1: The Nature of Software

1. Software

   Definition of software：软件是指令、数据结构和文档描述信息（和给用户提供的服务）的合集；是一种用计算机语言表达的逻辑，是可以在计算机硬件上执行的指令的集合

   Characteristics of Software：软件是经过设计的，而不是制造的；软件不会磨损，会恶化；大多数软件是定制的。

   The difference of software and hardware：软件是开发的，而硬件是制造的；软件不会磨损，而是退化，硬件会磨损（这是本质区别）；软件是定制的，可复用化程度低，硬件复用化程度高；软件不存在备用部件，可替换性低，硬件存在备用部件

2. The changing nature of software

   1) 软件需要适应新的计算环境或者技术的需求
   2) 软件必须增强以实现新的业务需求；
   3) 软件必须拓展，使其能与其他系统交互合作；
   4) 软件架构必须改建，使之能适应多样化的网络环境

2: Software Engineering

1. Software engineering – a layered technology:

   The definition of Software engineering：运用系统的、规范的、可量化的方法，开发、运行和维护软件，将工程化的方法运用到软件开发中；软件工程研究软件开发、运行和维护的工程化过程、方法和工具；建立在计算机科学和数学的基础上

   The goal of Software engineering

   Layer: tools, methods, process and a quality focus：以质量为关注点，自下而上依次是：过程（基础），方法（核心），工具（关键）

2. A process framework

   The generic five process activities: communication, planning, modeling, construction and deployment 沟通，策划，建模，构建，部署

   沟通：与客户、团队成员和其他利益相关者深入沟通交流，详细理解项目需求，减少误解的风险

   策划：制定项目计划书，确定项目的目标和计划，项目估算，时间安排等

   建模：**制定项目体系框架，帮助团队和客户更好地理解系统的结构和功能**

   构建：将设计的模型转化为实际系统的组件，进行代码的编写和测试，保证正常运行

   部署：将构建好的系统或系统的部分交付给用户使用，用户测评，得到反馈

3. Software development myths：软件及其开发过程中被人误解的说法

4.Umbrella activities

   软件项目跟踪与控制；风险管理；软件质量保证；技术评审；度量；供应链管理；可重用性管理；工作产品准备与生产

3: Software Process Structure

1. Prescriptive models

The function of process models：过程模型用于确定软件开发和演化的顺序，建立一个阶段到下一个阶段的标准，不同的过程模型在于执行软件过程框架的五个阶段的顺序不同

Understand the signification and characteristics of the process models

Pattern & Framework

Pattern 模式：是解决设计问题的解决方案或思路和模板，用于指导开发，是抽象层面的方法论

Framework 框架：是可复用的架构和代码合集，是半成品，用于加速软件开发，是具象具体的实现

2. The waterfall model 线性方式完成，按照五个步骤的顺序完成

特点：线性不可逆，大量文档驱动，固定阶段划分，阶段评审机制，风险高

适用：需求明确清晰、开发过程中变更较少的情况，或者熟悉的系统；不适用于快速交付，客户无法说明确定需求或者频繁修改需求，高度依赖创新和快速迭代的项目

优点：需求明确且固定，早期就可以清楚地了解软件的需求和目标；线性方法模型简单；开发软件高度规范化，标准化，后期维护少

缺点：线性工作方式，不可逆，需要一次性弄清所有问题，有不确定性；得到成品的时间晚，有一定的风险

V cycle model：由瀑布模型改进得来的一个变体，将早期的软件开发活动与后期的验证和确认活动关联起来。（本质上是加了一系列测试，质量保证）

通用框架模型是如何运用到 Waterfull Model 螺旋模型上的？

沟通：与客户、团队和其他利益相关者沟通，了解需求，减少误解

策划：根据需求，完成软件项目计划书，制定项目的目标和计划

建模：制定系统的体系框架，形成开发方案（需求建模和设计建模），帮助团队和客户理解需求

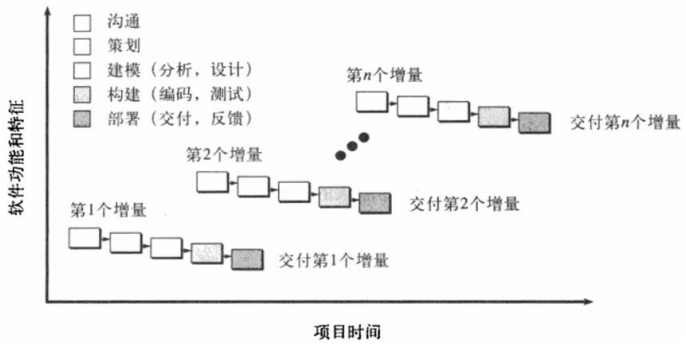
构建：将设计模型转化为实际组件，进行编码测试，保证正确

部署：将构建的系统交付给用户，得到反馈。

瀑布模型是线性模型，按照五个步骤的先后顺序完成，模型过程不可回退

3. Incremental process models（阶段式提交）增量过程模型

增量模型将一个大型复杂软件分解成若干增量，每一个增量都是一个可以运行的软件版本，每个部分都建立在已经交付的部分的基础上



适合需求清楚、周期比较短的项目

适用：可以分批次交付、系统可模块化的软件，或者难以一次性开发

OO-based

Why use incremental model?

优点：模块化、组件化，开发顺序灵活，一些情况下可以并行开发。每一次迭代得到的增量都是一个可运行的版本，提高了软件开发过程可见性；规避风险，一个开发周期的错误不会影响到整个软件系统；可以按照组件优先级实现，逐渐扩展，有效适应用户需求的变更；

缺点：要关注每一个增量的交付，早期的增量很容易被忽略

通用框架模型是如何运用到 Incremental Model 螺旋模型上的？

沟通：**在每一个增量的开始**，与客户、团队和其他利益相关者沟通，**确认当前增量要实现的功能和优先级**，了解需求，减少误解

策划：根据初步需求，制定**当前增量开发**的目标和计划

建模：制定**当前增量计划**的体系框架，进行分析，形成开发方案（需求建模和设计建模），确保体系结构的可扩展性和一致性

构建：**开发当前增量**，进行编码测试，**保证增量正确性和对已有系统的适应性**

部署：**将该增量集成到已有系统**，交付给用户，得到反馈。**为下一轮增量的开发提供依据**

4. Evolutionary process

需求不明确的软件开发，引入软件演化的思想。演化模型是迭代的过程模型

演化过程模型分为原型开发和螺旋开发。

Prototyping Model 原型模型

瀑布模型的改进，适合需求不清楚的系统

思想：在限定的时间内，快速开发出一个可实际运行的系统模型，用户在运行使用整个原型的基础上，通过对其评价，提出改进意见，对原型进行修改，统一使用，评价过程反复进行，原型逐步完善，直到满足用户的需求为止

简化软件的实现，只实现部分功能，快速生成软件样品，这个样品就是软件原型，然后基于原型与用户进行沟通，确定修改意见，再对原型进行修改和增加功能。经过多次迭代最终生成满足用户需求的完整软件。

适用：用户需求模糊、经常发生变化；系统规模不太大，不太复杂

优点：快速构建、构建方便，容易修改，更容易满足用户预期；增加了用户和开发人员的互动，用户在项目中起到主导作用，满足用户的动态需求，降低开发风险

缺点：用户的参与使得其不适合大型、复杂项目开发。构建的第一个系统几乎无法使用；客户不切实际的期望；性能问题

特点：旨在帮助客户或开发人员了解需求；快速设计（专注于用户可见的软件方面的表示）；原型由客户评估，并用于识别和完善需求；（部分）原型被丢弃

通用框架模型是如何运用到 Prototyping Model 螺旋模型上的？

沟通：**在原型开发的初始阶段**，与客户、团队和其他利益相关者沟通，收集用户初步（不完整的）需求，减少误解

策划：根据初步需求，制定**原型开发**的目标和计划

建模：制定**设计原型**的体系框架，进行系统分析，形成开发方案（需求建模和设计建模）

构建：**快速实现原型系统**，在短时间内进行编码测试，**重点是快速可见可体验**

部署：**将原型展示给用户，收集反馈，根据用户要求修改**原型，之后反复进行沟通、策划、建模、构建和部署的迭代。

Process pattern

Spiral Model （风险分析）

螺旋模型是一个风险驱动的过程模型，它结合了原型模型和增量模型的特点

每一次迭代过程中增加了**风险控制机制**，强调风险分析。

螺旋模型是开发大型软件系统的理想过程，常用来指导大型软件项目的开发周期性的方法来进行系统开发，开发出很多版本。以进化的开发方式为中心，在每个项目阶段使用瀑布模型法。这种模型的每一个周期都包括需求定义、风险分析、工程实现和评审 4 个阶段，由这 4 个阶段进行迭代。

优势：降低风险；不区分开发、维护，将开发和维护活动融入到每个迭代周期中；确定一些列里程碑作为支撑点，确保相关利益者满意

缺点：仅适用于大型软件；仅适用于内部（内部）软件；依赖大量风险评估

通用框架模型是如何运用到 spiral model 螺旋模型上的？

沟通：每轮螺旋起始都需要与客户、团队和其他利益相关者沟通，理解需求，减少误解

策划：**设定本轮目标、时间进度，并进行详细的风险评估，这是核心**

建模：制定项目体系框架，进行系统分析，形成开发方案（需求建模和设计建模）

构建：将设计模型转化为组件，进行编码和测试，构建原型或系统组件

部署：将已构建的系统或系统部分交付用户使用，收集反馈，作为下一个螺旋的起点。

在螺旋模型中，每个迭代的过程都有风险控制机制，直到最终完成交付。

5. Specialized process models

Component based development（需要面向对象技术支持）

Object-oriented process models

6. Unified process model(5 个阶段)

Inception（起始）, Elaboration（细化）, Construction（构建）, Transition（转换），Production（生产）

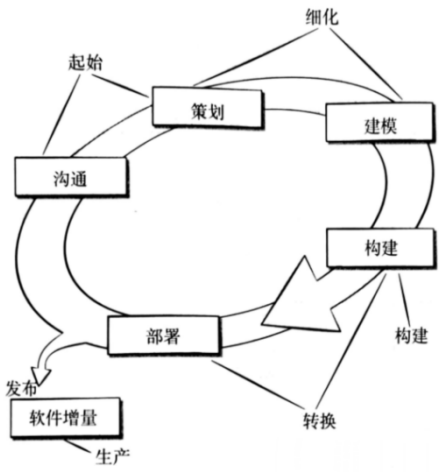


图 2-6　统一过程

面向对象、以用例为驱动、以架构为核心的迭代式软件开发过程模型

Agile Development

1. What is Agility?

敏捷开发强调可运行软件的快速交付而不看重中间产品

敏捷开发最具强制性的特点是：能够通过软件过程来降低由变更需求所引起的代价　快速、增量（迭代）

敏捷开发的目标或追求：开发人员和客户之间的主动持续沟通；让客户满意；尽早增量发布；小而高度自治的开发团队；非正式的方法；最小化的工作；整体精简开发的原则

敏捷开发的原则：尽早且频繁地交付有价值的软件，满足客户需求；积极接受需求变更；在整个开发期间，负责人、开发人员、用户一起工作，紧密合作，面对面沟通；可工作的软件是进度的标准；减少不必要的工作（根本原则），实现可持续开发；自组织团队，以受激励的个体为核心；可工作的软件是衡量进度的标准；良好设计，提高软件品质；定期反思和改进

2. Agile Process

XP　(pair programming 结对编程)

是敏捷开发中应用最广泛、最富有成效的一种开发过程；是一个轻量级的、可适应的、严谨周密的开发过程；适用于小团队

极限编程分为四个阶段：策划、设计、编码、测试

策划：听用户故事，客户表达故事的价值，规划迭代计划

设计：保持 KIS 原则（保持简洁），遇到困难采取重构原则

编码：先进行单元测试，进行结对编程

测试：进行系统的集成和测试

Scrum

4: Understanding Requirements

1. A bridge to design and construction

The definition of requirements engineering：帮助软件工程更好地理解要解决的问题，是设计和实施之间的桥梁

2. seven Requirements engineering tasks

初启 (Inception)；导出 (Elicitation)；细化 (Elaboration)；谈判 (Negotiation)；规格说明 (Specification)；确认 (Validation)；需求管理 (Requirements Management)

3. Initialing the requirements engineering process

4. Eliciting requirements

通过开发系统原型获取用户需求

5. Developing user-case

5: Requirements Modeling

1. Requirements analysis

The three goals of analysis modeling（Information/Data, Function, Behavioral)

分析建模的三大核心目标：信息建模（描述系统需要处理和保存的实体及其关系），功能建模（描述系统提供哪些功能以满足用户需求），行为建模（描述系统在特定情境下的反应和状态变化）

The concepts of analysis modeling：通过结构化模型来表达系统应该做什么，而不是怎么做

Specification and Requirements：

需求：用户对系统的描述，是用户希望实现的功能，说明软件必须做什么，是需求分析的输入，关注的是做什么

规格说明：是开发者对系统的描述，是技术要求清单，是需求分析的输出。关注的是如何做

Customer and End-User

客户：支付或购买软件的人，对软件提出需求的人，通常是决策者

用户：最终使用软件的人，是软件的使用者。两者可能位同一人也可能不是

2. Analysis modeling approaches

   The principles of modeling

3. Data modeling concepts

   E-R diagram, relationship of objects

4. Scenario-based modeling

   UML

   Use-Cases in UML：use-case diagram/ activity diagram/ sequence diagram/state diagram/class diagram

   OO analysis: Behavioral, Class, Use-Case

5. Creating a behavioral model

6.Class-based modeling

   Identifying analysis classes

   CRC(class-responsibility-collaborator) Modeling

6: Design Concepts

1. Design within the context of software engineering

   Map the analysis model into design model

2. Design concepts

   abstraction 抽象, Refinement 求精, architecture 架构, patterns 模式, modularity 模块化, information hiding 信息隐藏, functional independence,功能独立 refactoring 重构, design class 设计类

3. The design model

   the concepts of the design process. 设计模型是从分析模型过渡到实现模型的桥梁

   four design models：

   Data Design: 计将分析过程中创建的数据模型转换为实现软件所使用的数据结构、数据库模式和访问策略。

   Architectural Design: 定义了软件的整体结构和各个模块之间的关系（组件如何组合在一起）。// 决定系统的可拓展性，可维护性和稳定性

   Interface Design: 描述了软件内部模块之间，软件与外部系统之间，软件与用户之间的交互方式。// 确保用户操作顺畅性和系统集成性

   Component-Level Design: 系统架构确定之后，将模块进一步细化为具体可实现的组件（可编码的功能单元）（如处理逻辑，接口等）

4 characteristics of a well-formed design class：

Complete and sufficient（完整且足够）：一个类应该包含完成自身职责所需的所有属性和方法，但不多也不少

Primitiveness（原始性）：类的操作应该只执行单一、明确、不可再简化的功能，不应隐藏复杂操作或混合职责

High cohesion, Low coupling（高类聚低耦合）：类中所有属性和方法应围绕同一个功能主题展开，彼此紧密相关；类之间的依赖关系应尽可能松散，变动一个类不应频繁影响其他类

Analysis Model and Design Model（二者关系：过程维度、抽象维度）

分析模型：描述系统“做什么”，关注需求和业务逻辑

设计模型：描述系统“怎么做”，关注结构和实现方案

设计模型是分析模型的实现深化，分析模型表达概念性问题本质

5. Quality attributes

功能性，易用性，可靠性，性能，可维护性

7: Architectural Design

1. Software Architecture

The definition of architectural

软件结构定义了软件的整体结构和各个模块之间的关系（组件如何组合在一起）。决定系统的可拓展性，可维护性和稳定性。// 考虑架构风格，组件架构，组件属性，各个组件之间的关系。

2. Data design

The goal of Data Design in the Architectural Design

将需求分析阶段建立的数据模型，转化为系统内部可使用的数据结构、数据库模式和数据访问策略

3. Architectural styles and patterns ***what the fuck is that?***

components, connectors, constraints, semantic （语义）models;

Data-centered, Data-flow, Call and return, Object-oriented, Layered architectures？

Architectural complexity: dependencies（三种依赖关系）？

what the fuck is that?

8: Component-level Design

1. What is a component

OO view：面向对象开发中，组件由多个协作类构成，分析建模与设计建模是迭代的，分析类的完善需要进一步的设计建模来完成。

Conventional view（传统观点）：组件是程序功能的元素（包括处理逻辑，数据结构和接口），组件具有可重用性，软件是使用现有组件构成的

2. Design Class-based component

Basic design principles **四个基本设计原则**

① 开闭原则 OCP：模块应该为拓展开放，对修改关闭（也就是说，模块的行为可以得到拓展，当新需求出现时要通过拓展满足新需求，但是不能改动模块的源代码）

一个 ShapeService 类，该类负责计算不同形状的面积。目前，它只能处理矩形和圆形。假设现在需要添加一个新的形状，比如三角形，那么我们必须修改 ShapeService

类，添加一个新的方法来处理三角形。假设现在需要添加一个新的形状，比如三角形，那么我们必须修改 ShapeService 类，添加一个新的方法来处理三角形

```
// 原始版本：违反开闭原则
class Shape {
    public int type;
}

class Rectangle extends Shape {
    public int width;
    public int height;
}

class Circle extends Shape {
    public int radius;
}

class GraphicEditor {
    public void drawShape(Shape s) {
        if (s.type == 1) {
            // draw rectangle
        } else if (s.type == 2) {
            // draw circle
        }
    }
}
```

```
abstract class Shape {
    public abstract void draw();
}

class Rectangle extends Shape {
    public void draw() {
        System.out.println("Draw Rectangle'
    }
}

class Circle extends Shape {
    public void draw() {
        System.out.println("Draw Circle");
    }
}

class GraphicEditor {
    public void drawShape(Shape s) {
        s.draw();
    }
}
```

如果添加新图形（比如 Triangle），必须修改 GraphicEditor 的 drawShape 方法；新增图形类只需继承 Shape 并实现 draw 方法，GraphicEditor 类无须修改，符合开闭原则。

② Liskov 替换原则 LSP：子类对象能够替换父类对象，而且不会影响程序的正确性。也就是说，子类应该能够继承父类的方法和行为，并在此基础上添加新的行为，而不是改变父类原有的行为

父类会的子类必须会（子类可以有更多的功能，但是不能违背父类的功能）

比如说，定义了一个类，鸟，方法是飞行，但是企鹅也是鸟，企鹅继承了鸟的方法，但是企鹅并不会飞，这就是违背了里氏替换原则

```
class Bird {
    public void fly() {
        System.out.println("Bird flies");
    }
}

class Ostrich extends Bird {
    public void fly() {
        throw new UnsupportedOperationException("Ostrich can't fly");
    }
}
```

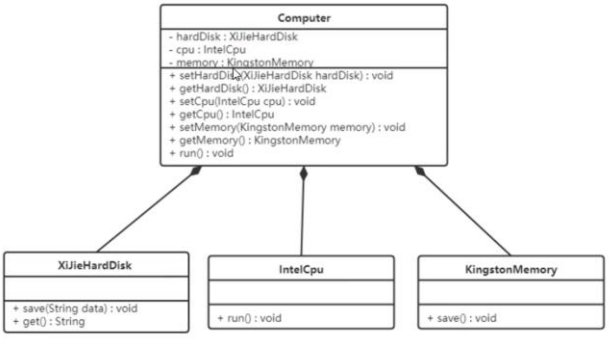
```
interface Bird {
    void eat();
}

interface Flyable {
    void fly();
}

class Sparrow implements Bird, Flyable {
    public void eat() {}
    public void fly() {
        System.out.println("Sparrow flies");
    }
}

class Ostrich implements Bird {
    public void eat() {}
}
```

Ostrich 是鸟类，但不会飞，替换 Bird 时会导致程序异常，违背了 LSP

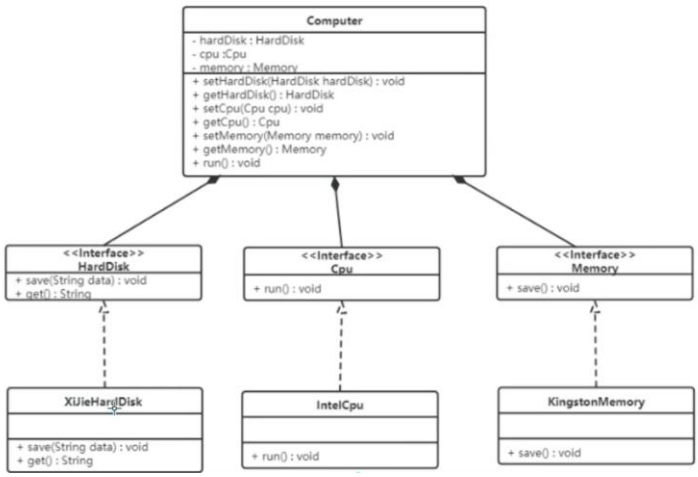
将“飞行”从“鸟”的共性中拆分出来，只有会飞的鸟实现 Flyable

③ 依赖倒置原则 DIP：
通过接口或抽象类来进行编程，而不是直接依赖于具体的实现类
高层模块（业务逻辑）不应依赖于低层模块（具体实现），二者都应依赖于抽象（接口或抽象类）
应该是具体实现依赖于逻辑

现要组装一台电脑，需要配件 cpu，硬盘，内存条。只有这些配置都有了，计算机才能正常的运行。选择 cpu 有很多选择，如 Intel，AMD 等，硬盘可以选择希捷，西数等，内存条可以选择金士顿，海盗船等。



不要依赖于具体，要依赖于抽象



④ 接口分离原则 ISP：
将不同的功能定义在不同的接口中，避免其他类在依赖该接口时依赖其不需要的方法。这意味着接口应该被细分为更小的、更具体的接口，以减少接口之间的依赖冗余性和复杂性

```
interface Worker {
    void work();
    void eat();
}

class Robot implements Worker {
    public void work() {}
    public void eat() {} // 机器人不需要吃饭
}
```

```
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class Human implements Workable, Eatable {
    public void work() {}
    public void eat() {}
}

class Robot implements Workable {
    public void work() {}
}
```

Two qualitative criteria for measuring module independence: Cohesion & Coupling

耦合：描述不同组件之间相互依赖的程度，耦合越低各个模块之间的联系越少，越独立，越容易维护

内聚：描述组件内部的各个功能的相关程度，内聚程度越高，说明模块内部功能越聚合，模块责任越统一；理想的软件设计是：高内聚，低耦合

Analysis Class and Design Class

3. Conducting component-level design

The steps of OO Component-level Design

1. 确定问题域中的设计类
2. 确定基础架构中的设计类
3. 设计类，描述数据结构，设置恰当的接口，设计属性，描述处理流程
4. 描述持久性数据源
5. 设计行为表征
6. 设计部署图，部署
7. 重新考虑，重新设计，替代方案

4. Design conventional components 设计传统组件

flow diagram 流程图

9: User Interface Design

1. The Golden Rules

1. 让用户掌控控制权：尊重用户意图，不强迫用户执行不必要的操作；为用户提供灵活的交互，允许用户中断和撤销交互；简化交互并允许用户自定义个性化交互，隐藏复杂的内部技术信息

2. 减轻用户记忆负担：减少用户短期记忆的负担，用户不需要记住大量信息就能完成操作；界面提供有意义的默认值，提供直观可见的快捷方式；使用默认值、图形隐喻、分步渐进展示等方式。

3. 保持界面一致性：让用户理解当前任务所在的上下文；同一系列中保持风格和操作统一；不轻易更改已有的交互模型，除非有明确的改进理由

2. User Interface analysis and design

user analysis, task and work environment analysis

Interface design

implementation

Interface validation

3. Interface analysis

Steps of interface analysis

4. Interface design Steps

Design GUI according to Use-Case Diagram

10: Testing strategies and techniques

1. A strategic approach to software testing ※

Verification and validation（验证与确认）

验证：验证软件是否实现了特定的功能，是否实现了正确的设计，做的是“我们的产品正确吗”

确认：确认软件是否满足用户的事情，做的是“我们是否开发了正确的产品”
也就是说，验证是“把事情做正确”，而确认是“做了正确的事情”

2. Test strategies for conventional software（过程与文档）

Unit testing 单元测试，验证单个程序模块（接口，局部数据结构，边界条件）的正确性，通常由开发者完成

integration testing 集成测试，验证多个模块之间的交互是否正确，关注模块间协作是否异常，由开发人员或者测试人员完成

Top-Down integration & Botton-Up integration

自顶向下集成测试从高层模块开始，逐步向下集成测试底层模块，适用于验证整体结构和控制流程。

自底向上集成测试从最底层的模块开始，逐步向上构建系统，适合先验证底层逻辑和数据处理的正确性。

regression（回归）testing & smoke（冒烟）testing

回归测试是在修改代码后，重新运行已有测试用例，确保原有功能没有被破坏。

冒烟测试是一种快速、初步的测试方法，用于确认软件构建是否基本稳定，主要功能是否能正常运行。

Acceptance testing（Validation testing）验证测试/验收测试

确认软件是否满足用户需求和业务需求，分为确认测试和系统测试

3. Validation testing 确认测试：是否满足用户的需求

4. System testing 系统测试：由独立的测试团队在完整的系统环境中测试整个软件系统，是对整个系统是否“按规格说明”的测试

Use-Case Diagram

Function testing, specification // specification 为规格说明的意思

5. The art of debugging

Failure 失败（外错误）是指系统偏离其所需的行为。

Fault（故障、内错误、错误、bug）由于某些因导致某些软件产品中人为错误

The relationship of testing and debugging

测试是发现错误的过程，调试是定位并修复错误的过程。

6. White-Box testing

Flow Graph Notation **详见测试专题大题 必考※**

Cyclomatic Complexity（环复杂度与独立路径）

白盒测试和黑盒测试之间的差别

白盒测试：从程序开发者和程序内部逻辑的角度进行测试，了解内部结构和算法，关注程序内的代码逻辑和路径是否正确实现

黑盒测试：从用户和外部功能角度进行测试，关注程序的输入输出是否符合要求

7. Basis path testing

8. Control structure testing（条件/循环）

9. Black-Box testing

Equivalence Partitioning（等价类划分）等价类划分通过将输入数据分为有效和无效的等价类，从每类中选代表值进行测试

Boundary Value Analysis（边界值分析）边界值分析关注输入的边界条件

10. OO Testing Methods

面向对象测试方法是针对类、对象、继承和多态等特性进行测试，确保对象之间的行为与协作符合设计要求。