2014年秋季硕士生课程（X24CS1132 /Z11CS1132 ）

面向对象技术

The Technology of Object Orientation

西安电子科技大学软件学院　陈平

本课程的相关实施信息

学习目标

- 建立以工业化方式研制、生产和维护软件的观念;
- 学习和掌握一种软件开发范型（*Software Development Paradigm*）;
- 在实验过程中加深理解，提高发现问题、分析问题和解决问题的能力;
- 为形成较高的软件境界打下基础;
- 为进一步的研究打下基础。

本课程的相关实施信息

主要内容与先修课要求

- 主要内容:
  - 类型系统（*The Type System*）
  - 面向对象程序设计（*Object-Oriented Programming, OOP*）
  - 面向对象分析与设计（*Object-Oriented Analysis and Object-Oriented Design, OOA&OOD*）
- 先修课:
  - 数据结构
  - 编译原理
  - 软件工程
  - C 程序设计

本课程的相关实施信息

实施手段

- 课堂讲授 40 学时（13次），实验所需时间与能力有关，通常不少于 60 小时。
- 完成三次作业和一个实验项目（例：过去的实验项目）。
- 最终成绩:
  - 作业 20%；实验项目 25%；闭卷考试 55%。
  - 作业或实验项目雷同者，平分应得的成绩。
  - 考试作弊者按学校规定处理。

本课程的相关实施信息

通信协议

- 正式选课后，请同学们通过网络将自己的姓名、学号、邮件地址提交给辅导教师（*http://219.245.68.129/*，*再进入“教辅支撑平台”*）。
- 辅导教师：楮华副教授（邮件：*hchu@mail.xidian.edu.cn*）
- 选修的同学课后可在上述平台下载本次课的幻灯片（.pdf格式）。
- 选修的同学可通过上述平台以在线留言的方式与辅导教师交流。
- 作业和实验项目报告均通过上述平台提交。
- 非选课同学将得不到响应。

§1. 绪　论

1.1 软件开发面临的普遍性问题

1.1.1 硬件与基础设施的发展对软件的压力

- 30多年来，CPU 速度的提高一直服从摩尔定律;
- 存储密度以每年 100% 的速度增长;
- 网络带宽持续增长，接入方式多样化;
- 移动通信、蓝牙（*Blue tooth*）技术扩大了应用领域;
- 主流硬件始终没有突破冯•诺曼体系结构。

1.1 软件开发面临的普遍性问题

1.1.2 应用对软件能力提出了更高要求

- 软件能力（*Software Capabilities*）：软件开发过程中所采用的软件技术、软件工具和抽象层次等因素对开发目标所起的作用。
- 软件能力是一种相对概念:
  - 产品的规模和（或）复杂程度改变时，同样的软件技术（工具、抽象层次）就呈现出不同的作用;
  - 产品特征不变时，不同的软件技术（工具、抽象层次）也将呈现出不同的作用。

1.1 软件开发面临的普遍性问题

1.1.2 应用对软件能力提出了更高要求

- 采用更复杂的数据结构
- 采用更复杂的体系结构
- 提供用户定制及二次开发支持
- 具有更高的安全性与可靠性
- 版本间隔明显缩短

1.1 软件开发面临的普遍性问题

1.1.2 应用对软件能力提出了更高要求

- 采用更复杂的数据结构
  - 表示复杂的实体
  - 表示非结构化数据
  - 大数据
  - 非指针的关联数据结构
  - 大量、不同种类数据结构的组织与使用
- 采用更复杂的体系结构
- 提供用户定制及二次开发支持
- 具有更高的安全性与可靠性
- 版本间隔明显缩短

1.1 软件开发面临的普遍性问题

1.1.2 应用对软件能力提出了更高要求

- 采用更复杂的数据结构
- 采用更复杂的体系结构
- 提供用户定制及二次开发支持
- 具有更高的安全性与可靠性
- 版本间隔明显缩短

- 广泛采用分布式或 CS 结构或 BS 结构
- 构件化
- 广泛采用多进程、多线程的并发结构
- 几乎没有不使用数据库的应用系统
- 基于多种操作系统
- 基于异构网络平台
- Web服务、SOA
- 云计算、“智慧地球”…

1.1 软件开发面临的普遍性问题

1.1.2 应用对软件能力提出了更高要求

- 采用更复杂的数据结构
- 采用更复杂的体系结构
- 提供用户定制及二次开发支持
- 具有更高的安全性与可靠性
- 版本间隔明显缩短

- 用户可定制的 I/O
- 基于可视化语言、Script 或准自然语言的应用逻辑表示
- 系统配置与管理工具
- 易用的、多种语言的 API

1.1 软件开发面临的普遍性问题

1.1.2 应用对软件能力提出了更高要求

- 采用更复杂的数据结构
- 采用更复杂的体系结构
- 提供用户定制及二次开发支持
- 具有更高的安全性与可靠性
- 版本间隔明显缩短

- 防范黑客和内部人员的非法侵入与恶意攻击
- 数据传输和存储的加密
- 软件安全性
- 持续的可靠运行
- 不间断运行的扩充与升级
- 灾难备份与恢复

1.1 软件开发面临的普遍性问题

1.1.2 应用对软件能力提出了更高要求

- 采用更复杂的数据结构
- 采用更复杂的体系结构
- 提供用户定制及二次开发支持
- 具有更高的安全性与可靠性
- 版本间隔明显缩短

- 需求的不确定性与变更的必然性
- 互通、互操作范围的扩大
- 硬件与通信产品的发展
- 体系结构的进化
- 互联网应用的竞争压力
- 遗产（Legacy）系统的继承

1.1 软件开发面临的普遍性问题

1.1.3 残酷的市场竞争对软件企业提出了更高要求

- 满足用户对工期与质量的近乎于苛刻的要求;
- 持续提供系列化产品和优质服务;
- 面对技术人员的不稳定性与产品开发持续性的矛盾;
- 面对技术创新的必要性、风险与代价;
- 面对市场热点跟进与避免泡沫破裂的矛盾。

1.1 软件开发面临的普遍性问题

1.1.4 软件相对于硬件固有的弱势地位

- 硬件的平均重复度>>软件的平均重复度;
- 硬件具有可见的空间表现，易于检测和评估;
- 硬件具有坚实的理论基础和工业化基础;
- 直接面向应用是软件的“天职”，而硬件以不变应多变;
- 软件的质量衰减规律与硬件相反，在软件与硬件之间确定质量责任存在着困难。

1.1 软件开发面临的普遍性问题

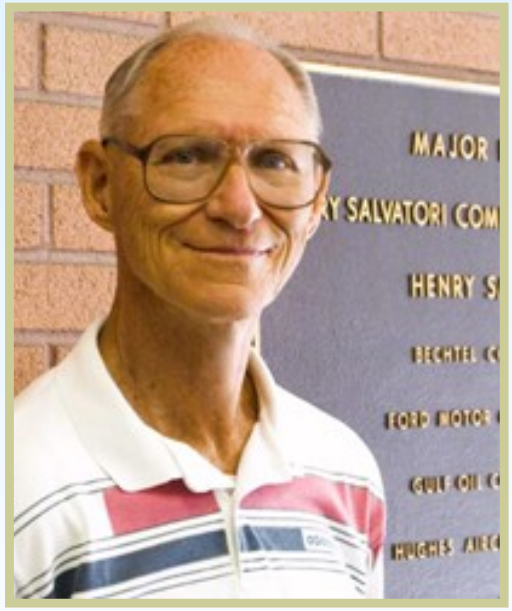
1.1.5 软件开发技术落后的现实

- 缺乏起指导作用的公理体系和基本原理，在实现之前不能准确地估计目标产品的特性;
- 成本集中于设计，而产品的制造与销毁基本上没有成本，导致了惯用手法是尝试与纠错（“*Just Try!*”）;
- 成功与失败的经验是质量保证的主要因素;
- 除了重用（*Reuse*）以外还没有更有效的提高生产率的手段。

推荐一篇论文

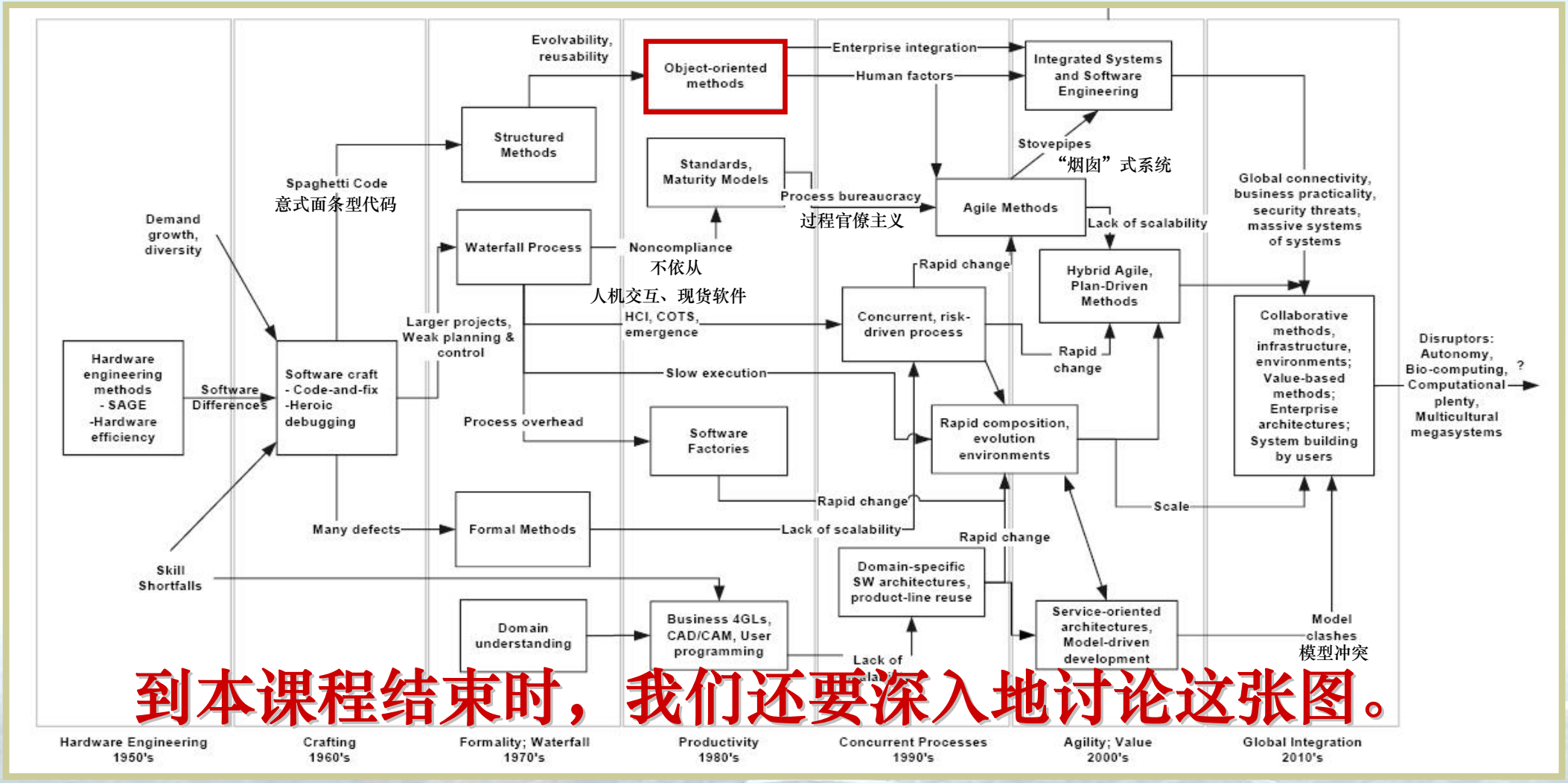
- Barry Boehm, A View of 20th and 21st Century Software Engineering, *Proc. ICSE'06*, May, 2006, pp. 12-29.
- 论文以每10年为一个单元，回顾了软件工程领域从1950年代到21世纪前10年主要的成功经验和教训，讨论了将在2010年代乃至2020年以后影响软件工程实践的变化因素，以及评价和应对这些变化因素的一些策略。
- 论文对于我们全面认识软件技术、软件工程的发展历程与发展趋势是很有帮助的，因为作者在讨论重要的标志性成果时，既阐述其正面作用，也指出其负面影响或潜在的问题。

Barry Boehm 简历



- Barry Boehm 生于1935年，1957年获哈佛大学数学学士学位，1961年和1964年分获加州大学洛杉矶分校（UCLA）数学硕士和博士学位，1992年起为南加州大学（USC）计算机科学系的软件工程教授、系统与软件工程中心主任。
- 他1955年在 General Dynamics 公司任程序员；1959 年至1973年在兰德公司工作，最终成为信息系统部负责人；1973年至1989年在TRW公司工作，任防御系统部首席科学家；1989年至1992年任美国国防部 DARPA 信息科学与技术办公室主任和 DDR&E 软件与计算机技术办公室主任。
- 他的研究领域包括：软件过程建模，软件需求工程，软件体系结构，软件度量与成本模型，软件工程环境，基于知识的软件工程等。
- 他曾被授予 ACM 软件工程杰出研究奖、IEEE Harlan Mills 奖，被马萨诸塞州大学授予计算机科学荣誉博士学位。
- 他是美国国家工程院成员，是诸多领域的顶级专业学会的会士（Fellow），包括：计算机领域的ACM，航空领域的AIAA，电子领域的 IEEE等。

这篇论文中的一张图值得认真琢磨



到本课程结束时，我们还要深入地讨论这张图。

Figure 6. A Full Range of Software Engineering Trends

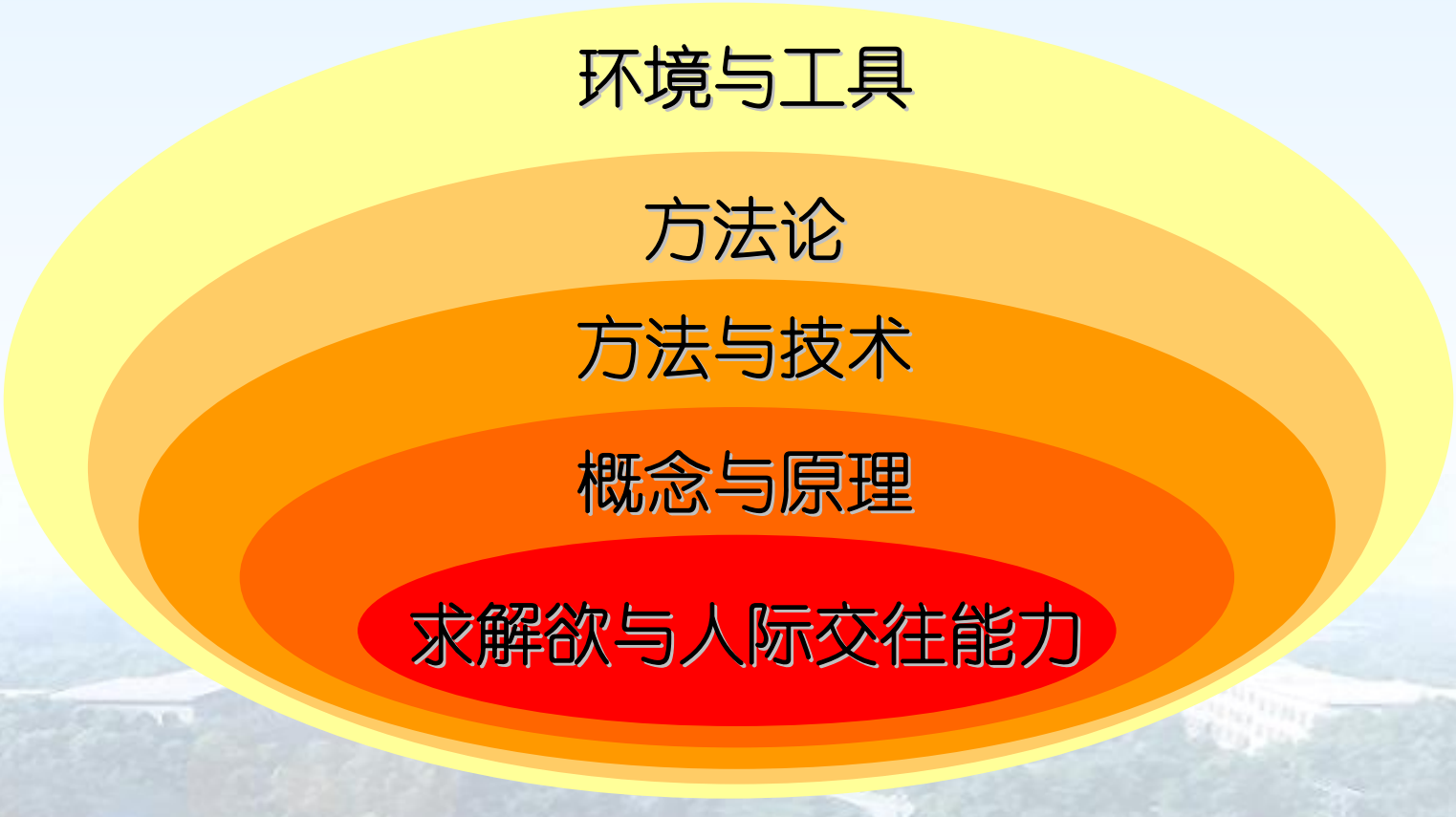
1.2. 软件开发人员应具备的基本素质

1.2.1 程序员≠软件工程师

- 上世纪末，超常的人员需求与令人不可能不动心的高待遇使大量人员涌入 IT 产业;
- 各种易学易用的工具与性能价格比越来越高的硬件降低了从事软件工作的门槛;
- 人们对软件的缺陷普遍持宽容态度;
- 人员素质优劣造成的质量差距为 20 倍的事实已不罕见;
- .COM 泡沫的破灭，使得企业真正珍惜的是杰出的职业软件开发人员。

1.2. 软件开发人员应具备的基本素质

1.2.2 系统分析与设计人员的能力结构



1.3. 面向对象技术对软件开发观念的影响

1.3.1 借鉴工业化社会成功的经验

- 构件化和重用是 200 年来工业化社会发展的成功经验;
- 应当承认，软件的确不同于硬件和其他大多数工业品（Why?），因而要实现构件化和重用有相当的难度。
- 软件产业与传统产业的共同点是：只有当产品的部件可以互换、关于产品的知识和技能可以继承的时候，才能从根本上提高生产率和降低人员变更的风险;
- 与软件产业十分类似的一个成熟产业是建筑业。

1.3. 面向对象技术对软件开发观念的影响

1.3.2 确定可行的软件工业化目标

- 传统软件工程以软件的工程化开发为目标，强调方法论、工具与环境、质量保证体系、项目管理、配置管理，但基本理念是基于具体需求、从零开始的开发；
- 软件形式化开发以软件的自动化生产为目标，强调形式化需求描述、将需求自动转换为设计、程序自动生成与验证，但形式化描述的建立是高耗时、高成本的；
- 面向对象（OO）以软件的组装式生产为目标，强调各种粒度的软件重用、接口与表示和实现分离、统一对象模型，继承和发展了传统软件工程，但目前缺少基础理论的支撑，因此从理念到实施可能失控和发散。

1.3. 面向对象技术对软件开发观念的影响

1.3.3 采用以状态保持与转换为特征的计算模型

- OO 技术将计算看成是一个系统的演变过程，系统由对象组成，经历一连串的状态变化以完成计算任务。
- 对象具有状态保持能力和自主计算能力。
- OO 设计和实现的重点是多个对象的网状组织结构和协同计算，而不是过程调用的层次结构。
- 本质上适应了并发、分布系统及互联网的计算特征。

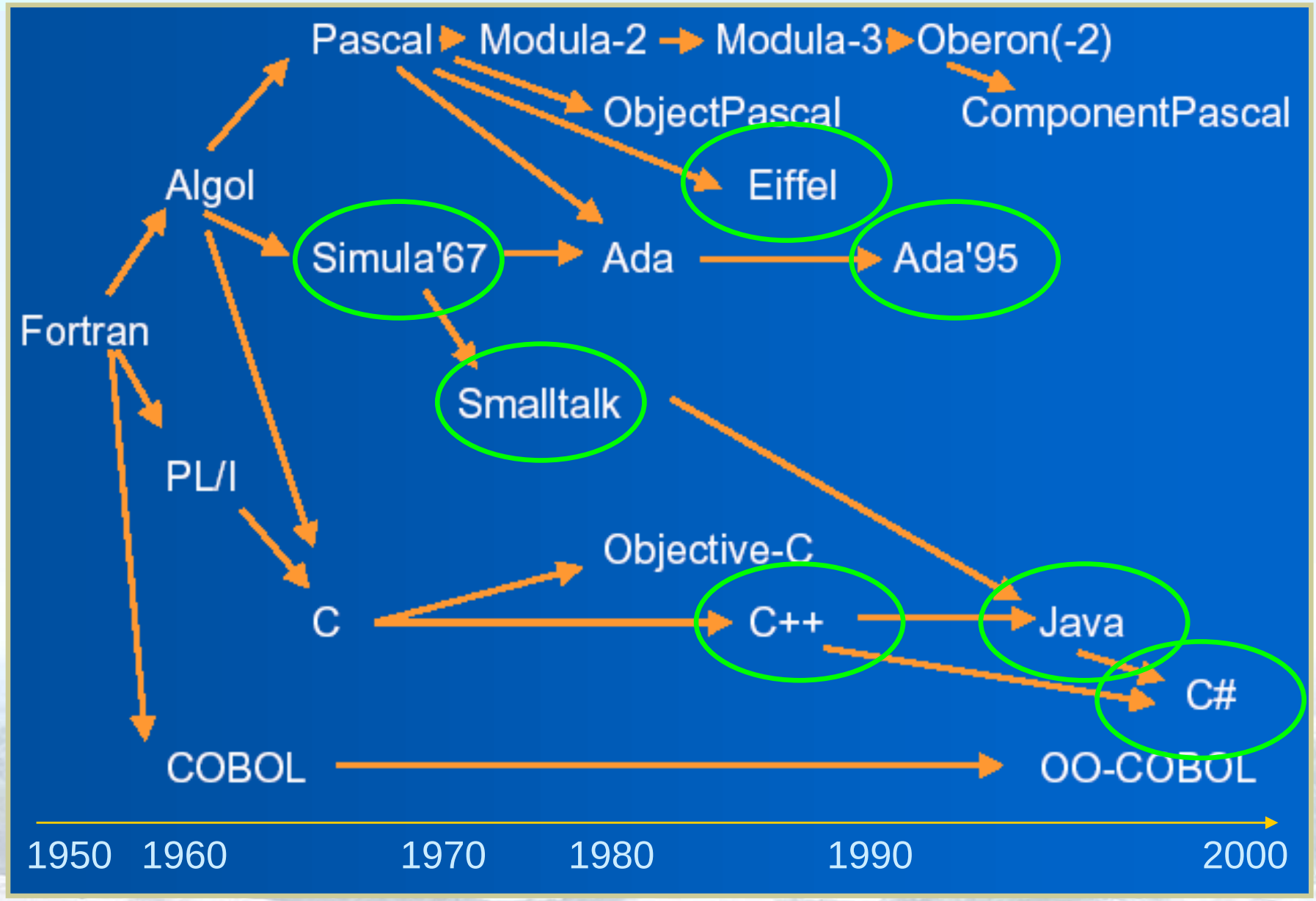
请在本课程结束时再根据自己的体会来理解上面的描述。

1.3. 面向对象技术对软件开发观念的影响

1.3.4 减少软件开发阶段之间的模型差异

- OO 技术强调需求确定、逻辑设计、物理设计、程序设计、系统扩充与维护等各阶段的模型一致性，避免转换不同模型而引起的信息损失。
- 不同阶段的模型差异在于详细与精确程度。
- 实践表明仍然存在不一致问题（以后专门讨论）。

1.4. 面向对象技术的产生与发展



几点期望

- 掌握基本概念;
- 掌握方法和语言;
- 关键是转变观念，注意境界的形成;
- 敢于提出问题，积极展开讨论;
- 重视实践环节，并善于对问题和解进行抽象。

§2. 类型系统

2.1. 类型的概念

借鉴工业化社会成功的经验

- 构件化和重用是工业化社会200年来发展的成功经验;
- 但是，软件毕竟不同于硬件，构件化有相当的难度。从五十年代以来一直在进行这方面的努力，集中表现在软件抽象层次和相应机制的发展上。

2.1. 类型的概念

程序设计语言的发展是以提高抽象程度为代表的

- **Fortran**引入了子程序的概念和机制—过程抽象;

```
extern int printf( const char*, ... );

main ()
{
  printf("Hello world!\n");
  return 0;
}
```

2.1. 类型的概念

程序设计语言的发展是以提高抽象程度为代表的

- Fortran引入了子程序的概念和机制—过程抽象;
- **Algol 60**引入了**block**的概念和机制—块抽象;

```
main ()
{
  short i;
  for (i=0; i<10; i++)
    {                    /* block begin */
      printf("Hello world!\n");
      Sleep(1000);       /* 间歇 1000ms */
    }                    /* block end   */
  return 0;
}
```

2.1. 类型的概念

程序设计语言的发展是以提高抽象程度为代表的

- Fortran引入了子程序的概念和机制—过程抽象;
- Algol 60引入了block的概念和机制—块抽象;
- **Modula-2、Pascal**引入了**module**的概念和机制—模块抽象;

2.1. 类型的概念

程序设计语言的发展是以提高抽象程度为代表的

例：字符堆栈模块

```
// 模块的接口（interface）：
namespace Stack {
  void push(char);
  char pop();
}
// 模块的使用：
void f()
{
  Stack::push('c');
  if (Stack::pop() != 'c')
    error("impossible");
}
```

```
// 模块的实现（implementation）：
namespace Stack {
  const int max_size = 200;
  char  v[max_size];
  int   top = 0;

  void push(char c)
  { /* check for overflow and
    push */ }
  char pop()
  { /* check for underflow and
    pop */ }
}
```

2.1. 类型的概念

程序设计语言的发展是以提高抽象程度为代表的

- Fortran引入了子程序的概念和机制—过程抽象;
- Algol 60引入了block的概念和机制—块抽象;
- Modula-2、Pascal引入了module的概念和机制—模块抽象;
- **Ada引入了package和generic的概念和机制—数据抽象。**

2.1. 类型的概念

为什么还是没有形成有效的重用支持?

- 程序一般不能脱离数据结构，而过程将数据结构和操作互相分离，让数据结构的设计服从操作的设计。这样，只有简单数据结构上的操作容易被重用，如数学例程。
- 数据结构是相对稳定的，功能是相对不稳定的。
- 软件总是需要在重用时适应新的变化和要求，需要有一定的柔性。

2.1. 类型的概念

能够支持重用的构件化必须先同时解决三个问题

- 以数据结构为中心的分析、设计和实现;
- 数据结构与操作的一体化表示和控制;
- 支持对已有程序无损的扩充与修改。

2.1. 类型的概念

一个基本问题

- 软件构件化的基本元素是什么？
  - 一种观点是：类型；
  - 另一种观点是：对象。
- 我们按前一种观点来进行讨论。

2.1. 类型的概念

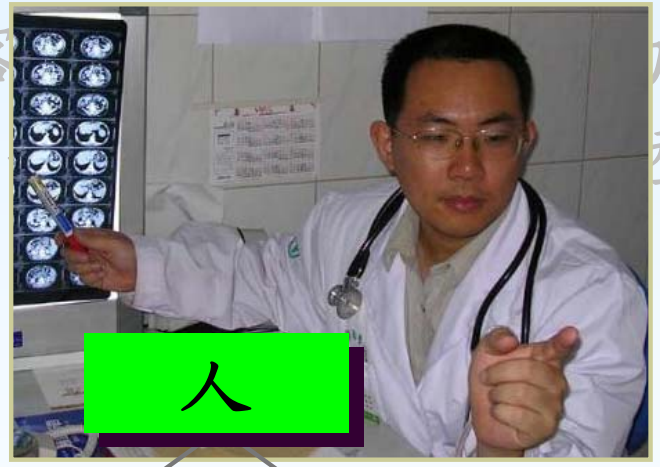
2.1.1 什么是类型?

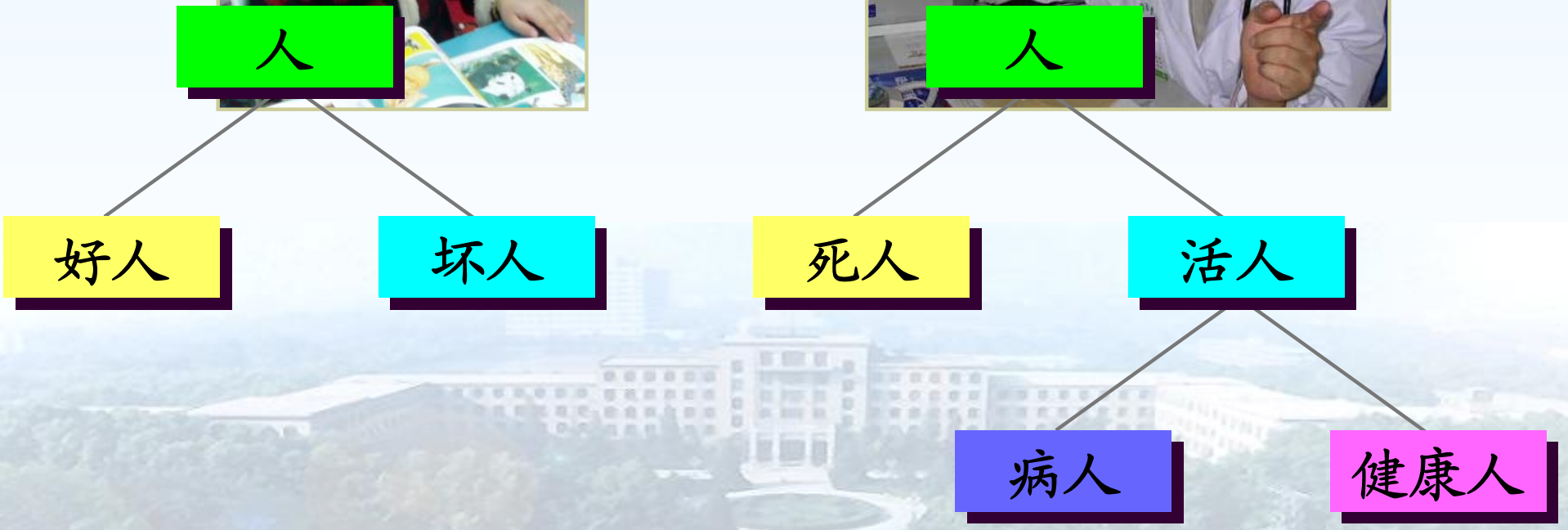
- 人们出于不同的目的，按照客观事物的特性、行为或用途，把它们分成不同的种类，每一种类就是一个类型。

2.1. 类型的概念

2.1.1 什么是类型?

- 人们[illegible]的，按照客[illegible]或用途，[illegible]同的种类，[illegible]型。





2.1. 类型的概念

2.1.1 什么是类型?

- 一个类型通常是在表示一个集合:
  - 例如: 中国人; 中国公民; 西电学子。
- 类型通常有含义明确或者约定俗成的名称或符号:
  - 例如: 飞机; 喷气式飞机; 747。
- 使用类型，可能是在使用由名称或符号对特定的集合所进行的抽象:
  - 例如: 学生; 教师; 政客。

2.1. 类型的概念

2.1.1 什么是类型?

- 就我们的论题而言，主要关心的是能够在软件中体现出来的类型，即数据类型。
- 一个数据类型 t 被定义为:
  - 一个值的集合 – 值集（记为V_t）
  - 一个由作用于该值集的那些操作构成的集合 – 操作集（记为F_t）

2.1. 类型的概念

2.1.1 什么是类型?

- 一个程序设计语言中固有的数据类型（如整数类型、字符类型等），叫做该语言的基本类型;
- 每一个基本类型的值集由语言及其运行环境所决定，使用者不能改变，而其操作集则可以由使用者扩充。
- 例如，C语言中的类型 **short:**

V_{short} = { -32768，…，-2，-1，0，1，2，…，32767 }

F_{short} = { +，-，*，/，… }

```
short myAdd (short x, short y)
{ return (x>=0 && y>=0)?x+y:0; }
```

2.1. 类型的概念

2.1.2 怎样描述类型?

值集的描述
- 枚举法—罗列所有的值
- 构造法—给出有代表性的值和构造规则
- 限定法—指明某已知值集的子集

例:

```
enum SummerMonth { Jun = 6, Jul = 7, Aug = 8 };
type   Natural = 0 .. maxint;
```

枚举法

限定法

2.1. 类型的概念

2.1.2 怎样描述类型?

操作集的描述 {
- 语法—指明操作名和被操作的数据类型
- 语义—指出操作的意义（采用自然语言、公理语义学、操作语义学、指称语义学等）

例：布尔类型 bool 的一个二元操作—与（**and**）

语法— and : $V_{bool} \times V_{bool} \rightarrow V_{bool}$

在不会引起误解的场合，这种映射习惯上被表示为类似 and : bool × bool→bool 的形式

语义— x and y = if x then y else false

2.1. 类型的概念

2.1.3 构造新的类型

- 用户要扩充程序设计语言中的类型（得到自定义类型），需要该语言提供的支持：基本类型和类型构造符。

```
typedef struct {
  int   x, y;
} a;

typedef struct {
  a     w;
  char  u;
  float v;
} b;
```

类型构造符

类型构造符

- 用户自定义类型的值集可以用相关类型的值集推出。例如：

$$V_a = V_{int} \times V_{int}$$

$$V_b = V_a \times V_{char} \times V_{float}$$

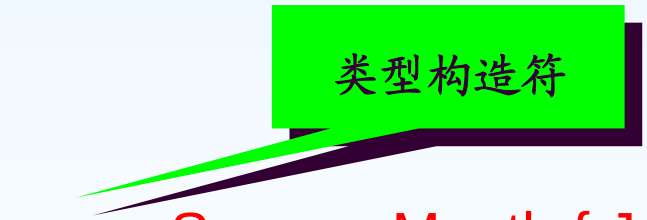
$$= V_{int} \times V_{int} \times V_{char} \times V_{float}$$

- 但是，用户自定义类型的操作集还没有固定的推导模式。

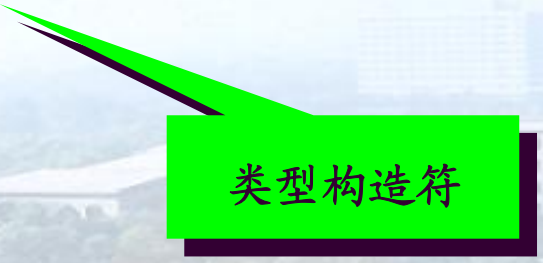
2.1. 类型的概念

2.1.3 构造新的类型

- 用户要扩充程序设计语言中的类型（得到自定义类型），需要该语言提供的支持：基本类型和类型构造符。




```
enum SummerMonth { Jun = 6, Jul = 7, Aug = 8 };
type   Natural = 0 .. maxint;
```



2.1. 类型的概念

2.1.3 构造新的类型

- 注意，构造新的类型与定义数据结构在概念上是有区别的:
  - 构造一个类型，除了描述值的集合之外，还应定义作用于该值集的所有操作;
  - 定义一个数据结构，则没有义务确定操作。
- 例如:

```
a     a1, a2, a3;
int   i;
i = a1.x + a2.x; // 合法操作，语言已提供了 + : int × int→int
a3 = a1 + a2;    // 如果没有定义 + : a × a→a，则非法
```

2.1. 类型的概念

2.1.3 构造新的类型

- 语言中一个类型 t 的定义描述（记为D_t），可以看成是一个由类型构造符所体现的、用与 t 相关的那些类型的值集来构造V_t的、有着确定语义的操作。
- 由于D_t与V_t之间的关系是确定的（由对应的类型构造符决定），因此通常我们可以在考察自定义类型 t 的值集V_t时，用考察D_t来代替（用考察构造集合的操作这样的办法来考察集合）。
- 由谁来执行构造V_t的操作？

要点与引伸

- 强调：操作集也是类型的一部分。
- 一个类型的值集通常是稳定的，而操作集可能经常需要扩充，也必须允许扩充。

下一次课的内容

- 类型系统 – 类型的概念
  - 类型系统
  - 类型化
  - 类型系统的作用
- 类型系统 – 程序设计语言中的类型
  - 类型的声明
  - 可枚举类型
  - 布尔类型
  - 字符类型与数类型
  - void类型
  - 结构化类型之一：数组
  - 结构化类型之二：记录
  - 指针类型
  - 联合类型与变体记录