

有点知识团队发售

COMP 201

复习总结

作者：周熔基，胡璟



由 扫描全能王 扫描创建

201 这门课大概可以概括为知识点复杂繁杂，但也并没有深入，所以本资料主要在于：

1. 帮助同学对这门课有一个概貌且整体的了解
2. 针对老师给的复习范围中的知识点多加复习

希望对同学们有所帮助！

这门课大概可以划分为 8 个主要的部分：

1. Software engineer 的概貌性介绍 L1
2. Software 的构建模式以及几种不同的 Software model L2-3
3. Software requirement 是什么，如何分类，如何规范，以及其重要性 L4-6 L11-12
4. System model 的种类 L7-10
5. Software design 的几种方式 L13-17
6. 如何使用 UML 图来完成 Software design L18-21
7. Software verification & validation & testing L22-24
8. Management of software project 如何管理项目的进度及预算 L25-26

老师给的考试范围：

后面的数字时题目数量，括号里面的数字代表他们属于上面划分好的某个话题

1. Use cases 5 【6】
2. Functional non-functional 4 【3】
3. Formal specification and Asml 2 【3】
4. Cost estimation 1 【8】
5. Methodology 2 【2】
6. Security including Bell-LaPadla 5 【3】
7. Testing 5 【7】
8. State chart 1 【4】
9. Petri net 6 【4】
10. Cohesion and coupling 2 【5】
11. UML diagrams 2 【6】
12. Flow graphs and testing 2 【5】
13. Class diagrams 4 【6】
14. Interaction diagrams 2 【6】
15. Project management 2 【8】
16. OO design inheritance and encapsulation 1 【5】
17. Java programming (private scope, extends, dynamic binding) 5

下面我们就开始每个部分分别讲解



目录

Part 1 3

Part 2 4

Part 3 9

Part 4 15

Part 5 21

Part 6 29

Part 7 41

Part 8 50



Part 1 Introduce

----- Software engineer 的概念性介绍

主要知识点：

1. Software engineer 基本概念
2. Software engineer 的重要性

首先来明晰四个概念，对后面也会有所帮助：

- ◆ **Software** is Computer programs and associated documentation
- ◆ **Software process** is a set of activities whose goal is the development or evolution of software
- ◆ **Software engineering** is an engineering discipline which is concerned with all aspects of software production
- ◆ **Software Process Model** is a simplified representation of a software process, presented from a specific perspective

由此可见，**software engineering** 是一个研究和指导如何进行一个完整的软件开发的学科；这里的软件开发所指的，并不仅仅是我们以前学到的编程，而是：与用户沟通需求，设计开发方案，开发软件，调整开发进度，对开发的软件进行测试，管理人员调度和预算等等。因此这门课的目的，可以理解为教我们如何进行一个完整的软件开发工作。

- Software products consist of developed programs and their associated documentation with several essential product attributes such as **maintainability, dependability, efficiency and acceptability**.

这门课教我们从不同角度来完成一个“好的”软件开发过程

学习这门课的重要性：

当我们面对用户和市场的时候，我们要考虑我们所写的东西是否是用户需要的？软件完成后我们还要考虑完成之后如何更新，维护（往往这部分的预算要比单纯的开发要多）？我们的开发成本是否超过了预算？是否能在用户需要的时间内完成？

当我们实际工作的时候，往往不是个人行动而一个团队，那么团队成员之间如何分工？如何相互协调工作？如何把握进度？

除此之外，作为一个“未来的工程师”，我们还应该遵守哪些行业道德？比如保密客户信息，保护知识产权，不在别人未经授权的情况下使用其计算机？

这些问题都会在这门课中得到明确的讲解

最后再来聊一下 Software process 和 Software process model

Software process 是软件开发过程中一系列活动，比如 Specification, Development, Validation, Evolution 等等，在后续的部分会详细介绍

Software process model 指的是表达这些活动的模型，比如 waterfall model, evolutionary model 等等，这些模型会展示不同的开发活动顺序，反馈模式等等，我们也会在后面介绍到



Part 2 Software model

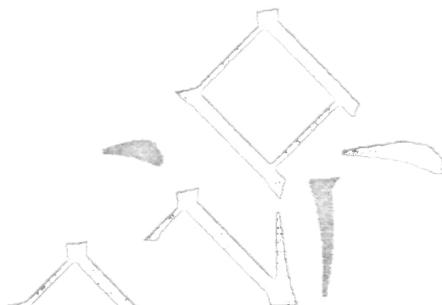
—— Software 的构建模式以及几种不同的 Software model

涉及考点:

1. Methodology
2. State chart
3. Petri net

主要知识点:

1. 什么是 process model
2. Software 的构建模式
3. System model 的种类
4. Petri model



Process Model

我们前面说了 Process 的含义 a set of activities, 例如做一个蛋糕的 process 是 {加橄榄油, 打鸡蛋, 加淀粉, 加糖, 加热}, 我们如何才能将这些 activities 经过统筹来使这个运行过程合理化并高效化呢? 这个时候我们需要 process model 来安排这些事情:

与做蛋糕同理, 这些 process 也可以用在做一个 software product, 我们把 process 称之为 software life cycle: 一般来讲 software process 分为以下这几个阶段:



当然软件的开发和完成一般的 process 有所不同, 比如软件经常会在在开发过程中间改变了需求, 所以软件也要做出对应的改变, 又或者软件需要在完成后还能做持续的更新, 所以这部分的内容就是介绍了几个我们开发软件经常会用到的 process model.

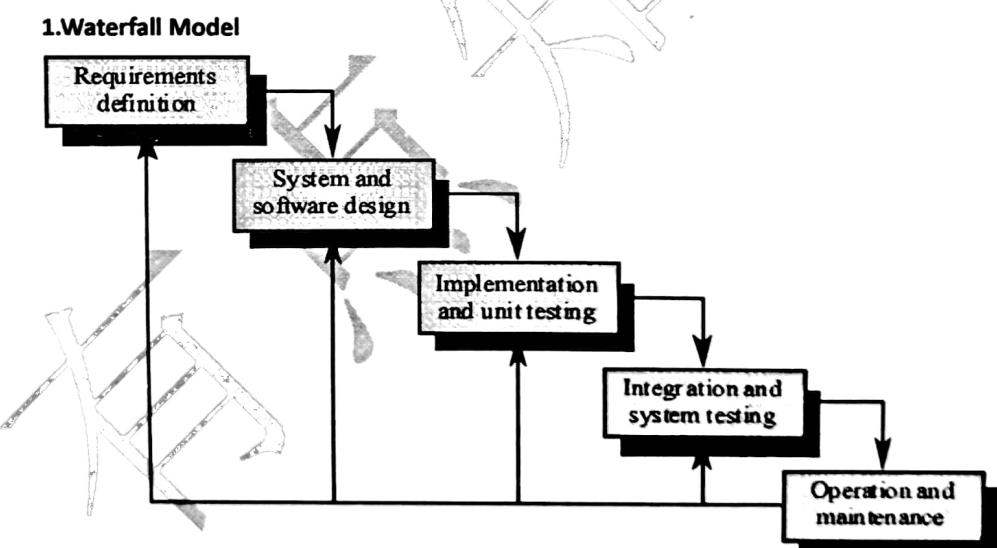
不同的 process model 会从不同的角度和不同的侧重点来适应不同软件开发的要求, 这里我们分别介绍, 一般常见的有:

- Waterfall Model
- Evolutionary/Incremental Development
- Formal System Development

下表是一个对这几个模型的笼统概括, 后面还会进行分别的详细介绍



	Waterfall	Evolutionary/Incremental Development	Formal System Development
适用系统	军用及航天航空领域	除了强调安全性的产品外都可以用	金融方面等要求安全性的产品
特点	Separate and distinct phases of specification and development	Specification and development are interleaved(相间的)	A mathematical system model is formally transformed to an implementation
缺点	如果用户需求改变，需要从头开始	无法直观看出开发进展 如果需求修改过多次会导致代码结构混乱	开发人员需要经过特殊训练，开发成本高
优点	直观看出开发进展 开发过程结构清晰明确	开发全程积极与客户沟通 并可以根据需求修改	准确性和可靠性较高，安全性也能保证



- classic engineering
- static process model
- Separate and distinct phases of specification and development

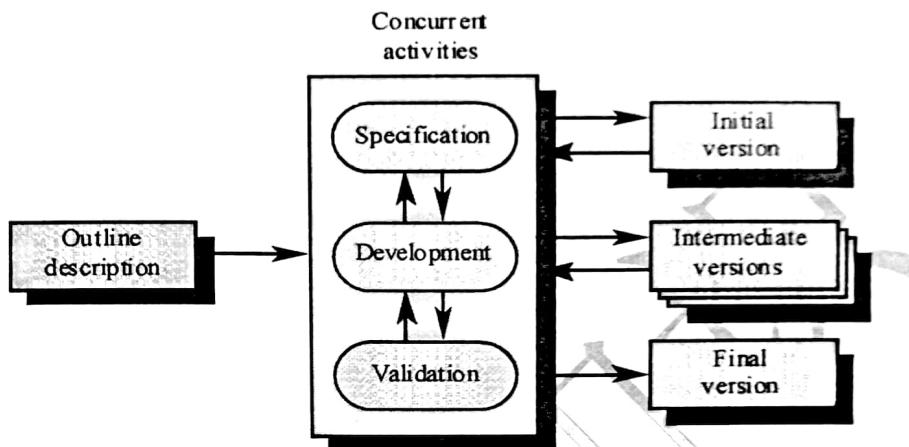
由图可以看出，软件在开发前需要计划好所有 process，不同的 activities 是一个一个完成的，前一阶段结束了才会开始新的阶段

这样会导致的问题就是当你处于后面的阶段的时候无法修改之前的问题，比如此时客户提出了新的需求，将很难及时作出更改，因此，只有在开发前十分确定需求的进程才会使用这个模型。



但是一般来讲，现实中这样的 software product 比较少见，大部分分析测试都是在写出的代码上完成的，所以在软件开发的中后期才经常能检测到问题；而瀑布模型作为一个静态模型，他忽视了不断变化的需求和用户的参与度，所以不适用于大多数开发，但仍然会用于军事及航天领域。

2. Evolutionary Development model (Incremental development)



- Getting feedback from users
- Work with the customer and evolving software though process from begin to end
- Specification and development are interleaved(相间的)

比起瀑布模型，它可以更好的和客户交互，如果客户中途改变了需求，前者需要推翻从头再来，而 Evolutionary development 可以直接作出相应的更改，从而减少了相应的开发费用。

但是这个模型也有一些缺点：因为各个 activities 之间的过多的交互，会导致开发人员无法直观的得知整个 software product 的开发进展；同时，如果中途添加了太多功能，会导致软件结构混乱，不便于管理代码，也会增加开发成本。

Evolutionary Development 尤其适用于需求框架基本提出，但是还没有明确细节的需求。同时，由于过多的更改需求会导致系统结构的混乱，所以对要求安全性很强的 software product 不应该使用这个模型。

这里还要提一下，其实 Evolutionary Development 和 Incremental development 是有一些区别的，后者应该属于前者，简单来说：

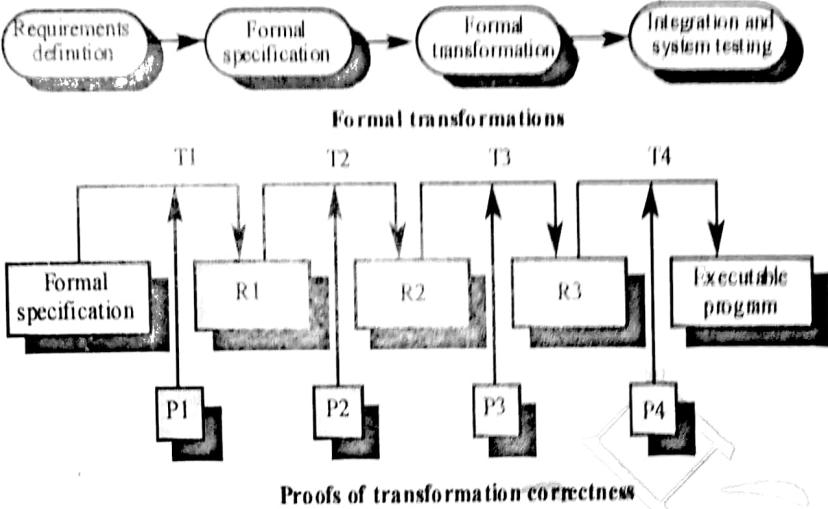
Evolutionary Development+ Process Iterate = Incremental Development

但是老师 ppt 这张图给的标题是 ED，但是在书中的标题是 ID，所以就把他们两个并列在一起说了，基本原理是一样的，问过老师说这个不用细抠，大概了解了就好。

这里的迭代不是一个模型，迭代是一种可以用于任何 generic process model 的方法，在开发现代大型系统的时候十分有用。

3. Formal Systems Development





把数学模型转化成可执行的程序，转化过程如图二，保证了较高的正确性，所以这个模型能够符合客户需求，并有着较高的准确性。

但这个模型的缺点在于需要经过特殊培训的开发者才能够使用这一模型，所以导致可能开发初始成本较高以及开发事件较长。

Process

前面介绍了 Process model，接下来我们则是明确一下，前面提到的 software life cycle 的四个主要 activities 分别应该用来做什么的。

1.Specification

软件规范，确定这个产品所需要的服务和会受到的系统限制的过程。可以简单理解为，对软件开发过程提出的产品要求。

我们需要通过客户的需求来得出合理且准确的软件规范，至于确立客户需求并转化为软件规范当然这部分我们会在 Part3 再次详细介绍

2.Design & Implementation

规范来说，This process of converting the system specification into an executable system，这个部分是把之前提出的需求变成现实的部分：

Design 的部分设计到比如设计一个软件结构，设计数据结构，接口，数据库，网站等等各种结构；当然，也有很多模型来帮助我们合理的设计一个软件，在 Part4 会再详细介绍。

Implementation 的部分主要是把这些结构变成可以运行的程序

当然在实际开发中，这两个部分往往是同时交错进行的，所以我们把他们放在一个部分来说

因为他们是交错的，所以有的时候并没有正式的设计文档来区分他们



在 implementation 的部分，同时还包含 Programming & Debugging 的部分，当然也有的时候把这一部分作为单独的活动

3.Validation

这部分分为两块：

Verification: intended to show that a system conforms to its specification （系统是否符合软件规范）

Validation: intended to show that a system meets the requirements of the system customer （系统是否满足客户要求）

这部分在 Part6 会更详细的介绍

4.Evolution

软件是可以随着需求而改的，当完成后提出其他需求的改变，这时候就是 Evolution 的部分了，这部分指当软件交付给客户后还会发生的一系列改进。

有的时候后续升级和维护软件所需要的的成本甚至会高于开发成本



Part 3 Software requirement

===== requirement 的分类

涉及考点:

1. Functional/non-functional 4 【3】
2. Security including Bell-LaPadula 5 【3】
3. Formal specification and Asml 2 【3】

主要知识点:

1. User/System requirement & Software specification
2. Functional/Non-Functional requirement
3. Software requirements document
4. Requirements Engineering Processes
5. Bell-LaPadula model
6. Formal specification



User/System requirement & Software specification

我们如何能确保所设计的软件符合用户的想法？因此我们需要充分理解客户的需求，找到并了解我们需要解决的问题，并且找到解决问题的方法；

需求的定义是很广泛，可以是简单的描述，可以是抽象化的语句，可以是约束，需求规定了系统应该做什么，要受到什么约束，应该怎样完成。从使用者而言我们可以分为一下 3 类：

1. User requirements 为客户提供

一般采用自然语言或者图表的形式来展现，通常通俗易懂但是可能会产生很多问题，比如：

---Lack of clarity 文档表达不够清晰，精确度不够

---Requirement confusion 混淆需求，比如功能性需求和非功能性需求的混淆

---Requirements amalgamation 同时表达几个需求导致需求混乱

综合而言，用户需求可能容易被理解，但是不过明晰

2. System requirement 描述系统服务

System requirements are intended to communicate the functions that the system should provide

从某种意义上来说，system requirement 比 user requirement 更详细一些，但是仍然属于不规范的需求；一般来讲，system requirement 用结构化的自然语言或者 PDL (PDL-based requirement definition)，一种类似于编程语言的语言，可以定义某些操作，但是比单纯的编程语言有更大的灵活性



3. Software specification

软件规范文件则可以说是完全为了开发者而编写的：这里可能会使用一些规范和约束使得这个文件不如自然语言那么简单易懂，但是它可以清晰明确的来表达需求，便于开发者理解和沟通。

可以理解为这部分是面向开发者的，同时具体的语言规范我们后面还会再讲解到

Functional/Non-Functional requirement

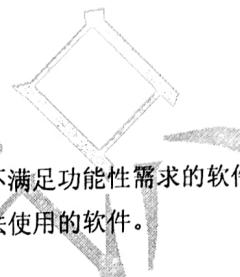
当需求陈述的不够清晰的时候，很容易发生问题，所以我们一定需要满足需求的以下两点：

----完整性 Complete：需求应该全面

----一致性 Consistent：需求之间不应该冲突

笼统而言，我们把需求分为两类

往往非功能性需求比功能性需求更为重要，因为如果不满足功能性需求的软件是一个不好的软件，但是如果不能满足非功能性需求的软件是没法使用的软件。



1. Functional requirements 功能性需求

Functional requirements set out services the system should provide

描述一个系统的行为，系统需要提供服务。功能需求可以是计算，数据处理，页面等细节

例如：计算器软件可以实现 4 位数的加减乘除运算

计算器要有清零功能

2. Non-functional requirements

Non-functional requirements constrain the system being developed or the development process

非功能需求描述设计或实现过程中的硬性条件，或者说，限制条件

例如：计算器软件在 1 秒内要完成运算

软件使用的内存不超过 10MB

语言要用 java 开发

但有的时候，有时我们很难来验证这个是否满足非功能性需求，我们需要一个明确的标准来检验非功能性需求。

Software requirements document

软件需求文档是系统开发人员所需要的官方声明，应含有规范的，完整的需求文档，来定义系统应该做什么；

对于某些模型，比如演化模型，需求可能会发生多次改变，而有些模型，比如瀑布模型，则需要在一开始就明确需求，所以我们需要需求文档来记录这些：



有很多记录需求文档的语言，比如
Forms-based approach: 支持结构化语言的最佳方案，用于描述软件系统的输入输出和功能

Insulin Pump Control Software/SRS/3.3.2	
Function	Compute insulin dose: Safe sugar level!
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 milts.
Inputs	Current sugar reading (r_2), the previous two readings (r_0 and r_1)
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose \$ the dose of insulin to be delivered
Destination	Main control loop
Action:	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requires	Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Post-condition	r_0 is replaced by r_1 then r_1 is replaced by r_2
Side-effects	None

Tabular Specification: 同样是用来协助和补充自然语言的规范语言，在同时存在多种可能性的模型中比较适用

Condition	Action
Sugar level falling ($r_2 < r_1$)	CompDose = 0
Sugar level stable ($r_2 = r_1$)	CompDose = 0
Sugar level increasing and rate of increase decreasing ($((r_2-r_1) < (r_1-r_0))$)	CompDose = 0
Sugar level increasing and rate of increase stable or increasing. ($((r_2-r_1) \square (r_1-r_0))$)	CompDose = round ($(r_2-r_1)/4$) If rounded result = 0 then CompDose = MinimumDose

需求文档的基本结构：

Preface: 定义文档的预期读者，版本历史记录以及版本更改的基本原理和更改摘要

Introduction: 描述了系统和所提供功能的需求以及它与现有系统的交互方式

Glossary: 定义文档中使用的术语



User requirements definition: 使用自然语言描述为用户提供的服务和系统的非功能性需求，客户可以理解的图表

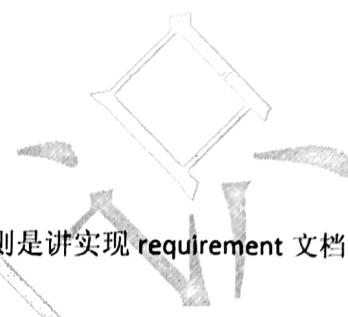
System architecture: 系统架构的高级概述，显示跨系统模块的功能分布

System requirements specification: 功能要求和非功能要求的详细描述
System models: 定义系统模型

System evolution: 描述由于硬件演变和用户需求变化等原因导致的系统的预期需求变化

Appendices: 有关正在开发的系统的详细信息，例如硬件和数据库描述

Index: 文件/图表/功能索引



Requirements Engineering Processes

前面说了 **requirement** 的意义以及分类，这里则是讲实现 **requirement** 文档的基本过程：

首先我们要进行 **Feasibility studies**/可行性研究，对项目从技术和经济上进行科学论证，检验新项目是否值得进行

1. Requirements elicitation; 研究用户需要哪些服务？

这一过程收集和系统相关的信息，并从此信息中提出有效的简明的真实需求

这一过程所有相关涉及者被称为利益相关者，一般来讲包含用户，工程师，相关领域研究人员等

2. Requirements analysis; 如何对需求进行分类并确定优先级？

这部分和上一部分是密切相关的，有的时候，利益相关者并不真正清楚他们想要什么，或者用自己的方式表达，所以难以相互沟通。

同时不同的利益相关者之间也可能存在冲突，政治因素也可能会影响系统要求

因此，我们需要对需求进行分析并进行分类，并筛选出不同优先级使需求不会产生矛盾

我们可以使用 **viewpoint** 来进行表达，将利益相关者根据不同的 **viewpoint** 进行分类

3. Requirements validation; 所完成的项目是否符合用户的要求？

我们可以从以下几个方面进行检验：

Validity 系统是否能够提供满足客户需求的功能

Consistency 需求之间是否含有冲突

Completeness 是否包含客户要求的所有需求

Realism 实施这些需求的预算是否充足

4. Requirements management. 有时候需求文档需要被修改或者升级，这时如何管理需求文档？



当完成这些步骤之后，我们基本得到了一份规范的需求文档，这时候我们也可以使用自动化测试工具，比如 Cucumber 来检验需求是否合理

Bell – LaPadula model

在保证信息正确性的同时，我们需要保证信息文档的安全性；Bell-LaPadula 这个模型是一个多级安全的机密性保护模型，所有文档有固定的权限，权限的等级会用于限制对文件的访问，这种模型可以有效的加强信息保密性，我们把文件的权限等级设置成如下 4 种：

Top-Secret 绝密（4），

Secret 秘密（3），

Sensitive 敏感（2），

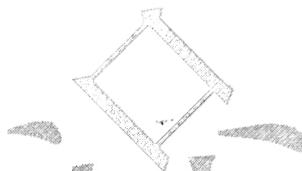
Unclassified 未分类

同时有如下两个访问规则：

No read-up: 如果我是 2 级，我就无法读取 3 级或 4 级文档

No write-down: 文档不能复制/包含在安全许可较低的其他文档中；

如果我想向 2 敏感文档添加一个 4 绝密，结果将是一个 4 绝密文档；如果我的分类是 2，我不能生成未分类的文档



Formal specification

前面提到了规范需求文档的重要性，那么我们讲一下如何规范需求文档；

这里使用形式化方法，来描述系统的软件和硬件的开发和验证，这是现代软件开发的必要的沟通方式

形式化方法可以减少系统文档中的错误，强制性分析需求来节约成本，在铁路信号系统，航天系统，医疗系统等地方具有格外多的效率

形式化方法最常使用在 waterfall 模型里面，因为 waterfall 模型尤其要求系统需求和设计要被详细表达，减少歧义，完善分析，发现潜在问题和需求中模糊的地方，从而更好的完成最终的成果

ASML

Abstract State Machine Language，这是一种对于数字系统的结构行为的建模语言；

AsmL 模型被认为是抽象的，因为它只影响建模的部分系统结构，这个模型的目标是用最少的细节来准确的描述系统行为，这以为着我们要从全局考虑，忽视不重要不相关的细节

抽象可以帮我们将复杂的问题简化，因此 AsmL 提供了多种功能来描述系统的相关状态

Abstract State Machine 优点类似图灵机，不过具有更高度的抽象，我们可以将其是



为定义遵循初始状态的一系列状态

例如下图：

Problem:

Suppose we have a set that includes the integers from 1 to 20 and we want to find those numbers that, when doubled, still belong to the set.

Informal

Solution:

```
A = {1..20}
C = {i | i in A where 2*i in A}
Main()
step
    WriteLine(C)
```

Formal
(ASML)

Example: 用 ASML 写排序语言

```
var A as Seq of Integer
swap()
choose i in {0..length(A)-1}, j in {0..length(A)-1} where i < j and A(i) > A(j)
A(j) := A(i)
A(i) := A(j)
sort()
step until fixpoint
    swap()
Main()
step A := [-4,6,9,0,2,-12,7,3,5,6]
step WriteLine("Sequence A : ")
step sort()
step WriteLine("after sorting: " + A)
```

A is a sequence (i.e. Ordered set) of integers

Choose indices i,j such that $i < j$ and $A(i) > A(j)$ (thus the array elements i,j are not currently ordered).

Swap elements $A(i)$ and $A(j)$

Continue to call swap() until there are no more updates possible (thus the sequence is ordered)

Continue to do next operation (swap()) until "fixpoint", i.e. no more changes occur.



Part 4 System model

----- System model 的分类及讲解

涉及考点：

1. State chart
2. Petri net

主要知识点：

1. System model 的种类
2. Petri model

System Model

System model 是什么呢？System model are abstract description of system whose requirements are being analyzed. 这类模型的目的便是如何把系统规范改成模型，因此系统建模可以帮助分析者理解系统功能，且便于与客户沟通；

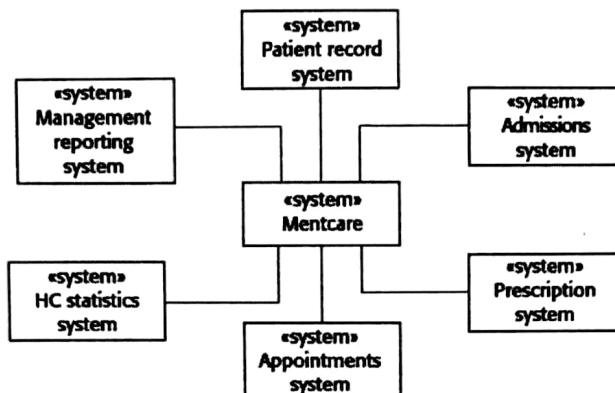
系统模型有很多优点，比如因为是图表所以简明易懂，并且可以容易的突出描述重点【当然不同模型的重点是不同的】，当然，系统模型也有一些缺点，比如，不能用来描述非功能性需求，同时可能会产生过多的记录文件，而其中可能有些记录并没有那么大的作用，对于使用者而言过于繁杂；

接下来我们分别说一下一些不同的系统模型

Context model

按照老师的说法，他觉得这些模型也没有明确的分明，大概上来讲只分为 Context Model 和不是 Context Model 的模型，并且他这部分的考点只列了 State chart 和 Petri net，所以在这之后我们只着重介绍这两个部分，其他的图如果大家想学的更多的话也可以去参考 Lecture7 的 PPT

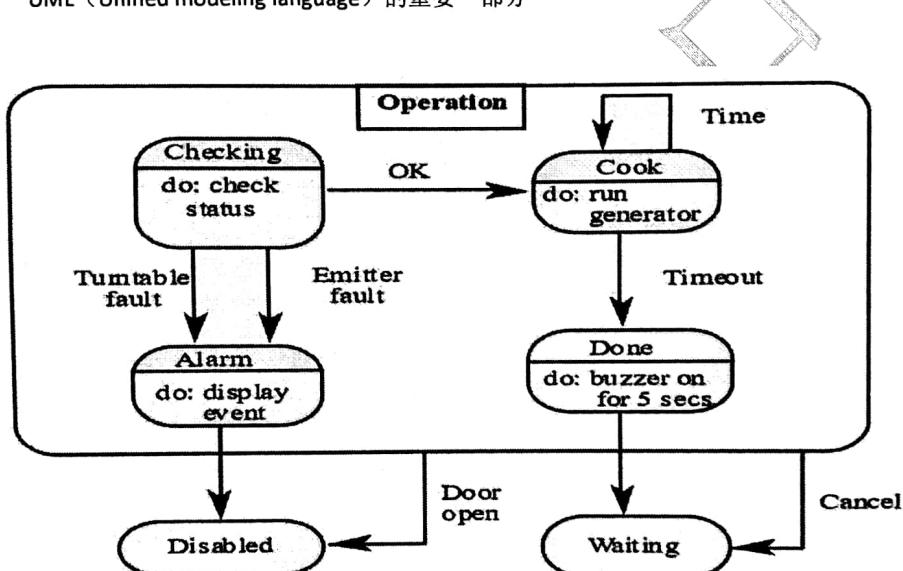
Context model 用于描述系统的边界，解释那一部分属于正在开发的系统的一部分，而哪些不是



而其他的模型，诸如 **interaction model**，注重表现用户与环境的交互，常见的表达方式比如用例图或者时序图，在后面的 UML 的部分会再次想学习讲解；
Structural model 则侧重于展示系统的结构，**Behavioral model** 则侧重于展示系统在执行过程中的动态行为等等。

Start chart

用于展示系统如何响应内部和外部事件的行为，在 **real time system** 中经常会用到，**State chart diagram** 展示的是系统的状态在不同节点及 **event** 之间的变化，也是构成 UML（Unified modeling language）的重要一部分



在这里我们介绍 **FSM(Finite state machine)**，也可以称为 **FSA(Finite state automata)**/有限状态自动机，由四个部分组成：

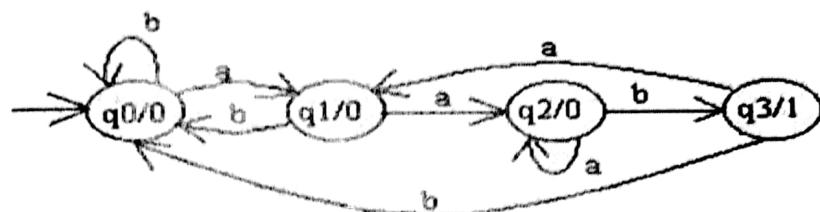
- a set of states,
- a start state,
- an input alphabet, and
- a transition function that maps input symbols and current states to a next state

这里我们分别说一下 **Moore** 状态机和 **Mealy** 状态机



1. Moore Machine:

可以理解为只比 FSA 多“输入”“输出”两个额外属性，输出与每个状态都有关系



Input: bababababab
Output: 000000100001

比如上图是一个用来计数的 Moore Machine，如果 input 里面含“aab”字节，则 output 中便含有“1”字节

2. Mealy Machine:

计算上 Mealy 有时可以等同于 Moore (我们可以用 Moore 实现任何 Mealy, 也可以用 Mealy 实现任何 Moore)，但是 Mealy 将输出函数从状态部分移到转换函数的部分，使 Mealy 更为实用。



Input: 010110
Output: 101001

上图中是把 input 的 “1” 转化为 “0”，“0”转化为“1”

简单来讲我们可以理解为，Moore 的下一状态只由当前状态决定，而 Mealy 下一状态不但与当前状态有关，也与当前输入值有关。对于 Moore，每个节点 (state) 都标有输出值，对于 Mealy，每个转换 (transaction) 都标有输出值

Petri Nets

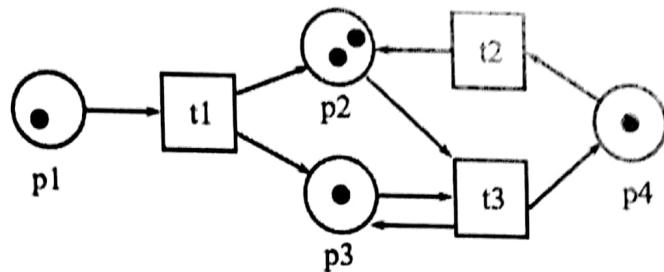
1962 年被 Carl Adam Petri 发明，最初用于模拟化学实验反应

下图中大圆圈代表 Place，方块是 Transition，小圆点是 Token

例如，对于下图的 T3 而言，input place 是 P2, P3, output place 是 P3, P4



一个 Petri Net 的 state 可以理解为 place 中 token 的数量，下图对于 (P_1, P_2, P_3, P_4) 的 state 是 $(1, 2, 1, 1)$



Enabling Condition 启动条件：

如果对于一个 transition，它的 input place 里面都含有 token，则认为它是可以被启动的

Firing 点燃？：

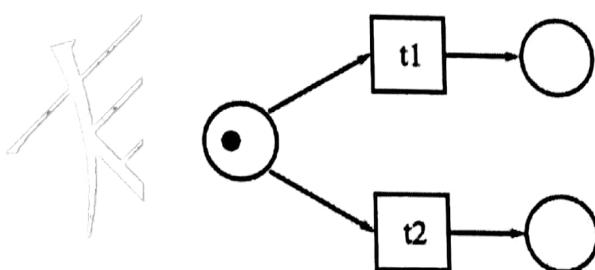
一个可以被启动的 transaction 可能被点燃，点燃的过程就是消耗 input place 里面的 token，并且在 output place 的地方产生 token

哪怕有很多个 transition 都可以被启动，但在同一时间只有一个会被 Fire

当一个 transition 没有 input place 的时候，随时都可以被点燃，每次在 output place 产生 token，当 transition 没有 output place 的时候，每次都会被点燃，消耗在 input place 里面的 token

Non-Determinism:

当两个 transition 同时争取同一个 token，他们会产生 conflict



如上图， t_1 和 t_2 同时竞争同一个 token，那么，下一个究竟 fire 那个 transition 就是随机的，因为 t_1 或者 t_2 都可以，这个就是 non-Determinism

State

分为三类

Current state: 当前状态

The configuration of tokens over the places.



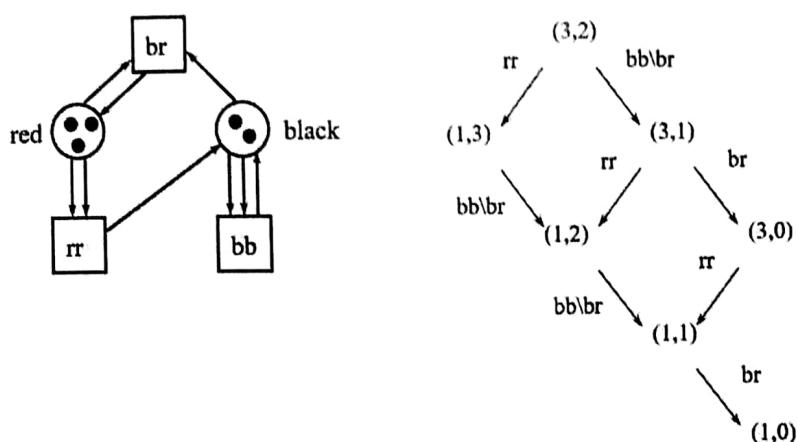
Reachable state: 从当前状态经过某些 Fire 后产生的状态

A state reachable from the current state by firing a sequence of enabled transitions.

Deadlock state: 没有 transition 可以启动的状态

A state where no transition is enabled.

如下图中, (3,2) 是 current state, 共有 7 个 reachable state, 1 个 deadlock state



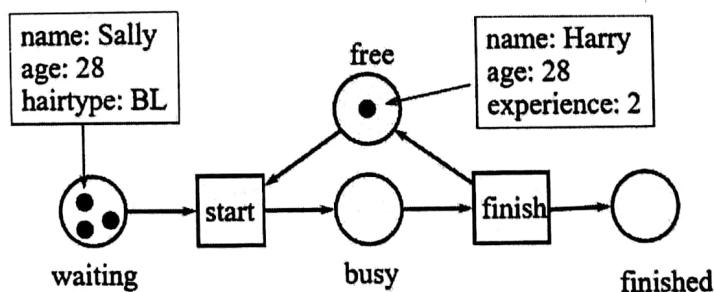
(1,0)

High-Level Petri Net

Extend with COLOR, TIME, HIERARCHY

Extend with color

每个 token 有一个单独描述它的 value, 也就是 color (这里不是仅仅指颜色哦), 如
下图

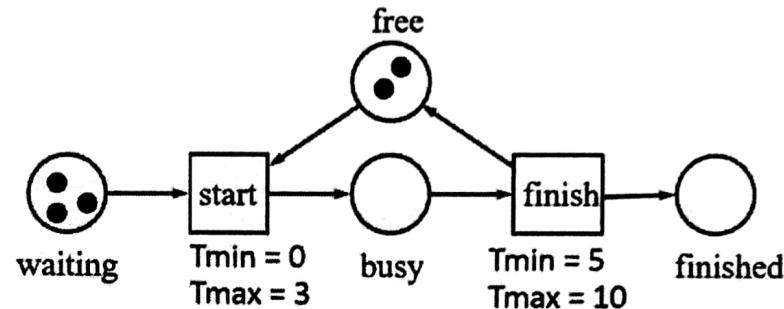


This results in a compact, manageable and natural process description



Extend with time

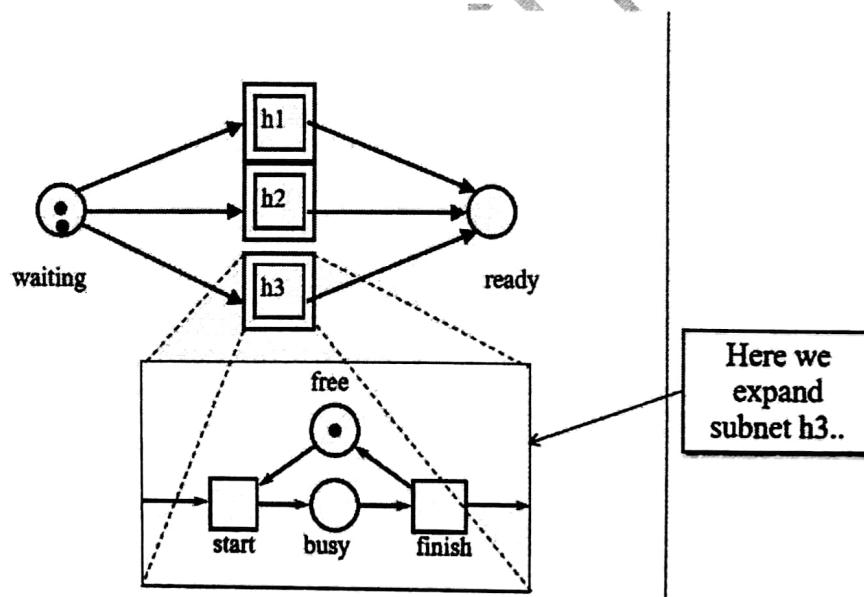
在 timed Petri net 里面，每个 transition 存在 tmin 和 tmax，如下图



这个 time 指的是当一个 transition 被启动后，fire 它的最快时间和最慢时间

Extend with Hierarchy

对 Petri net 里面的节点构造层次结构，这样在不同的抽象层次上建模，降低模型的复杂性



Part 5 Software design

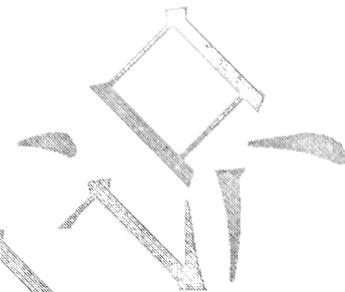
----- Software design 的几种方式

涉及考点：

4. Cohesion and coupling
5. Flow graphs and testing
6. OO design inheritance and encapsulation
7. Java programming (private scope, extends, dynamic binding)

主要知识点：

5. 软件设计准则和工作方式
6. Software 三种衡量方法 System model 的种类
7. 软件分布式结构
8. 设计过程中用到的方法



Design Methodology

这里主要先引出这一大块我们需要了解到的内容，所以概括性质比较强。我们首先考虑设计过程，在这里，写出需求文档之后如何分工是两个需要考虑的问题。实际上，设计和编程需要同步，所以最初的设计并不一定是最终的结果。为了改善这一问题，需要采用模块化。

在这之前，我们需要知道 o-o design，一种面向对象的软件化设计方法，它具有三个主要机制：

封装：隐藏内部实现

继承：服用现有代码

多态：改写对象行为

这三点将是我们未来要掌握的重点。

接下来，我们来看软件设计的 5 个准则：

- Modular Decomposability: 把问题变成一个个小问题；
e.g. Top-down design
- Modular Composability: 把模块组合起来
- Modular Understandability: 模块是否易于理解
- Modular Continuity: 模块改变不能影响整个系统
- Modular Protection: 模块异常不能影响整个系统

最后，模型建立完了，验证完了，就要存储了，一般来说，这个过程会通过子系统之间的交换和共享数据来完成。

具体的实现方法是：

(1) repository model 中央数据库

数据使用有效，集中式控制，数据模型易于看出；



但是子系统数据类型要一致，更新很昂贵，安全性不易实现

(2) **message passing** 有点像分布式数据库，一个个点状排列，互相交换数据，每个子系统都有自己的数据库，很像 207 那张曼城利物浦伦敦的图。

Design Methodo/Method

这里涉及到了一些原则和以前我们学到过的知识，只要是要把它和软件设计过程连接起来，我们写代码的时候的一些过程在这里被进行了标准化和示范化。

首先，我们来看接口，一个学了快两年的概念，抽象概念，接口不会随着模块内部的改变而改变，用户也不用了解接口内部信息；我们只要能分清 public 和 private interface 就好

接下来，我们来了解模块这里的五个原则，这个考题范围很灵活，随便插到一个考概念的选项就可以，不过在具体的实例中太难考了，不太会出。

【1】Linguistic Modular Units 模块和语言要对应

【2】Few Interfaces 通信通道的数量尽可能少，越少交换信息越少

这里不太好理解，但是逻辑上有点像上一节讲的评判标准，

不希望这里的问题影响整个系统，所以较少的通道就很好

【3】Small interface(Loose Coupling) 某一模块的更改不太会影响其他组件

【4】Explicit interface 模块的通信能够很容易看出来

【5】Information hiding(Encapsulation) 模块要有接口来隐藏属于自己的信息和操作

接着，我们引入 **Cohesion**，一个方法来检测组成成分间多么适应彼此

它的程度可以分为： Weak, Medium, Strong

课件中对 cohesion 和 encapsulation 作了对比，归纳下来表示如下：

cohesion 是一种抽象度量，这意味着开发人员不需要关心模块的内部工作。

Encapsulation 意味着开发人员无法在模块中使用隐藏信息，从而确保在使用连接的模块时不会引入细微的错误。这是两个概念之间细微但重要的区别。



由 扫描全能王 扫描创建

System Structuring and Model

这里主要涉及系统设计过程中需要考虑到的结构和由此产生的集中可能的模型，这些都是为了能为后面的分布式系统做好铺垫。学习过程中尽量用英文帮助熟悉概念，中文来对概念辨析。

在 specification 和设计过程中间有一个 Architectural design，涉及系统如何交换信息和组成，这个过程可以分为三个部分：

- {1} 分为子系统
- {2} 子系统间用模型建立关系
- {3} 子系统可以分解为独立的模块

Architectural Model 可以有着重表达，比如说 Static structural models

Dynamic process models
Interface models
Relationships models

接下来考虑系统结构，首先要分清模块和子系统，子系统本身独立，模块不算组件，只是向其他组件提供服务。

几种可能涉及到的模型：

- (1) Repository Model
- (2) Abstract Machine Model

对子系统的接口建模，分层只影响相邻层，不过其实实现很难

- (3) Control Modelling

用来关注子系统间的控制流程和模型，分为两种：

- Call-return model
- Manager model

- (4) Event-based control

每个子系统都可以响应其他来自子系统或者外部传递的事件
在这个过程中要注意，由外部事件驱动，和子系统的控制无关
也分为两种模型：

- Broadcast model
- Interrupt-driven model

- (5) Modular Decomposition

用来将子系统分为模型，也分为两种模型，Hhh

- Object model 接口，分类
- Data-flow model 转化输入输出，不适合交互



Distributed System Architecture

这里会涉及很多的模型和结构，需要分清他们在服务器还是在客户端进行操作。为了便于理解，这里需要用多个图来帮助区分，考试的时候，这种图不会让大家区分，但是作为题干或许可以帮助大家利用他的性质解决一些选项。

分布式系统架构

软件就是管理和支持分布式系统不同组件。本质上，它位于系统的中间。中间件 **Middleware** 通常是现成的，而不是专门编写的软件。

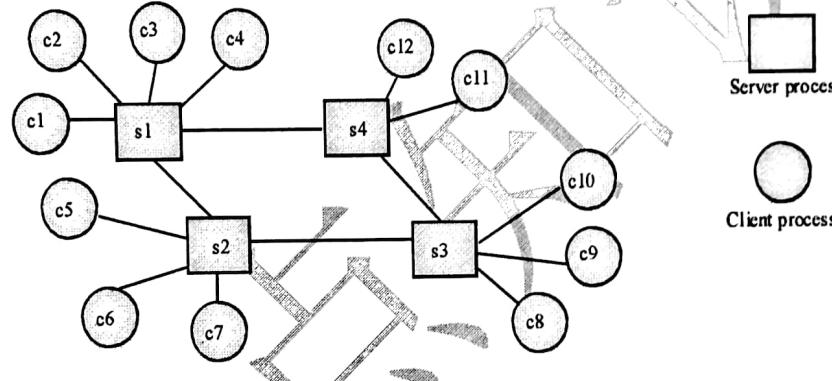
分布式系统架构可以分为 3 种：

(1) Multiprocessor architecture

简单的结构，有不同的处理器和多个进程。

(2) Client-server architecture

客户调用服务，服务器提供服务，方块代表服务器，与安全是客户端，一个服务器提供一组服务。



当然，这里只是显示了客户端和服务端间的关系，我们还需要考虑计算机上的实际处理情况，可以处理多个客户进程和服务进程。

(3) Layered application architecture

字面意思可以看出来，会分层，一共三层，分别为 presentation layer 用来输入

Application processing layer 特定的功能

Data management layer 数据库管理

这三层间是流通的，但是只有相邻的两层间可以传递和连接

图像可以自己构建出来，符合三层就好啦

架构讲完之后，我们来了解 4 个服务器和客户端上的模型和架构

(1) Thin-client model

服务器上执行程序处理和数据处理，客户端只负责演示软件

(2) Fat-client model

服务器仅用于数据处理，客户端上演示软件及于系统用户进行交互

把服务器内容减少，客户端任务加重了

当程序在本地执行的时候，任务更多的被派给了客户端。相比 Thin-client model 更为复杂。



必须在所有客户端完整安装此程序

(3) Three-Tier Architectures

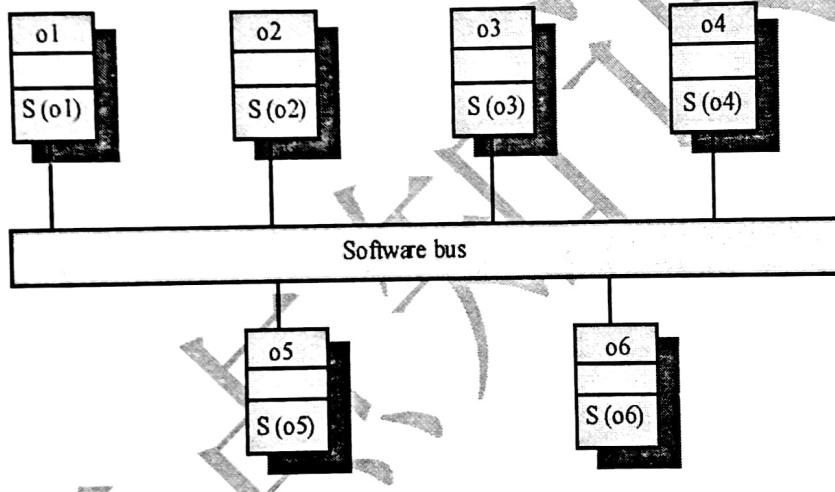
在此结构中，每个结构层可以在这层上的处理器处理，与 Thin-client model 比起来有更好的性能，与 fat-client model 比起来更好管理

同时，随着需求量的增加，有着较好的扩展性

(4) Distributed object architecture

客户端和服务器之间没有区别（对，你没看错），系统上的任何对象都可以提供服务并使用其他对象所提供的服务

从图片角度来理解，中间的横线延伸代表 Object communication，横线代表的就是我们之前提到过的 middleware system，这里他又被称为 object request broker



我们可以把每一个实体理解为一个对象，既可以为其他对象提供服务，也可以从其他对象那里接受服务

在这个过程中，对象与对象之间通过一个叫做“object request broker”的中间系统进行沟通

这个系统的缺点：设计较为复杂

优点：系统结构较为开放，扩展性好，易于添加新资源，允许不同编程语言的对象互相沟通



Concepts of Object Oriented Design

这里着重讲述了对象的行为和状态，同时初步讲述了对象的一些属性，比如 interface, inheritance, polymorphism，这种老生常谈的东西必然会考，而且如果考的细致的话，可以是出题的很多陷阱，需要留心。

首先，我们需要对对象理解一下。你可以把他看成一个用来与用户交互的东西，你给他发东西，他就给你回复，很乖。然后对象的行为要看他自己的内部状态，所以他具体怎么回复你是不确定的，只知道会回复。再有就是对象具有三个属性：behavior、state、identity，传递数据的过程用的是我们之前在第一节提过的另外一种方法 message passing。最后就是对象本身的特性，就是对象被封装并且独立，所以很容易别人维护。

下面分点来谈：

State:

一个对象当前封装的所有属性的数据，就是它的状态

一般来讲，一个对象可能由很多属性，有些是可变的，比如地址；有的是不能变得，比如出生日期；

Behavior:

可以理解为是对象的行为方式和反映方式，用改变状态来传递消息

对象可以理解信息，并对他们做出反应，这个过程的行为模式称为 behavior
但我们需要注意的是，如果一个对象的属性的值改变了，或者说它的状态改变了，但是它仍然是同一个对象

Example: 有一个 object 叫做 myClock

reportTime()

resetTimeTo(07:43), resetTimeTo(12:30) or indeed more generally

resetTimeTo(newTime)

这个对象的属性就是 time，它的其他信息会被隐藏，它的 behavior 是 time 这个信息传递给其他形象

Selector:

每个消息都有选择器，比如上面这个对象的选择器就是 reportTime() 和 resetTimeTo()，消息可以包含一个或多个参数，或者也可以不含

Interface:

对象的接口用来定义它可以接收哪些消息

一般来讲一个对象拥有两个接口：

公共接口：系统的任何地方都可以使用



私有接口：对象本身和系统的部分可以使用
 每个对象都是某个 class 的实例，每个实例的属性的值是不同，但是属于同一个类
 的实例的属性的名字和操作是一样的
 一个对象的接口必须在设计的时候就被明确，这样才有利于设计其他部分；一个对
 象也可以有多个接口

class:

1. 描述具有相同属性的一个对象集合

2. class 可以与大多数面对对象的现代编程语言接轨

Class 和 Type 严格意义来讲不是一个东西，Class 不仅定义了对象可以理解的信息
 的范围，同时定义了对象如何反映这些信息的行为模式

Inheritance, 这个是必考的，选项里可以随意涉及：

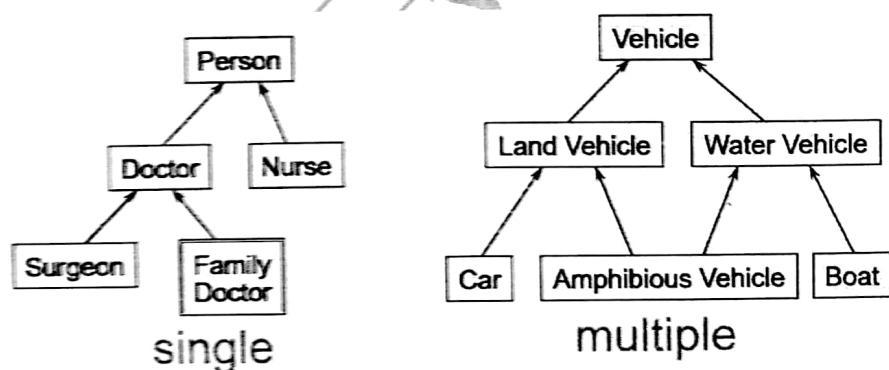
继承是指在有层次的关系中某些特定属性和操作的类的共享行为

我们可以广泛的定义一个类，再把他划分为更加精细的子类(sub-class)并定义
 每个子类包括其父类(super-class)所拥有的属性和操作，并且也有自己独特的属性
 和操作

我们可以理解为，sub-class 是 super-class 的扩展/extend

例如下图中，Person 是 doctor 和 nurse 的 super-class, doctor 是 person 的 sub-class

同时一个子类可以属于两个父类，比如右图中的 Amphibious Vehicle



当然我们也要注意，在设计的时候不应该滥用继承，如果改变父类中的一个属性的话，所以继承了父类的子类都要作出改变

Polymorphism:

允许在父类的范围中任意使用子类，可以有效的减少代码的重复性

Dynamic Binding (动态绑定):

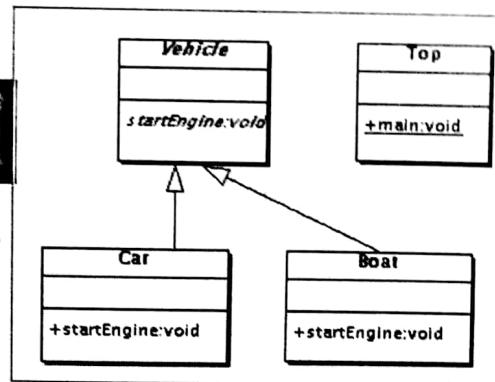
比如 C 是 B 的子类，且都有一个方法叫做 printName()



如果我们写
B temp = new C();
temp.printName();
那么将调用 C 的 printName()方法，这就是动态绑定，优先调取基础方法

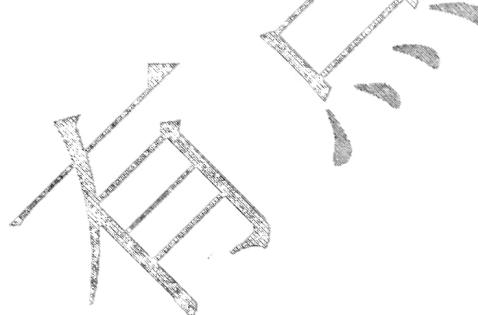
```
Vehicle v = null;  
v = new Car();  
v.startEngine();  
v = new Boat();  
v.startEngine();
```

Call Car startEngine()
method
Call Boat startEngine()
method



这里涉及的概念很多，但都很重要，需要认真研究，知道对每个概念是怎么描述的。
Cohesion and coupling; Flow graphs and testing; OO design inheritance and encapsulation 一共会出五道题，在整个考点中算中等重要。

不过对于题目的理解比较重要，因为这些基础概念很容易被涉及，所以需要引起足够的重视。这里的图有很多，可以帮助理解概念，一些概念的区别特别是之前提到的各种分类，比如分布式数据和模型，都需要借助图像来帮助大家区分清楚概念。



Part 6 Software design in UML

----- 如何使用 UML 图来完成 software design

涉及考点：

1. Use case
2. UML diagrams
3. Class diagrams
4. Interaction diagrams

主要知识点：

1. 理解功能性需求和非功能性需求
2. 看得懂 use case 图，会描述 use case
3. 分清 UML 的各种图和他们的优缺点
4. Class 图里面的概念理解
5. Interaction diagram 分清种类，看得懂每部分的指代

Use case

这是本书我们最一开始了解过的内容，因为大家已经做过 coursework，所以对于很多基础的东西应该已经了解的差不多了，但是这里也没有办法考的很难，所以这里就以给大家梳理知识为主。

首先，咱们回忆一下 功能性需求和非功能性需求

功能性需求：系统应该做什么

例如，提供一个使用用户名的登录工具和密码

非功能性需求：限制函数的提供方式

例如：用户名必须大于 6 个字符

软件必须用 Java 编写

修改翻译结果

然后咱们看看 use case 到底是啥

一个基于场景的 UML (统一建模识别的语言) 里的技术，用来定义在相互作用中的 actors 和交互本身。

一套使用 Use case 的案例应该描述所有可能的情况。

在用例图中，一个 actor 是一个系统的用户(即系统以外的事物:是人类/非人类的)表演一个特定角色。



一个 use case 的任务是哪个 actor 要在系统的帮助下执行，例如，在一个书店。
查找一本书的细节或收据的打印副本

用例图的优点：

- 更容易识别出系统需要的特定特征
- 可以忽视一些不必要的细节
- 可以更容易确定哪些 requirement 对客户更为重要
- 可以在早期开发的阶段更灵活的选择设计顺序,金钱时间方向的侧重投入等等.这样
- 可以更有效的完成 project

这里的话要注意，use case 对细节考虑很多，
每个用例的细节也应该是用例描述的一个文档

例如，

打印收据的时候

顾客通过电子邮件付款一种有效的付款方式。收银台应该印表示当前日期和时间的收据价
格、付款方式及工作人员谁处理了这笔买卖。

这时候就要写(备用情况)没有印刷纸

print 请到收款台输入“new till paper”。10 秒钟后再打印一次

最后，关于 use case 辨析一个点

Include: 指的是一个 use case 通常是 main use case 的一部分

Extend: 当一个 use case 有时会被需要的时候

然后就是我们在写作业的时候用到的几个描述方法：ID, name, Description, Pre-condition,
Event flow, Extension points, Triggers, Post-condition



Introductory Case Study

这里给大家展示一下类图，也是我们作业设计过的东西。比起协作图更加具有复杂性，主要是在于类的太多，有的时候不好把握意图。

首先我们先看看类图到底是什么。

我们可以把它归结为一通过描述对象之间的工作关系来理解系统或者确保我们的设计遵循模块化设计的原则。

简单来说，类图主要用于描述系统的结构化设计，可以显示类、接口以及它们之间的静态结构和关系。

其次，我们先来明确如何找到一个系统的 class【类】。

一般来讲，我们的系统需要满足一下两个条件：

1. 尽快且花费尽可能少的建立一个可以满足当前需求的系统。
2. 建立一个未来较容易维护及拓展的系统。

为了满足这两个条件，一个好的 class model 的主体类应该由较不容易改变需求的，不依赖于特定功能对象组成的类。[同时请注意取类名的时候也很重要，类似 java 变量命名，尽量选用不重复且有意义的名字]

现在主流有两种方法来检测一个系统究竟需要多少类，DDD 和 ROD。

1. Data-driven-design：识别系统所需的数据，并将这些数据划分为类，然后再将相对应的 method【也可以叫 responsibility】分给这些类。
2. Responsibility-driven-design：确定系统中的所有 method，并且将它们划分为类；在划分好后，在针对每一个类找预期匹配的数据。

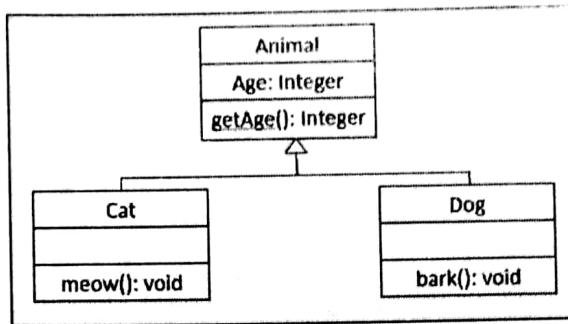
我们从一个简单的地方入手，来讲几种常见的 relation。

1. 泛化【带三角箭头的实线，箭头指向父类】：是一种继承关系，表示一般与特殊的关系，它指定了子类如何特化父类的所有特征和行为。例如：猫是一种动物。

为了帮助理解，接下来的内容主要会以图来呈现给大家，加深记忆。

如图，就是一个典型的泛化：





在这里，animal 是 cat 和 dog 的泛化，cat 和 dog 两个类都可以调用 animal 中的方法 getAge()。但是 animal 不可以调用 cat 中的 meow() 或者 dog 中的 bark()。当然泛化会增加系统耦合度（class 的独立性较差），例如如果更改了 animal 类，那么可能也需要更改 cat 和 dog 类。

2. 实现【带三角箭头的虚线，箭头指向接口】：表示类是接口所有特征和行为的实现

3. 关联【带普通箭头的实心线】：也就是上图我们用过的最普遍，表示两个类有关系。这样的话，就可以得到一个很清晰的静态关系。

如果说在系统描述里面，名词时类的话，那动词就是关联了。

我们可以理解为它是展示类与类关系的最基本的链接。

下面几种情况我们都可以称作 class A 与 class B 之间有关联：

an object of class A sends a message to an object of class B

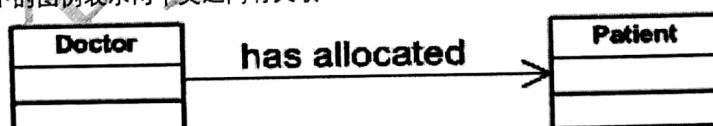
an object of class A creates an object of class B

an object of class A has an attribute whose values are objects of class B or collections

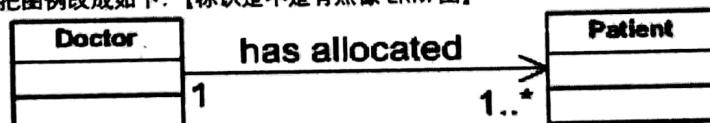
of objects of class B

an object of class A receives a message with an object of class B as argument

我们用如下的图例表示两个类之间有关联：



如果我们想表示更清晰的关系，比如，一个医生可以分配一个病人或者多个病人，那我们可以把图例改成如下：【标识是不是有点像 ERM 图】



注意右侧病人那里标注的 1 .. *，代表对应病人数量是 1 到无穷大间任意一个数字。如果表示比如 1-10 个则是 1..10；如果表示一对一关系则是 1。这个大二就学过，问题不大的。

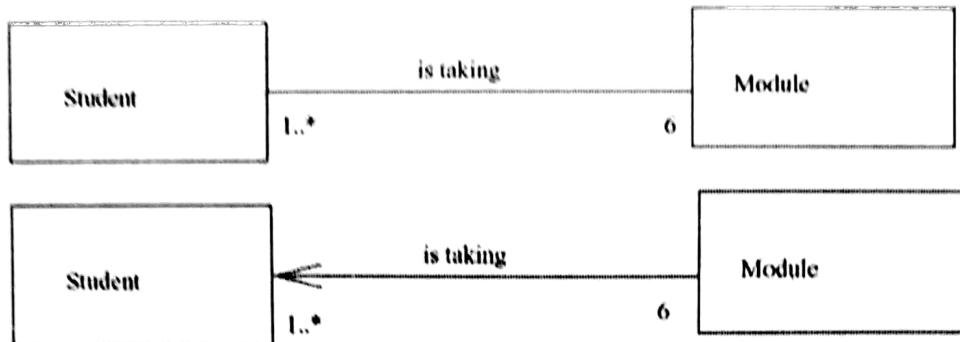


这里可以把关联分类，按照每个关联的特性

有导向性的关联和无导向性的关联

第一个表示双向关联，每个学生关联 6 个 module, 每个 Module 关联 1 至任意多的学生

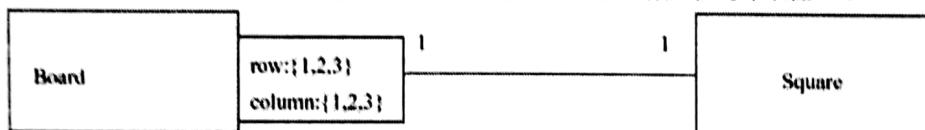
第二个关联则是单向的，只能是说 Module 知道学生，但是学生不了解 Module



还有另一种划分方法，是根据有没有仔细的限定条件和他们之间细微的方式

Qualified Association

给 association 一些更明确的限定条件，例如下图，之前只描述了 1 个 board 有 9 个 square 的关系，下图则更明确 square 由 row 和 column 两个 attribute 限定，每个取值 1-3

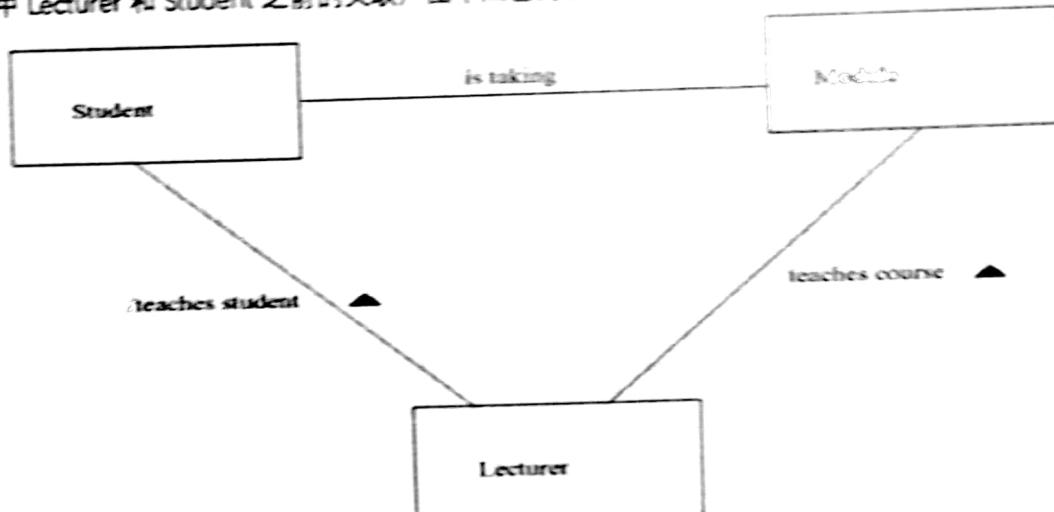


同理，我们也可以拥有 qualified composition 关系



Derived association 派生关系

一旦我们记录了主关联，派生关联会自动生成。派生关联的前面会自动添加斜杠。就如下图中 Lecturer 和 Student 之前的关系：图中黑色的小三角表示关系方向



4. Aggregation 聚合 Composition 组合

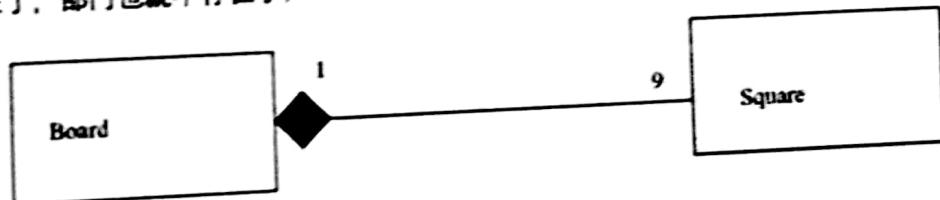
这两种关系都是记录一个类的对象是另一个类的对象的一部分，他们很容易搞错，需要仔细分清楚

Aggregation 可以理解为是整体和部分的关系，例如，一个部门由多个员工组成。与 Composition 的不同在于，整体和部分没有强依赖关系，部分可以脱离整体存在，比如即使

部门解散了，但是员工仍然存在。示意图如下：



而下图则是 Composition 关系，还是上面那个例子，公司和部门就是组合关系，如果公司不存在了，部门也就不存在了；



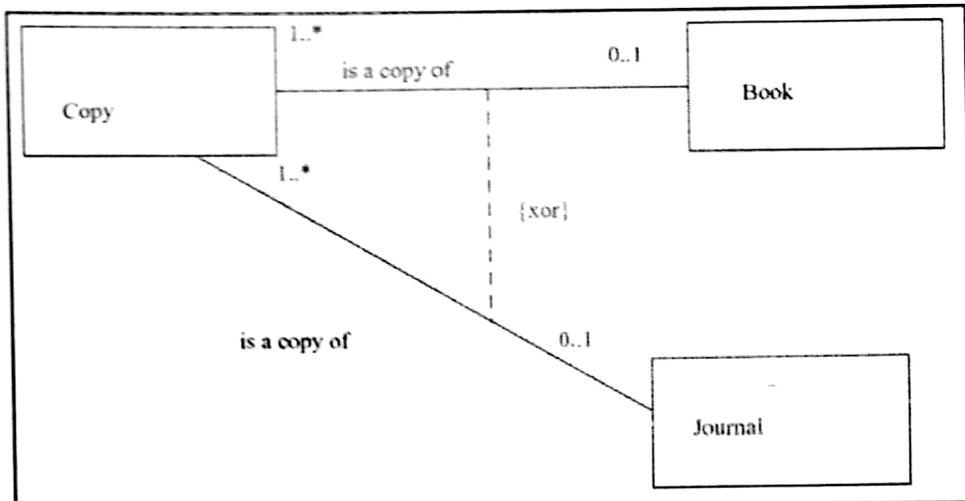
看完 relation 的分类以后，我们看看两个 relation 之间的关系：



Constraint 约束

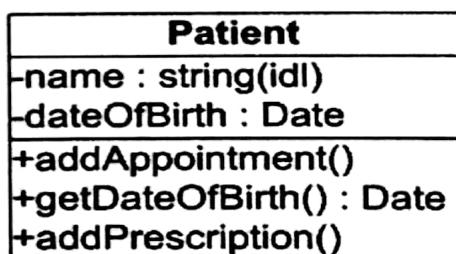
约束存在于两个 relation 之间，这里介绍比较常见的 Xor constraint

再没有添加约束之前，可能有一个 copy 即使书又是期刊，但这是无意义的，所以我们给下图中的 is a copy of 定义了 Xor 约束，确保这个 copy 要么是书，要么是期刊，二者只能选取其一：

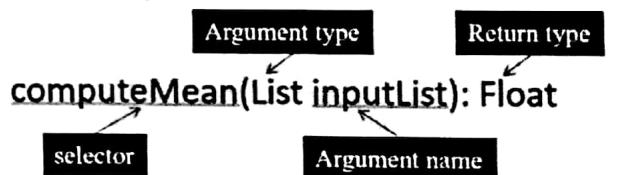


既然要画图，就要知道 attribute，这个之前就写过，只要知道怎么找就行了，除非考的很怪，不然不会考出什么没见过的东西

Attribute 主要用于描述对象中的包含的数据及操作，这些定义了对象的交互模式，如下图



下面这个叫做 operation 的签名，和 java 里面 method 的签名类似，标记了参数类型，名称，返回值等：



这里就讲完了类图，然后有一个和类很有关的概念，CRC Card



CRC Card

代表 Classes, Responsibilities, Collaborations

虽然 CRC 并不是 UML 语言的一部分，但是可以指导其设计并改进

CRC 卡是一个标准索引卡集合，包括三个部分：类名、类的职责、类的协作关系，每一张卡片表示一个类，如下图：

LibraryMember	
Responsibilities	Collaborators
Maintain data about copies currently borrowed	
Meet requests to borrow and return copies	Copy
Copy	
Responsibilities	Collaborators
Maintain data about a particular copy of a book	
Inform corresponding Book when borrowed and returned	Book
Book	
Responsibilities	Collaborators
Maintain data about one book	
Know whether there are borrowable copies	



一般来讲，一个 class 应该有 1-2 个 responsibilities，一般不超过 4 个，太多的 responsibilities 意味着系统的抽象级别太低；同时，太多的协作关系也是不好的，这代表系统的耦合度太高，连接了过多其他类。比如下图就代表一个设计的不太好的类

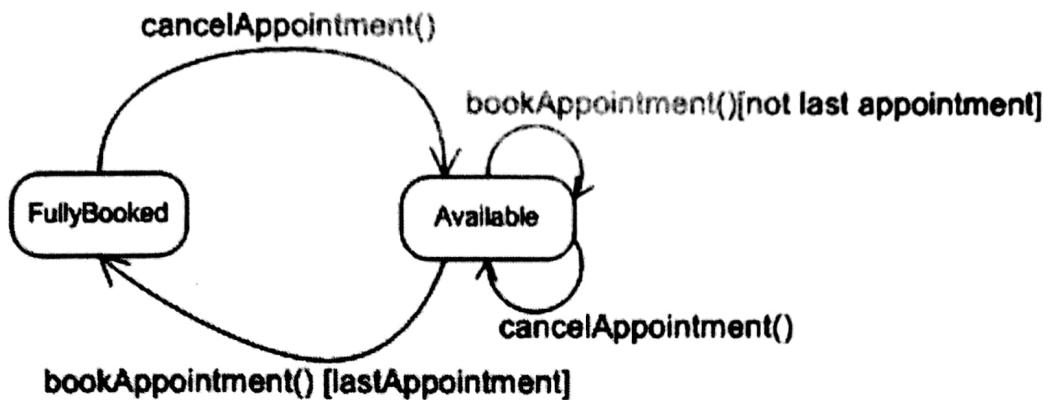
Word_Processor_Object	
Responsibilities	Collaborators
1. Spellcheck the document 2. Print the document 3. Open a new document 4. Save the document 5. Email document	Dictionary Printer File I/O Networking API

然后有一个边缘概念： state diagram

这里对它有所了解比较好，这个考点没有明确说明，但是可以和活动图对比，所以就也要了解



系统中的对象都具有一个状态，这个状态封装了它当前的数据，通过状态图建模来模拟状态的改变，如下图



Interaction diagram

UML 有两种交互图,一个是序列图 Sequence diagram,一个是 collaboration diagram 通信图,也叫做协作图

Sequence diagram

相比于其他几种图,时序图的主要优势是时序图格外适用于对复杂交互系统进行建模,简统来讲,时序图的垂直维度表示时间,水平维度表示不同的对象或者角色;

时序图主要有这几个基本元素

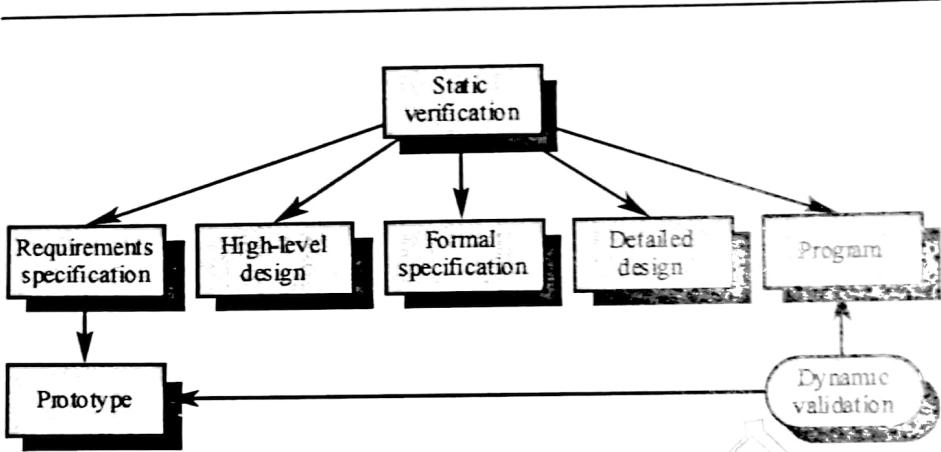
对象 Object: 表示的是时序图中的参与者,例如上图中的 receptionist, patient 等

生命线 Lifelines: 每个对象的底部都有 lifelines, 表示了对象存在的时间长度

控制焦点 Focus of Control: 用细长方形来表示,表示某个队形执行某段操作的时间

消息 Messages: 分为同步消息,异步消息,返回消息,图中即是横向的箭头





如上图，这里静态检验在五个部分进行了实施，需求规范、高设计、形式规范、细节设计和规划，非常符合我们之前学习的制作顺序，所以这里的概念是相通的。对于这个图而言，当系统没有开发完成的时候，只能进行静态检验，只有当 program 完成后得到可运行的版本，才可以进行动态检验；

因此，**incremental development** 的优点是可以在开发早期得到 program 的初步雏形，尽早进行动态检验

动态检验的重要性在于可以检验出非功能性需求

静态验证和动态验证是互补的，两者都应该在 verification 和 validation 的过程中使用

很简单，对于 verification 和 validation 我们只需要了解这么多，接下来，我们了解本书的重点内容 testing

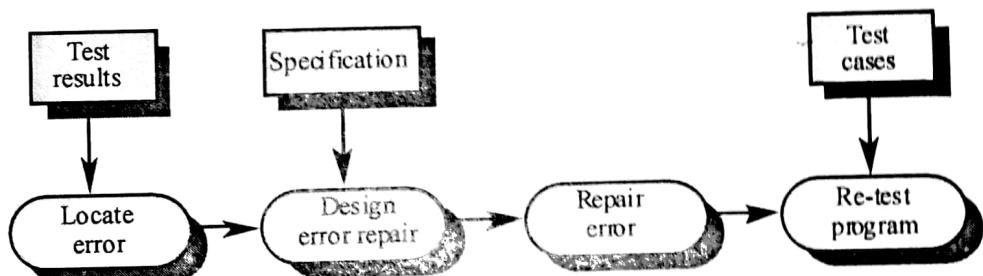
概念入手，testing 的种类，大致可以分为两类，当然这里能考的只有区分，不会在其他地方设计

Defect testing	缺陷测试，用于揭示系统中所存在缺陷
Statistical testing	统计测试，反应用户输入的频率，用于软件可靠性评估

当然，即使通过测试的软件也可能有缺陷，但是处于其他一些原因比如市场环境较为宽容，软件开发预算不足等原因，使我们可以包容这些缺陷

了解了 testing 的概念之后，继续对 testing 进行了解，我们经常容易把 testing 和 debug 混为一谈，所以我们说说 testing 和 debug 的区别，这两其实根本不是一个东西，前者用于确认程序中是否存在缺陷，就是所谓的古老年代的“问题”，我把它理解为 bug，后者用于修复这些错误，也就是 debug 了，下图则是 debug 的过程图，着重于讲解 locate error 和 repair error





同时，debug 一般搁在软件开发过程中较后面的部分，而 testing 则应该从开始就规划到开发过程中，这就和写代码的时候写一写测一测，写完了再 debug 是一个道理，所谓的 test planning 就是用于定义这个测试过程【而不是描述如何产品测试】

刚才既然提到了 test plan，我们就说说 test plan 的结构，可以有四点

The testing process	测试过程的主要阶段的描述
Requirements traceability	确保所有用户需求都经过了单独测试
Tested items	被测试的项目
Testing schedule	分配用于测试的时间和资源的安排表

Software Testing

接下来，本节全都学习这一个知识点，关于 testing 的每个部分都可以做为考点。上一节我们对 testing 概念有了了解了，然后开始深入他。

首先我们知道一下 testing 的过程，具体是怎么操作的

可以分为组件测试 Component testing 和集成测试 Integration testing 这两个部分

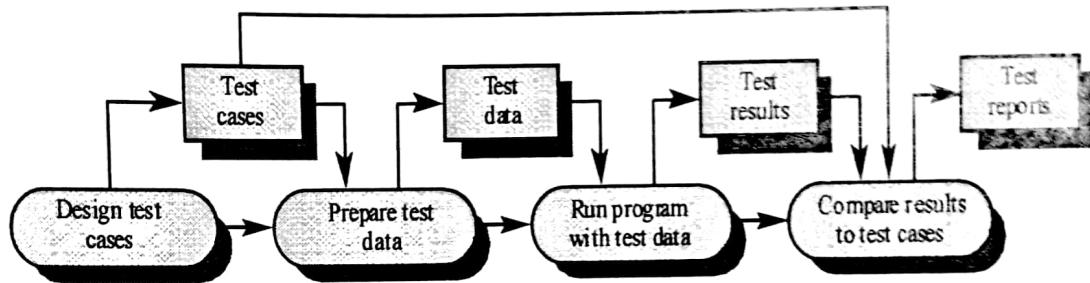
前者测试 program 的各个组件，通常是软件开发人员的工作；

而后者是测试几个继承的组件所形成的系统，一般由独立的测试团队来完成这一工作；

然后细化他，我们了解 defect testing/缺陷测试

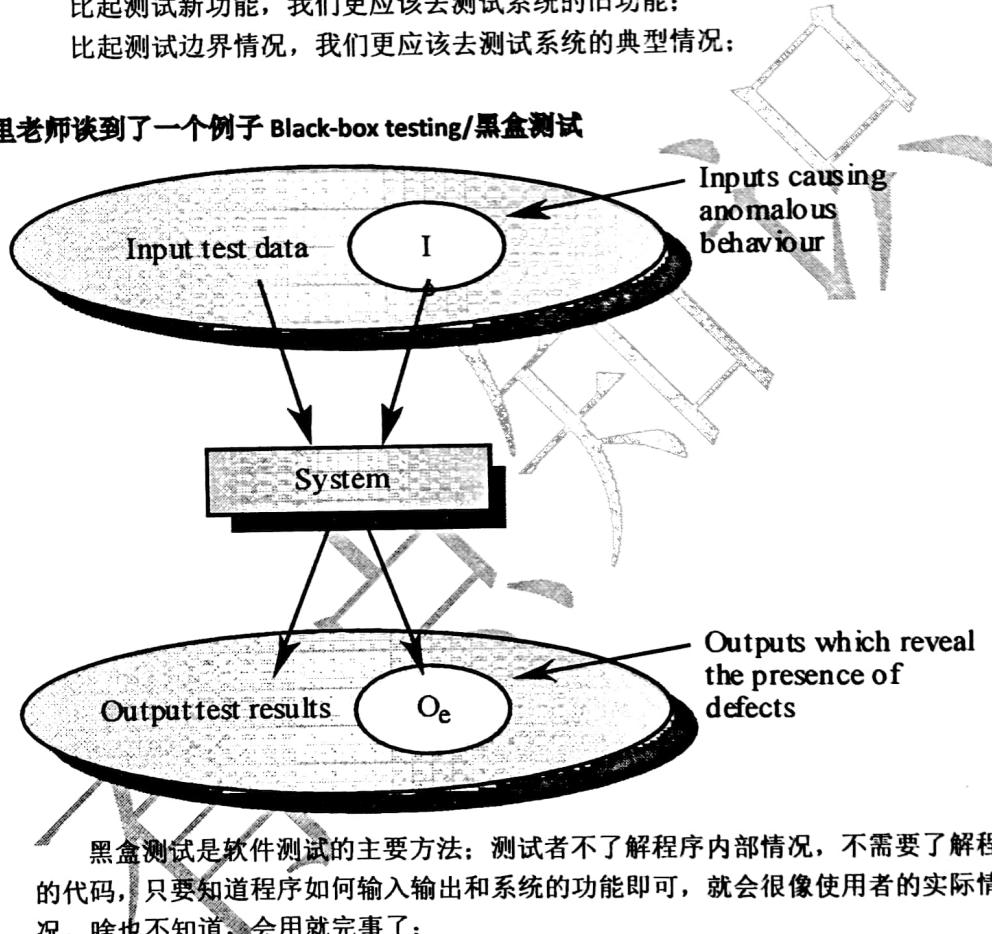
下图是 defect testing 的基本流程





缺陷测试的目标是发现程序中的不足，只有详尽的测试才能证明程序没有缺陷，但是实际上，彻底的测试是不存在的，所以我们要有**侧重点的去测试**
 比起测试新功能，我们更应该去测试系统的旧功能；
 比起测试边界情况，我们更应该去测试系统的典型情况；

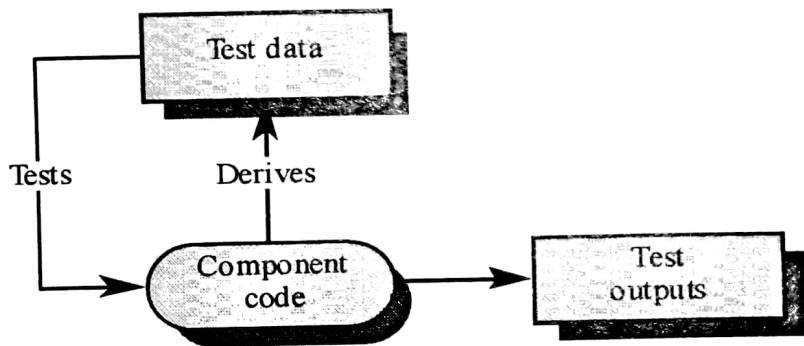
这里老师谈到了一个例子 Black-box testing/黑盒测试



黑盒测试是软件测试的主要方法：测试者不了解程序内部情况，不需要了解程序的代码，只要知道程序如何输入输出和系统的功能即可，就会很像使用者的实际情况，啥也不知道，会用就完事了；

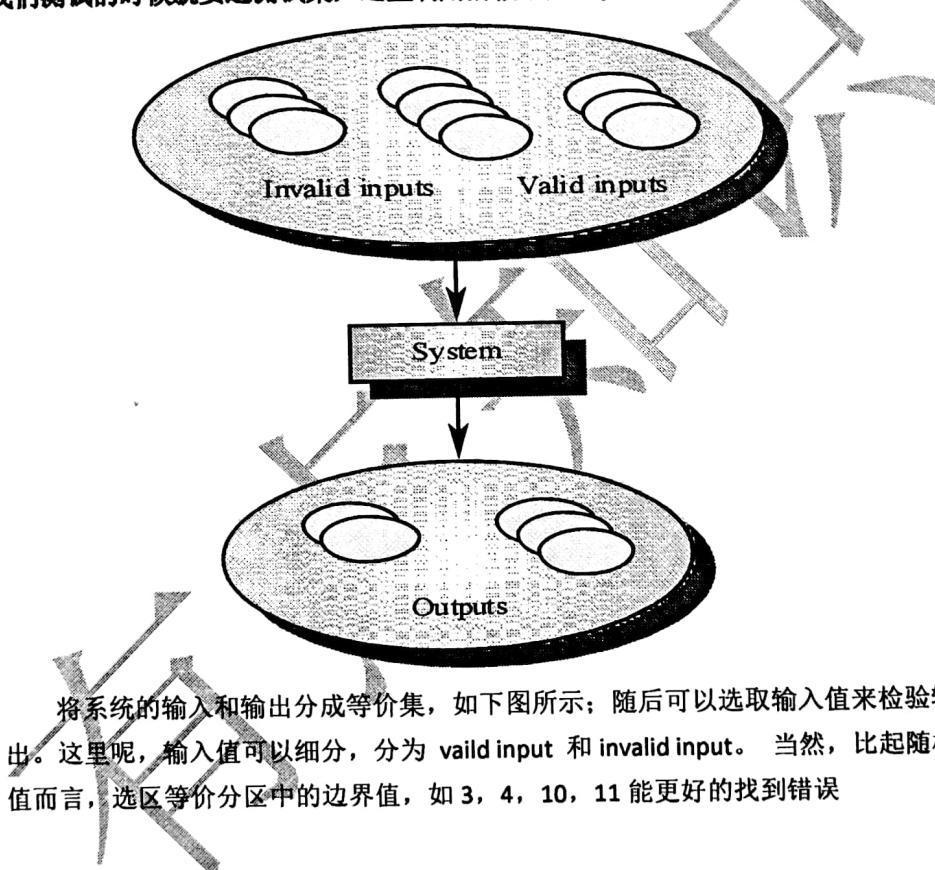
黑盒测试是从用户的角度进行测试时，而于此对应的白盒测试则是从程序设计者的角度对程序进行测试，下图则是白盒测试的示意图





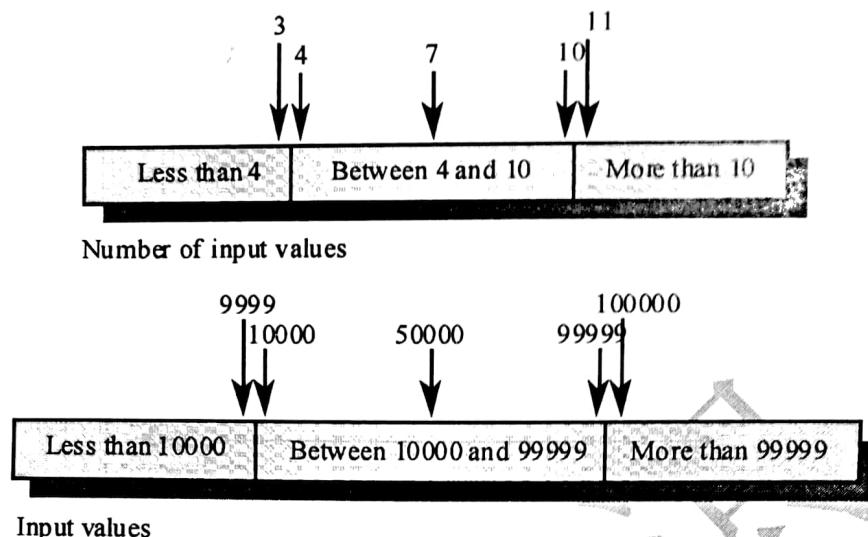
白盒测试也叫结构测试，可以根据程序结构推导测试用例

然后我们测试的时候就要选测试集，这里利用的概念是 Equivalence Partitioning/等价划分



将系统的输入和输出分成等价集，如下图所示；随后可以选取输入值来检验输出。这里呢，输入值可以细分，分为 valid input 和 invalid input。当然，比起随机选值而言，选区等价分区中的边界值，如 3, 4, 10, 11 能更好的找到错误





然后是一个和前面关系不太大的概念，也是用来测试的一条途径，叫做 Path testing
用来确保测试用例集把程序的每条路径至少执行一次；

Program flow graph: 用于描述程序控制流程，每个分支我们即可以理解为路径，而
Path testing 则是要历遍这些路径

Cyclomatic complexity/循环复杂度: "Number of edges - Number of nodes + 2"

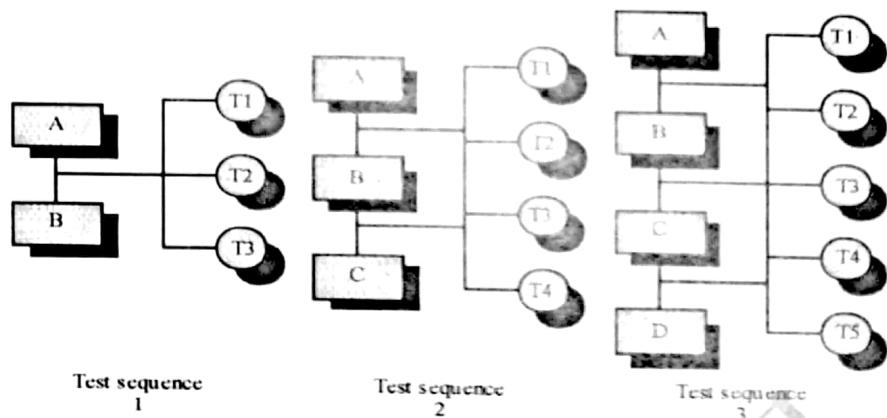
当然我们要注意，有的时候即使我们测试过了所有的路径，但是没有测试过所有的路径
组合，所以可能依然会导致错误

然后我们开始了解五种不同类型的测试，这里可以随意出题，灵活性很大

第一种 Incremental Integration testing

先回顾一下 integration testing 集成测试，也就是测试由多个组件组成的子系统；一般
来讲集成测试我们一般用黑盒测试，但有的时候会导致本地错误，而使用 Incremental
Integration testing 可以减少此类问题

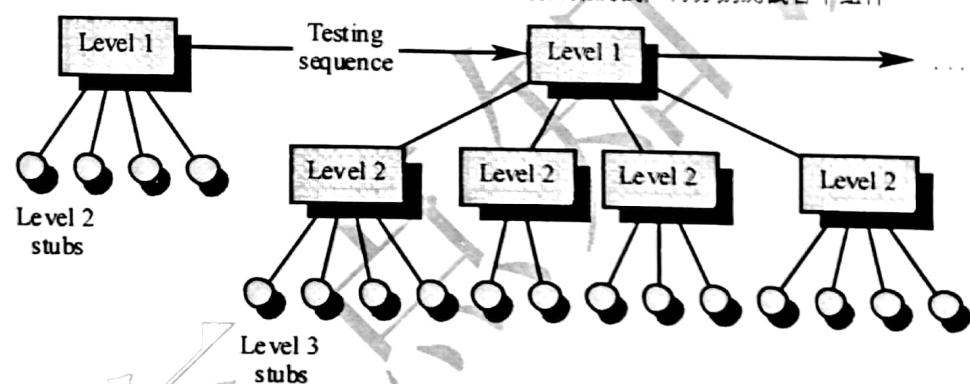




如图所示，在添加新模块的时候，还需要测试以前的 testing case，保证新的 case 不会破坏以前的 case；

第二种 Top-down testing

字面意思，自上而下的测试，从完成系统开始测试，再分别测试各个组件

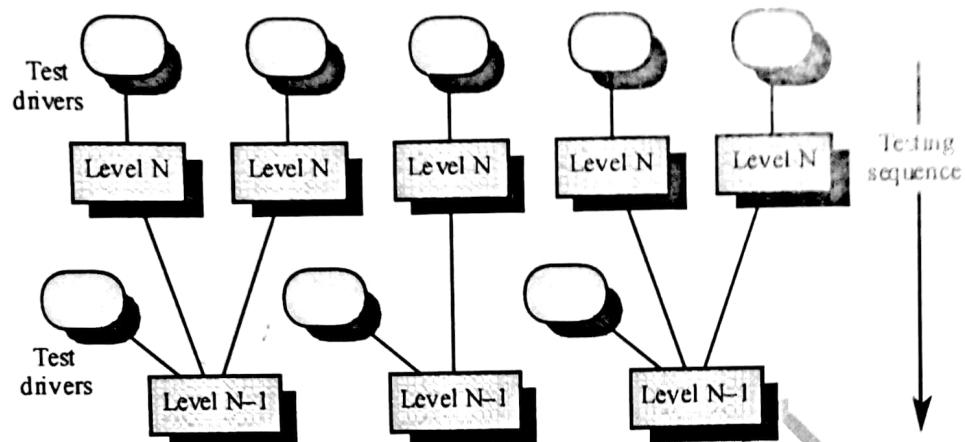


理论上来讲，这种不会常用，周期性太长了

第三种 Bottom-up testing

从底层开始，分别测试组件，直到完成完整的系统





Object-oriented systems	可以对类和方法进行简洁的分解，使测试时变得容易
real-time systems	可以更快的识别代码
systems with strict performance requirements	可以在测试过程的早期测试各个方法的性能，对整个系统的后续开发有好处

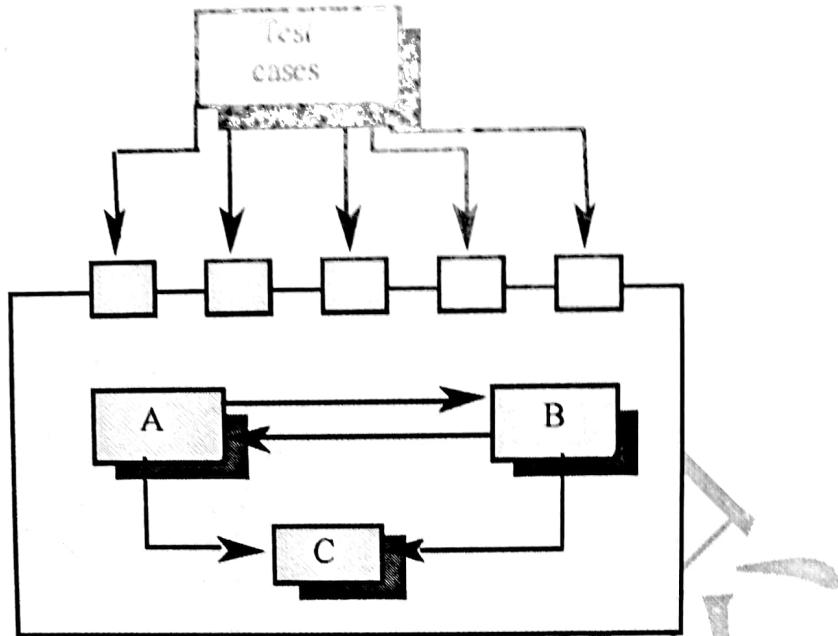
然后我们把 Top-down testing 和 Bottom-up testing 二者对比：最好都记住

情况	采用	原因
Architectural validation	Top-down testing	可以更好的发现架构中的错误
System demonstration	Top-down testing	允许系统再开发的早期就可以有演示，使得系统的后续开发更为顺利
Test implementation	Bottom-up testing	更容易测试
Test observation	二者都存在问题	需要额外的代码及辅助方法来观察测试

第四种 Interface testing

当构建比较庞大的系统的时候使用，目的在于检验不同模块之间的接口所引起的错误





这个图意义表达不太好，画他是怕出来得认识

既然讲到了这里，顺便谈一下 Interface 的种类，和之前讲过的那两种划分方法不一样：

Parameter interfaces

把数据从一个程序传递到另一个程序

Shared memory interfaces

程序之间共享内存块的接口

Procedural interface

其他子系统调用某子系统封装的一组方法

Message passing interfaces

子系统请求来自其他子系统的服务

Interface 可能产生的问题：

Interface misuse

Interface misunderstanding

Timing errors

最后第五种 Stress testing

检验系统设计的最大负荷，有时强行调用会导致某些缺陷；比如当网络过载时分布式系统是否会产生严重降级

Testing 的内容已经描述完了，对于每个图大家都要有了解，知道这是干啥的，需要区分的已经描述过了，其他的可以随意掌握，出题的时候，这里不太会全出概念，所以一旦放到一个实例里就容易被忽视，而且因为太灵活了，所以考法很多，只能靠理解来做，预计出五到十道题，选项会经常涉及。



Part 8 Management of Software Project

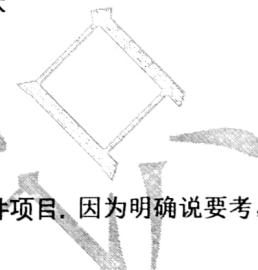
—— 如何管理项目的进度及预算

涉及考点:

1. Project management
2. Cost estimation

主要知识点:

1. 管理的时候怎么规划和安排
2. 管理活动和避免风险
3. Software Cost Estimation, 计算 cost 的办法和评估技术
4. Productivity



Process Management

Project management 这里指的是组织、计划和安排软件项目。因为明确说要考，所以这里把他详细分为四个部分来进行介绍

Management activities	软件项目管理及其特点
Project planning	讨论项目规划及规划过程。
Project scheduling	显示项目管理如何使用图形化进度表示
Risk management	讨论风险的概念和风险管理过程

首先了解 management activities

因为软件开发总是受制于开发软件的组织所设定的预算和进度限制，所以项目管理负责确保软件按时按期交付，并符合开发和采购软件的机构的要求。当然，许多工程项目管理技术同样适用于软件项目管理技术这是因为复杂的工程系统往往遇到与软件系统相同的问题。

因为良好的项目管理是项目成功的关键，但是软件的无形特性给管理带来了问题。虽然管理人员的角色各不相同，但他们最重要的活动是计划、估计和调度。计划和评估是贯穿项目过程的迭代过程

接下来，Project planning，这可能是最耗时的项目管理活动，从最初的概念到系统交付的持续活动。当有新的资料时，计划必须定期修订，可以开发各种不同类型的计划来支持与进度和预算有关的主要软件项目计划

这是一个 project planning 的分类



Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements of the system, maintenance costs and effort required.
Staff development plan.	Describes how the skills and experience of the project team members will be developed.

这里涉及到了活动管理，需要知道两个概念，项目中的活动应组织起来，以产生有形的产出，供管理部门判断进度

其中 milestones 是流程活动的端点，瀑布流程允许直接定义

deliverables 是交付给客户的项目结果

接着，我们看第三点，**project scheduling**，将项目划分为多个任务，并估计完成每个任务所需的时间和资源，同时组织任务，以最佳利用劳动力，最小化任务依赖性，以避免由于一个任务等待另一个任务完成而导致的延迟。

但是实际上，评估问题的难度以及开发解决方案的成本是困难的，工作效率与从事一项任务的人数不成比例，由于沟通费用的原因，在一个较晚的项目中增加人员可能会使它变得更晚再有就是意外，所以在计划时要考虑到偶然性这个奇葩的条件。

这里，还是原来的套路，画图显示进度，引入两个图

- **Activity charts** show task dependencies and the critical path
- **Bar charts** show schedule against calendar time

最后，第四点，**risk management**，我们可以分为四类

风险识别	识别项目、产品和业务风险
风险分析	评估这些风险的可能性和后果
风险计划	制定计划以避免或尽量减少风险的影响
风险监控	在整个项目中监控风险

按照程度划分评估每个风险的可能性和严重性的话

概率可能非常低、低、中等、高或非常高

风险影响可能是灾难性的、严重的、可容忍的或微不足道的



Software Cost Estimation

这一节就是个现实问题，只要能解决以下四个问题就 OK

完成一项活动需要多少努力?

完成一项活动需要多少 calendar time?

一项活动的总成本是多少?

项目估算、进度安排和交叉管理活动

既然说算钱，咱们先看看钱都去哪了

Cost 一般有这么几个出处:

硬件和软件成本

差旅费和培训费

工作成本(大多数项目中的主导因素)

参与该项目的工程师的薪金

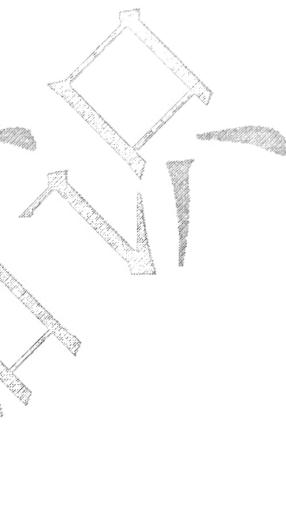
社会保险费用

工作成本必须考虑到管理费用

建筑、供暖、照明费用

网络和通讯费用

共用设施的费用(e.g. 图书馆、员工餐厅等)



知道了以后就想自己推算

这里会用 programmer productivity 来衡量单位时间内生产的有用程度，解释为 A measure (可以是 size-related/ function-related) of the rate at which individual engineers involved in software development produce software and associated documentation

不过其实，推测也是有困难的，不然讲啥了是吧，困难比如说: the size of the measure

/the total number of programmer months which have elapsed/ contractor productivity (e.g. documentation team) and incorporating this estimate in overall estimate, 这里这个 productivity 比较抽象，如果代码语言低级，productive 会产生影响。

而对于 productivity 可以认为，所有基于体积/单位时间的度量标准都是有缺陷的，因为它们没有考虑到质量

特别是生产率的提高通常是以牺牲质量为代价的

目前还不清楚生产力/质量度量是如何相关的

如果更改是常量(例如简化改进代码)，那么基于计算代码行数的方法是没有意义的



然后区分两个点：

Function points

基于程序特征的组合

外部输入和输出，用户交互，外部接口

系统使用的文件权重与每一个都相关

函数点计数是通过将每个原始计数乘以权重并对所有值求和来计算的

Object points

对象点是与函数点相关的另一种度量方法

对象点与对象类不同

程序中目标点的数量是一个加权估计

显示的独立屏幕的数量，系统生成的报告的数量和必须开发的模块的数量

然后了解了估计，我们看看怎么估计。

大致有五种技术：

- **Algorithmic cost modelling**

一种基于历史成本信息的公式化方法，通常基于软件的大小

- **Expert judgement**

一个或多个软件开发和应用领域的专家使用他们的经验来预测软件成本。

过程迭代，直到达成某种共识。

优点：估算方法相对便宜。如果专家有类似系统的直接经验，是否可以准确

缺点：如果没有专家，非常不准确！

- **Estimation by analogy**

项目的成本是通过将项目与相同应用领域的类似项目进行比较来计算的

优点：项目数据准确

缺点：如果没有类似的项目被处理，那是不可能的。需要系统维护成本数据库

- **Parkinson's Law**

这个项目花费了所有可用的资源

优点：没有超支

缺点：系统通常是未完成的

帕金森定律指出，工作的范围会扩大，以填满可用的时间。成本是由可用资源决定的，而不是由客观陈述决定的。

一个例子：这个项目应该在 12 个月内交付，并且有 5 个人可用。力 = 60 p/m



- **Pricing to win**

项目的成本是客户必须在项目上花费的所有成本

优点:你得到了合同

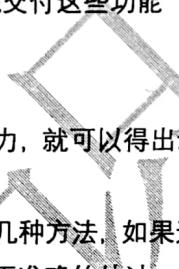
缺点:客户得到他或她想要的系统的可能性很小。成本不能准确反映所需工作

不过,这五个方法是有共同点的,他们都会使用两种估计方法:

这些方法中的任何一种都可以使用自顶向下或自底向上

自顶向下

从系统级别开始,评估整个系统功能,以及如何通过子系统交付这些功能



自底向上

从组件级别开始,估计每个组件所需的工作。再加上这些努力,就可以得出最终的估计

当然了,每种方法都有优点和缺点,所以评估应该基于几种方法,如果这些结果不能返回大致相同的结果,就没有足够的信息可用,为了作出更准确的估计,应该采取一些行动查明更多的情况,不过,price to win有时是唯一可行的方法

还有一种很没意思的方法 Experience based estimate

评估主要是基于经验的,然而,新的方法和技术可能会使基于经验的评估不准确

面向对象而不是面向功能的开发

适合客户机-服务器系统而不是大型机系统,现成组件,基于组件的软件工程,案例工具和程序生成器

然后就是一个计算 cost 的算法

成本是产品、项目和过程属性的数学函数,项目经理对这些属性的值进行估算

$$\text{Effort} = A * \text{Size}^B * M$$

A 是一个依赖于组织的常数, B 反映了大型项目的不成比例的努力, M 是反映产品、过程和人员属性的乘数

成本估算最常用的产品属性是代码大小

大多数模型基本相似,但是 A、B 和 M 的值不同

最后,软件系统的大小只有在完成时才能被准确地知道,有几个因素影响最终的大小:
使用 COTS 和组件; 编程语言; 分配系统

随着开发过程的进展,大小估计变得更加准确



资料到这里结束啦，如果有什么问题欢迎大家
在群里或者在公众号提问~如果有任何疑问也可以来
问我们，也最后祝大家期末都有一个好成绩!!!

