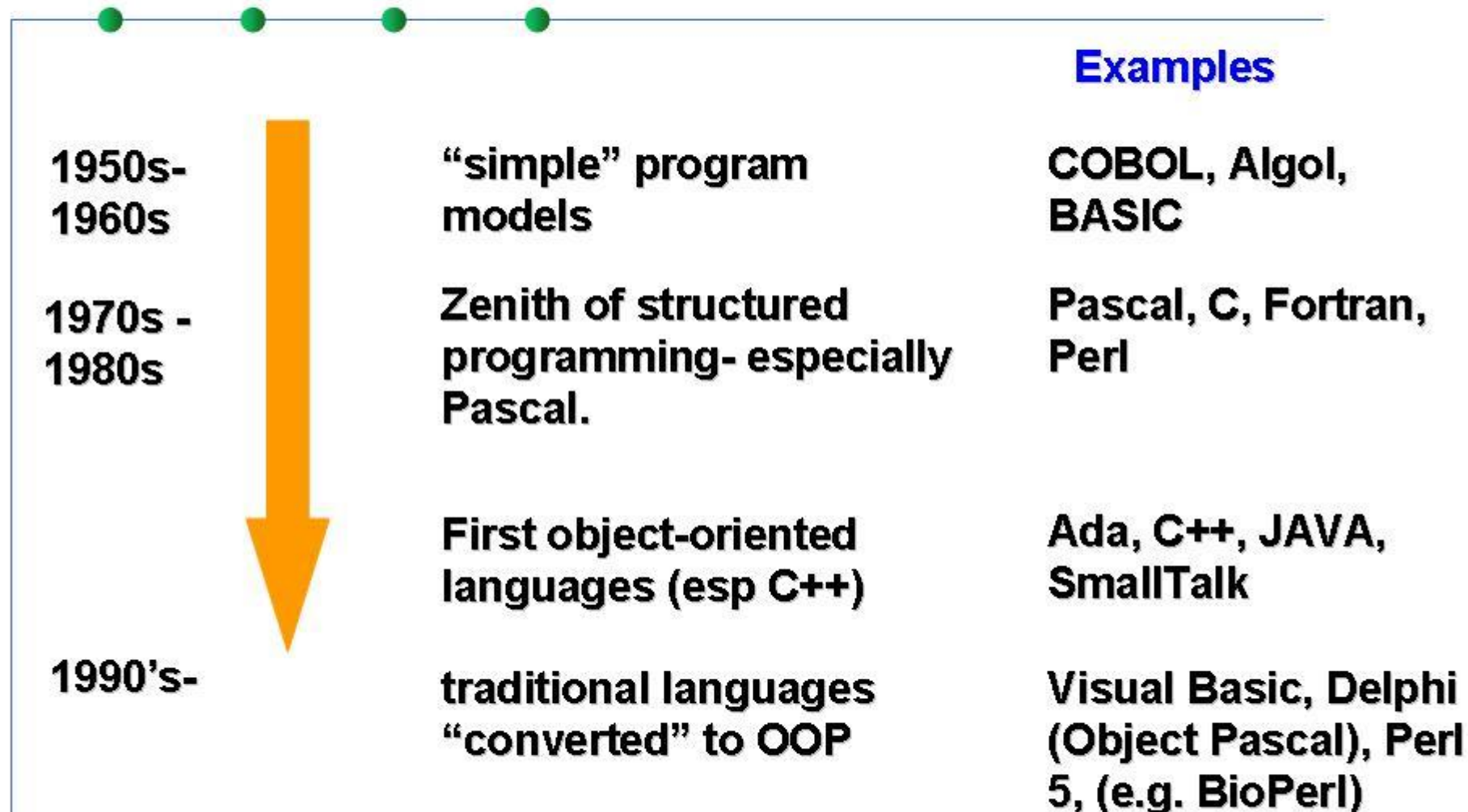
Main program and subroutine



主要内容



Evolution of Program Design



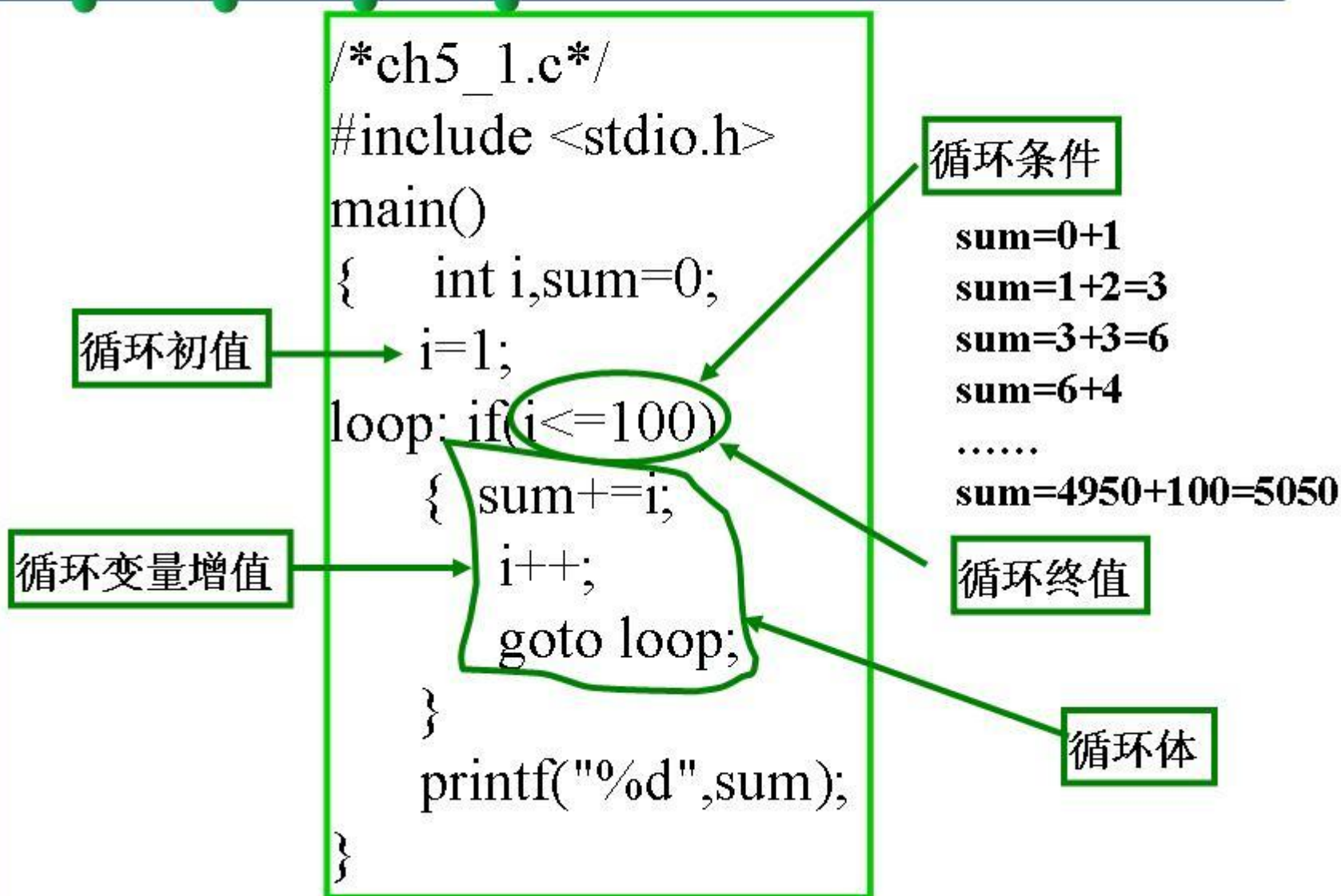
非结构化的程序(Unstructured programming)

- 所有的程序代码均包含在一个主程序文件中，经常使用语句标号和**goto**语句。
- 缺陷
 - 逻辑不清
 - 无法复用
 - 难以与其他代码合并
 - 难于修改
 - 难以测试特定部分的代码

非结构化方法

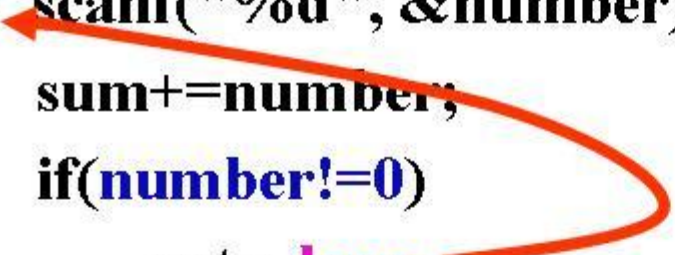
- 语句标号
- **goto**

例 用if和goto语句构成循环，求 $\sum_{n=1}^{100} n$



例 输入一组数据求和

```
#include <stdio.h>
main() {
    int number, sum=0;
    loop: scanf("%d", &number);
        sum+=number;
        if(number!=0)
            goto loop;
    printf("The total sum is %d\n", sum);
}
```



从键盘输入完一组数据后，再输入一个0作为结束输入数据的标志。



```
#include <stdio.h>
```

```
main() {
```

```
    int number,sum=0;
```

```
    read_loop: scanf("%d",&number);
```


```
        if(!number) goto print_sum;
```

```
        sum+=number;
```

```
        goto read_loop;
```

```
    print_sum: printf("The total sum is %d\n",sum);
```

```
}
```

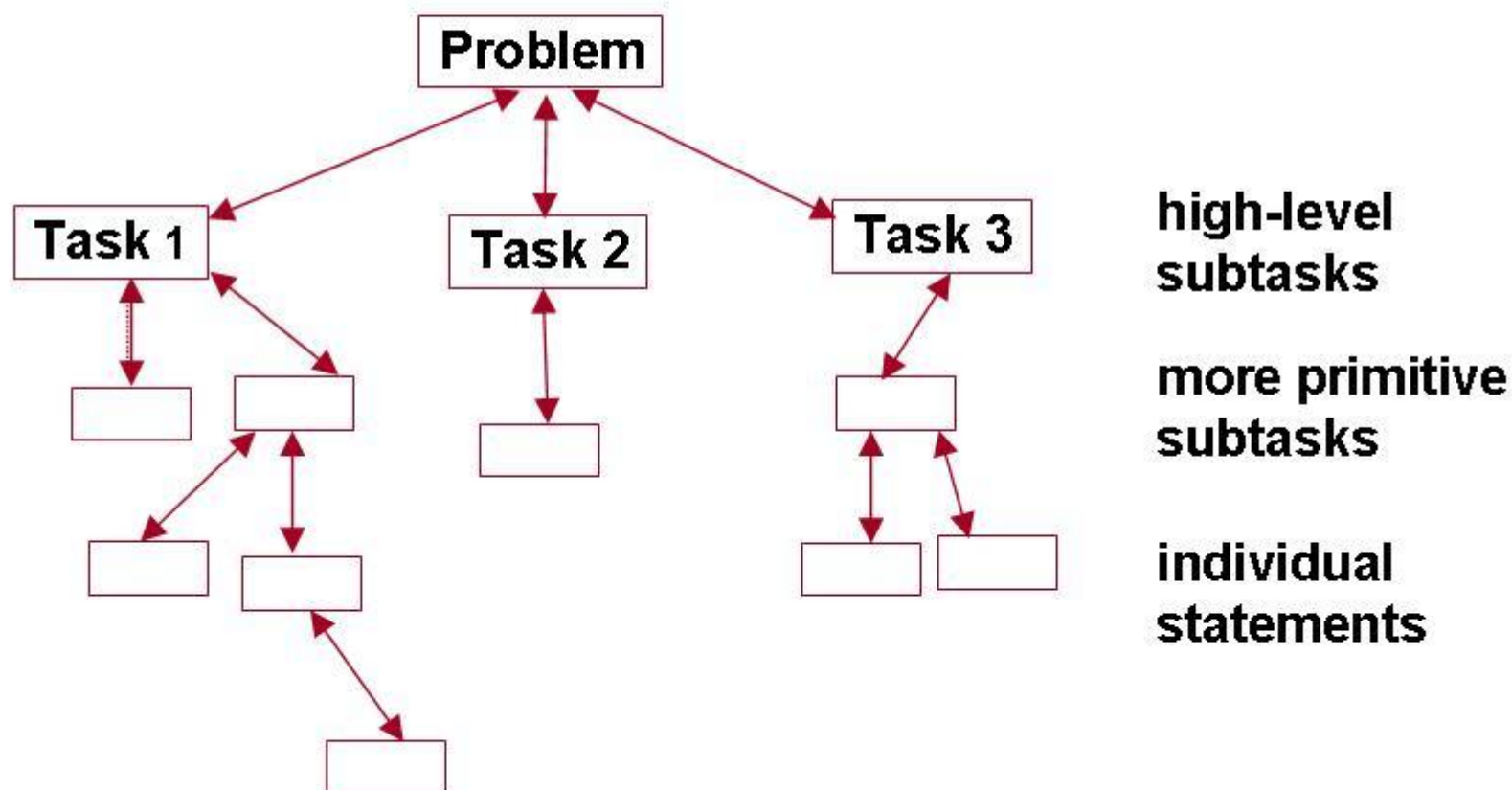
- 
- **goto**语句**不符合**结构化程序设计的原则，因为无条件转向使得程序的结构没有规律、可读性差。
 - 对于初学者来说应尽量避免使用**goto**语句，但如果使用**goto**语句能够大大地提高程序的执行效率，也可以使用。

结构化方法

- 设计思路：自顶向下、逐步求精。采用模块分解与功能抽象，自顶向下、分而治之
- 程序结构：
 - 按功能划分为若干个基本模块，形成一个树状结构
 - 各模块间的关系尽可能简单，功能上相对独立；每一模块内部均是由顺序、选择和循环三种基本结构组成
 - 其模块化实现的具体方法是使用函数(子程序)

结构化程序

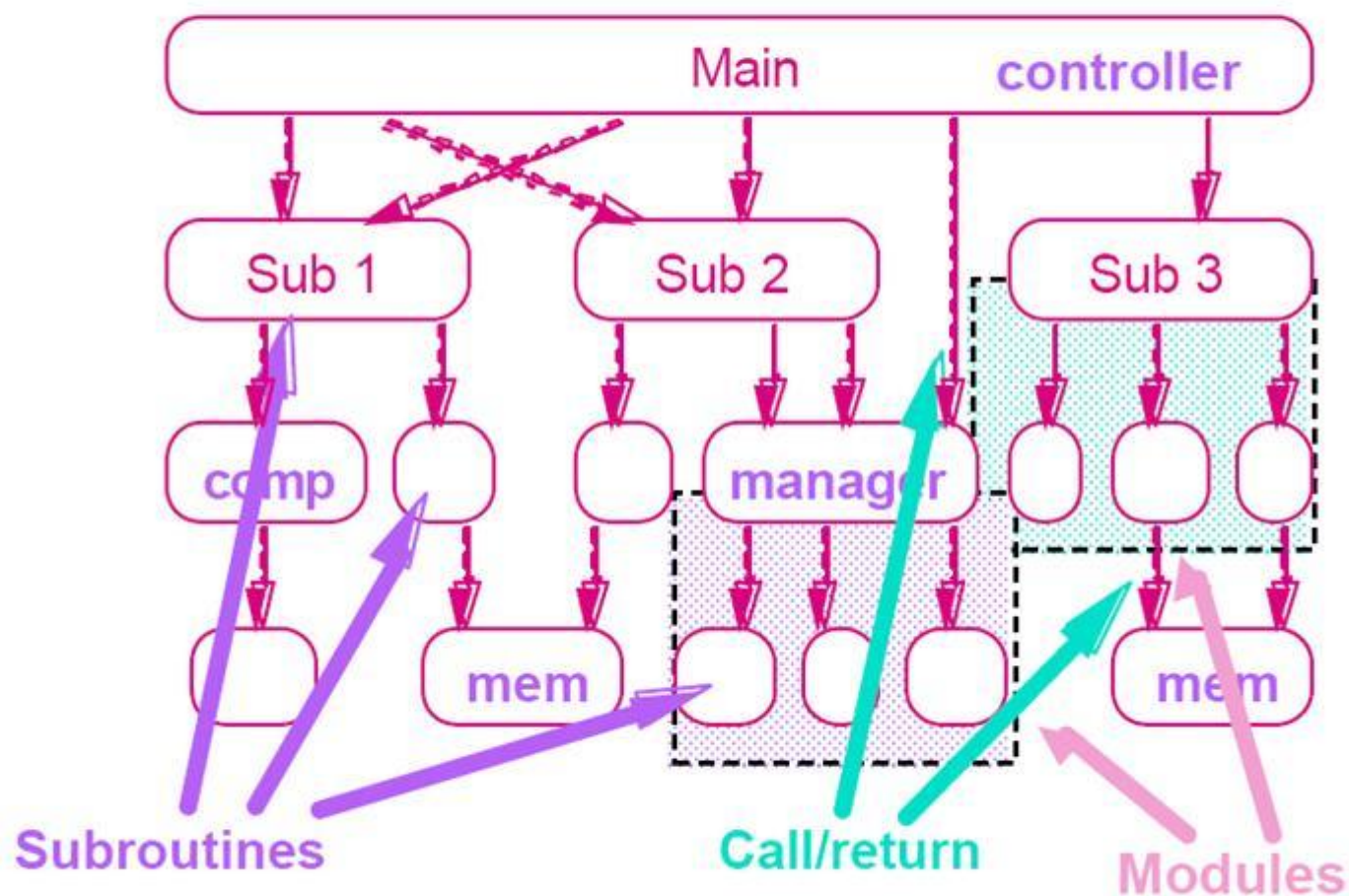
结构化程序：自上而下的设计(Top-down Design)



结构化程序

- 从功能的观点设计系统
- 结构化设计与逐步细化是这种风格的典型实例
- 逐层分解(Hierarchical decomposition):
 - 基于“定义-使用”关系
 - 用过程调用作为交互机制
 - 主程序的正确性依赖于它所调用的子程序的正确性

主程序-子过程风格(Main program and subroutine)



主程序-子过程风格的基本构成

- 组件
 - 主程序、子程序
- 连接件
 - 调用-返回机制
- 拓扑结构
 - 层次化结构
- 本质：
 - 将大系统分解为若干模块(模块化)，主程序调用这些模块实现完整的系统功能。

C程序的执行顺序

- 一个源程序文件，由一个或多个函数以及其他有关内容组成，是一个编译单位，**函数不是一个编译单位。**
- **C**程序的执行总是从**main**函数开始，调用其它函数后回到**main**函数，在**main**函数中结束整个程序的运行；
- 所有的子函数都是平行的，任何子函数都不属于其他函数；

例

```
#include <stdio.h>

void printstar()
{ printf("*****\n");
}

void printmessage()
{printf(" Hello,world.\n");
 printstar();
}

void main()
{
    printstar();
    printmessage();
}
```




■ 组件

- 主程序main()
- 子程序printstar()和printmessage()

■ 连接件

- 主程序main()调用子程序printstar()和printmessage()
- 因为没有参数的传递，所以比较简单。

- 
- 含有参数的子程序的一般调用过程如下。
 - 按从右到左的顺序，计算实参各表达式的值；
 - 按照位置，将实参的值一一传给形参；
 - 执行被调用函数（子程序）；
 - 当遇到**return**(表达式)语句时，计算表达式的值，并返回主调函数（主程序）。

函数的调用

函数调用的执行过程

1. 按从右到左的顺序，计算实参各表达式的值；
2. 按照位置，将实参的值一一传给形参；
3. 执行被调用函数；
4. 当遇到return(表达式)语句时，计算表达式的值，并返回主调函数。

例8-4 读程序，写出结果

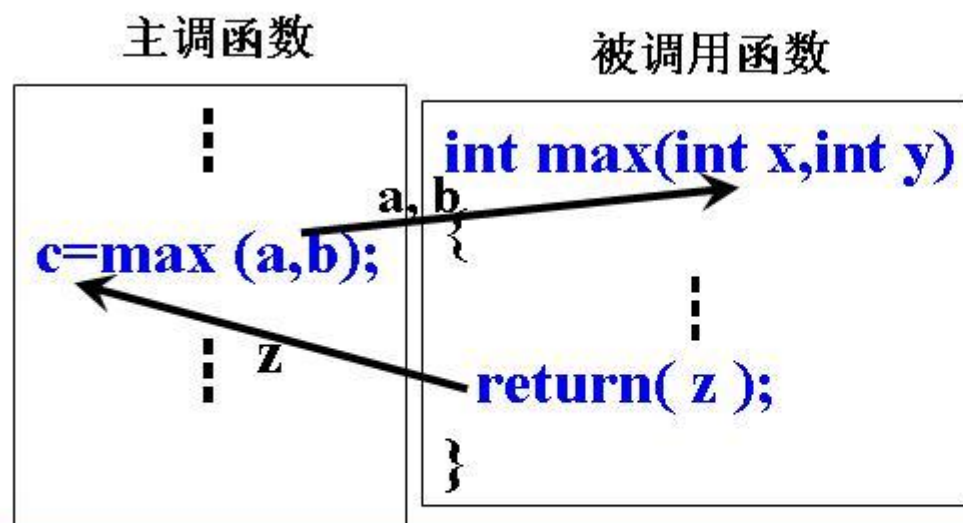
```
#include <stdio.h>

int iabs( float x)
{ return (x>0 ? x : -x); }

void main( )
{ float x=-1.2,y;
  y= iabs(2*x )
  printf( "x=%f , iabs(x)=%f\n",x,y);
}
```

函数参数和函数的值——调用返回机制

一个C程序由若干个函数组成，各函数调用时经常要传递一些数据，调用函数把数据传递给被调用函数，经被调用函数使用后，一般会返回一个确定结果，在返回调用函数时，把这些结果带回调用函数。



例

```
#include <stdio.h>
int max(int x,int y)
{ int z;
  z = x > y ? x : y;
  return( z );
}

void main()
{ int a,b,c ;
  scanf("%d,%d",&a,&b);
  c=max(a,b);
  printf("The max is %d", );
}
```

各函数的信息往来主要是由参数传递和返回语句实现的



■ 组件

- 主程序**main()**函数
- 子程序**max(a,b)**函数;

■ 连接件

- **main()**函数中调用**max(a,b)**函数，**max()**函数将实参**a**、**b**分别传递给虚参**x**、**y**，通过运算得到较大值**z**，并将**z**返回调用处，赋值给**main()**函数的变量**c**。

主程序-子过程风格的优点与缺点

■ 优点:

- 有效地将一个较复杂的程序系统设计任务分解成许多易于控制和处理的子任务，便于开发和维护
- 已被证明是成功的设计方法，可以被用于较大程序

主程序-子过程风格的优点与缺点

■ 缺点:

- 规模: 程序超过10万行, 表现不好; 程序太大, 开发太慢, 测试越来越困难
- 可重用性差、数据安全性差, 难以开发大型软件和图形界面的应用软件
- 把数据和处理数据的过程分离为相互独立的实体, 当数据结构改变时, 所有相关的处理过程都要进行相应的修改
- 图形用户界面的应用程序, 很难用过程来描述和实现, 开发和维护也都很困难。

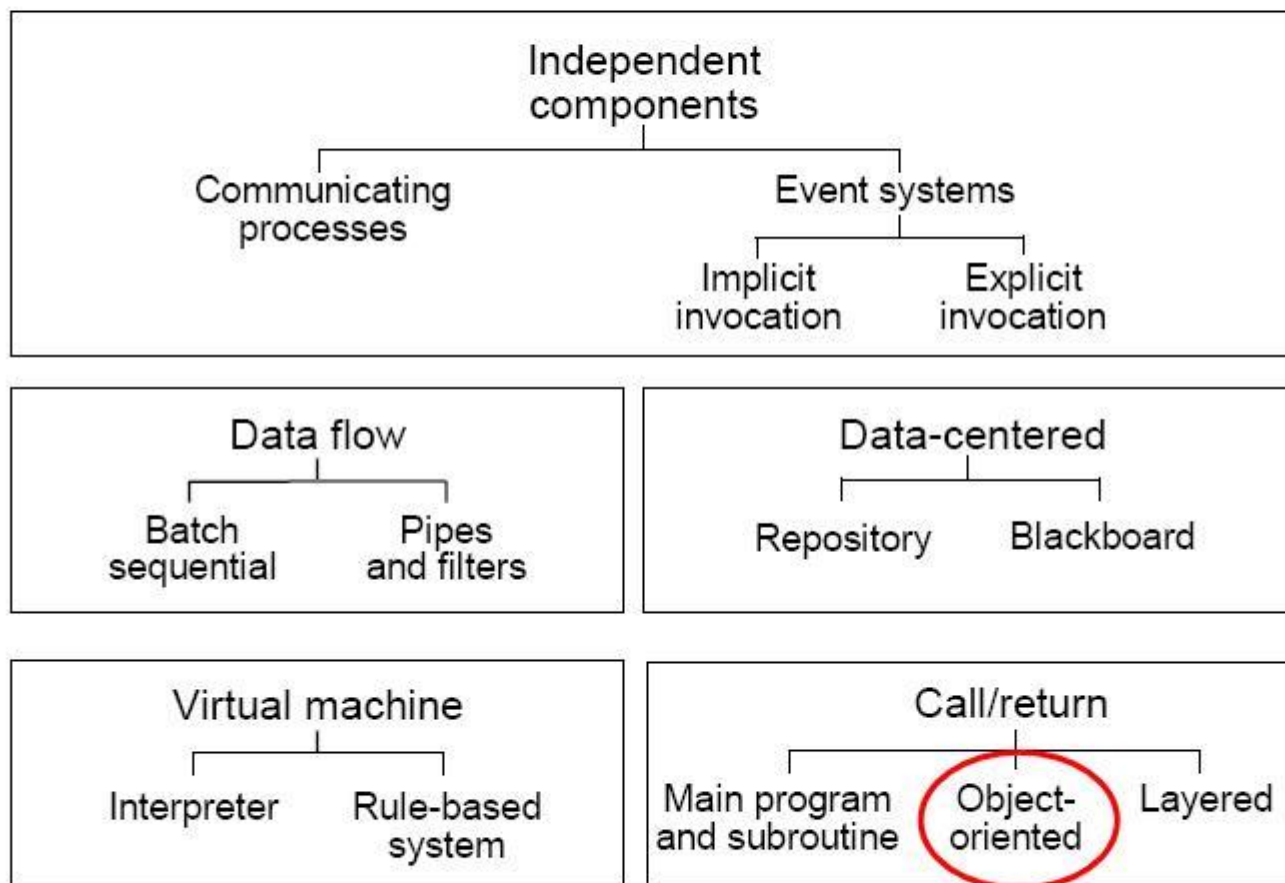


3.1.2 面向对象风格

OO systems

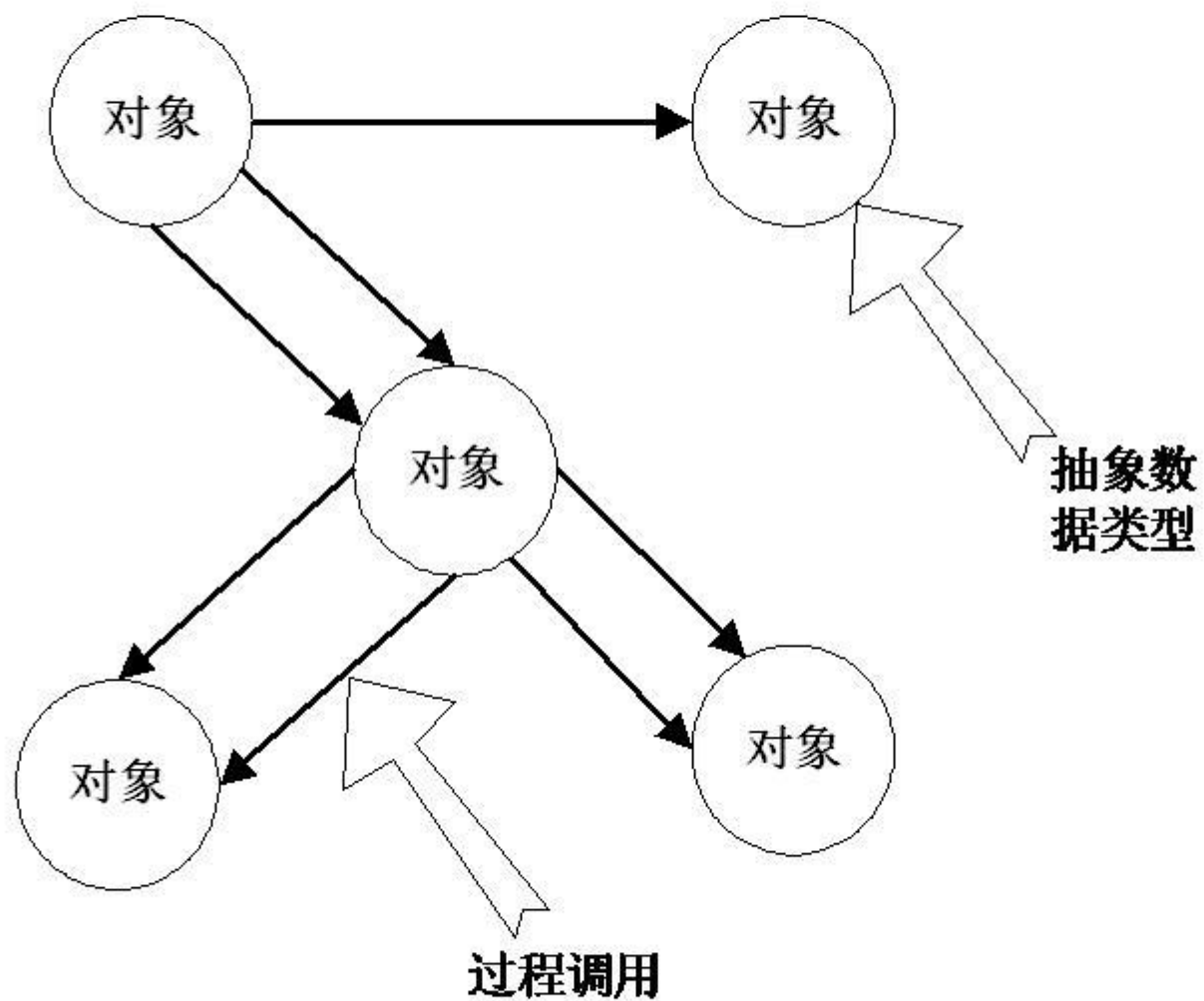


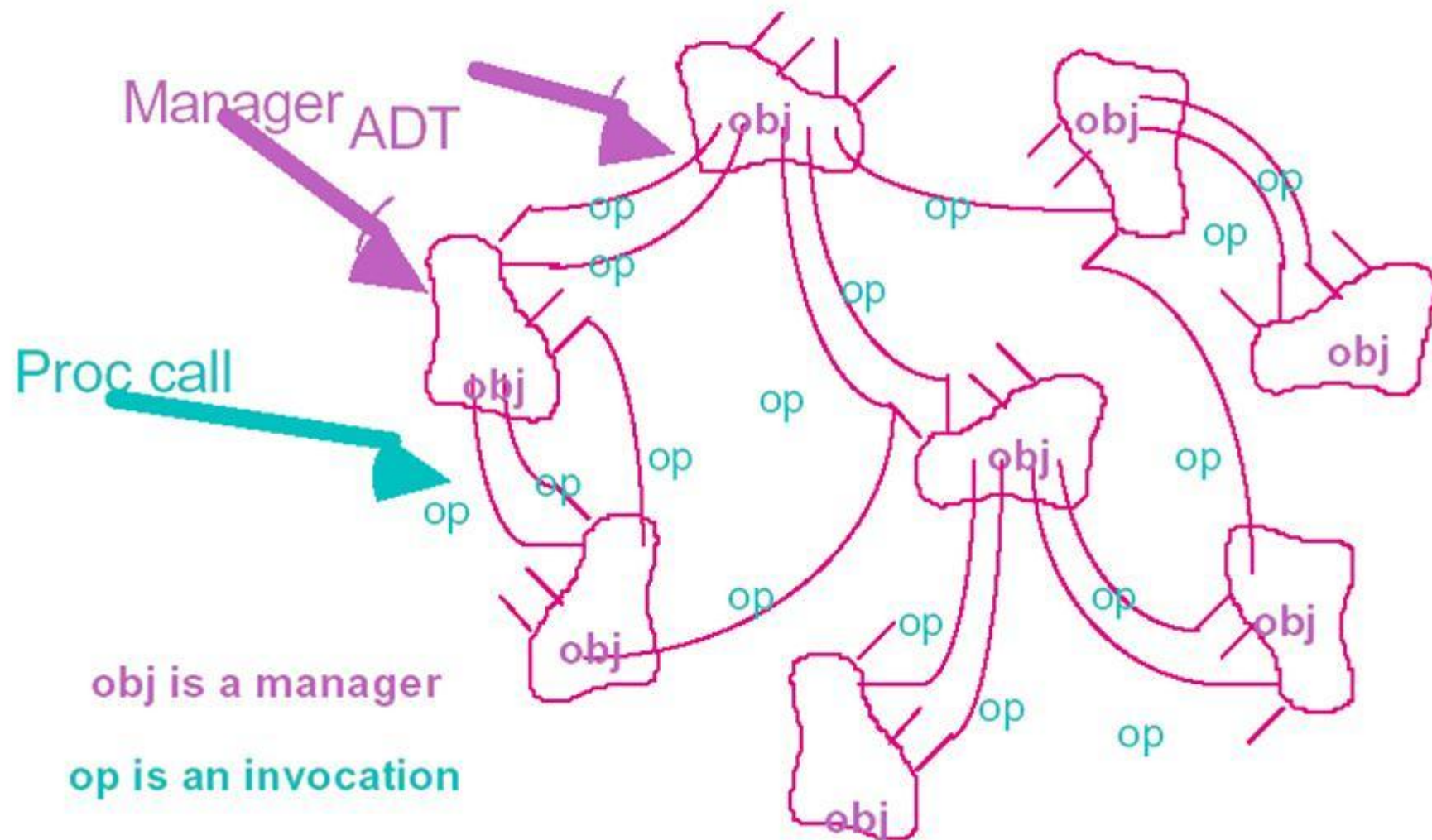
主要内容



面向对象风格

- 对象也叫抽象数据类型 **abstract data type(ADT)**
 - 系统被看作对象的集合
 - 每个对象都有一个它自己的功能集合。
 - 抽象数据类型概念对软件系统有着重要作用，目前软件界已普遍转向使用面向对象系统。
- 面向对象风格
 - 建立在数据抽象和面向对象的基础上
 - 数据的表示方法和它们的相应操作封装在一个抽象数据类型或对象中。





OO风格基本构成

- 组件

- 对象，或者说是抽象数据类型（类）的实例
- 类

- 连接件

- 对象是通过函数和过程的调用-返回机制来交互的
- 类通过定义对象，再采用调用调用-返回机制进行交互

对象的访问



■ 例 对象用作方法的参数

```
class Spot {
    private int x, y;
    Spot (int u, int v) {
        setX(u); setY(v);
    }
    void setX(int x1) { x=x1; }
    void setY(int y1) { y=y1; }
    int getX() { return x; }
    int getY() { return y; }
}

class Trans {
    void move(Spot p, int h, int k) {
        p.setX(p.getX() + h);
        p.setY(p.getY() + k);
    }
}
```

```
class Test {
    public static void main(String args[]) {
        Spot s = new Spot(2, 3);
        System.out.println("s点的坐标:"
            + s.getX()+" "+s.getY());
        Trans ts = new Trans();
        ts.move(s, 4, 5);
        System.out.println("s点的坐标:"
            +s.getX()+" "+s.getY());
    }
}
```

```
D:\java Test
s点的坐标:2,3
s点的坐标:6,8
```




■ 组件

- **Spot、Trans、Test**三个类
- **Spot**的对象**s**，**Trans**的对象**ts**。

■ 连接件如下

- 在**Test**类里面创建**Spot**类的对象**s**、**Trans**类的对象**ts**，**Trans**类的**move()**方法的参数里面有**Spot**类的对象**p**。
- **Test**类使用**Spot**类的对象**s**，调用了**Spot**类的**getX()**和**getY()**方法；**Test**类使用**Trans**类的对象**ts**，调用了**Trans**类**move()**方法，并把实参**Spot**类的对象**s**传递给了虚参**Spot**类的对象**p**。

OO特性

- 抽象
- 封装：限制对某些信息的访问
- 多态：在运行时选择具体的操作
- 继承：对共享的功能保持唯一的接口

- 交互：通过过程调用或类似的协议
- 动态绑定：运行时决定实际调用的操作
- 复用和维护

OO风格优点

- 复用和维护：利用封装和聚合提高生产力
 - 因为对象对其它对象隐藏它的表示，所以可以改变一个对象的表示，而不影响其它的对象。
 - 某一组件的算法与数据结构的修改不会影响其他组件
 - 组件之间依赖性降低，提高了复用度
- 反映现实世界
- 容易分解一个系统
 - 设计者可将一些数据存取操作的问题分解成一些交互的代理程序的集合

OO风格缺点

- 管理大量的对象：怎样确立大量对象的结构
- 继承引起复杂度，关键系统中慎用
- 必须知道对象的身份
 - 为了使一个对象和另一个对象通过过程调用等进行交互，必须知道对象的标识。只要一个对象的标识改变了，就必须修改所有其他明确显式调用它的对象，并消除由此带来的一些副作用(例如，如果**A**使用了对象**B**，**C**也使用了对象**B**，那么，**C**对**B**的使用所造成的对**A**的影响可能是料想不到的)
 - 对比：在管道-过滤器系统中，一个过滤器无需知道其他过滤器的任何信息
- 不是特别适合功能的扩展。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，从而影响性能

思考题

■ 1. 主程序-子程序

- 组件、连接件、工作机制、特点、实例
- 相关程序，语言不限

■ 2. 面向对象

- 组件、连接件、工作机制、特点、实例
- 相关程序，语言不限



谢谢

2018年10月10日