

## Software Quality – 软件质量

- Software quality and how to get to it
- Test-driven development
  1. White box testing
  2. Black box testing

### Why is Software Quality relevant? - I

**Reputation**声誉: Software developers and their organisations **rely on reputation**. A **poor quality product** (or family of products) can be **enormously damaging for business**. 低劣产品带来巨大损失 Software bugs can have immediate impacts on custom, especially in **customer-facing industries**. The automotive software problems with Volkswagen 大众汽车软件问题 have led to an enormous amount of negative publicity, which has a direct impact on sales.

**Limiting Costs of Product and Maintenance Cost**产品和维护成本限制: is an overriding factor 主要因素 in software development. **Poor quality** software tends to be **expensive to develop and to maintain**, which can have a detrimental effect on business. **Poor software** quality can lead to **technical debt**, where the organisation in charge of the software needs to invest a disproportionate amount of resources into maintaining and running the software to make up for (and to try and remedy) **poor design and implementation decisions**. 质量低劣的软件往往开发和维护成本高昂 + 可能导致技术负债

**Software Certification**软件认证: Depending on the domain of the software (e.g. in Aircraft or Rail), the development and use of software **might be restricted, and dependent on obtaining some form of certification**. For example, software in modern civilian aircraft often has to be certified to the **DO178 standard**, which requires the extensive use of software quality assurance procedures throughout the software development lifecycle.

软件认证原因: 要求在整个软件开发生命周期中广泛使用软件质量保证程序

## Why is Software Quality relevant? – II

**Organisational Certification组织认证:** The organisational procedures and structures that are employed for software development can **have a huge bearing** on the **quality of the software they produce**. There are **various ways** by which to categorise the extent to which an organisation employs good practice. International certification procedures and standards such as **CMM and ISO90017** exist, so that software development organisations can **advertise their “capability”** to develop **high quality software**. 国际认证程序和标准，如CMM和ISO90017，存在以便软件开发组织能够宣传他们开发高质量软件的“能力”。

**Legality合法性:** Depending on the country, there may be **overriding legal obligations** that apply to organisations that use software. For example, **in the UK**, organisations have to demonstrate that the risk posed by their technology (this includes software) is **“As Low As Reasonably Practicable” or “ALARP”**. In other words, every “practicable” measure must have been taken to **demonstrate** that (in our case) the software system does **not pose a risk to its users**. 国家不同，可能会有**重大法律义务**。例如，在英国，组织必须证明其技术（包括软件）所带来的风险是“尽可能低”，或者说是“ALARP”。

**Moral / ethical codes of practice道德/职业道德准则:** Even in cases where a software system is not covered by industrial certification and legislation, and where its failure is not necessarily business or safety-critical, there can **remain a moral obligation to the users**. **道德义务** Professional organisations such as the **American Computer Society (ACM)** have explicit ethical guidelines and codes of practice, with statements such as **“Software engineers shall act consistently with the public interest”**. 软件工程师应与公众利益一致行事 This clearly implies that they ought to **do whatever possible to maximise the quality of** their software and to **prevent it from containing potentially harmful bugs**. 尽一切可能提高软件质量，防止其包含可能有害的错误。

## Software Quality is Multi-dimensional-软件质量是多维的

- Subjective or “fitness for use”: **as perceived by an individual user** (e.g., aesthetics of GUI, missing functionality...)主观或“适用性”: 由个体用户感知的软件质量 (例如, GUI的美观性, 缺失的功能等)

Fitness for use: Joseph Juran embodied the idea that the quality of product revolves around its fitness for use . He argued that, ultimately, the value of a product depends on the customer’s needs. Crucially, it forces the product developers to **focus on those aspects of the product that are especially crucial** (the *vital few* objectives) as opposed to the *useful many*.

适用性: 约瑟夫·朱兰体现了产品质量围绕其适用性的理念。他认为, 最终, 产品的价值取决于客户的需求。至关重要是, 这迫使产品开发人员专注于那些特别关键的产品方面 (关键目标), 而不是有用的许多方面。

- Objective or “conformance to requirements”: can be **measured as a property** of the product (e.g., detailed documentation, number of bugs, compliance with regulations .... )客观或“符合要求”: 可以作为产品的属性来衡量 (例如, 详细的文档, 错误数量, 符合法规等)

Conformance to Requirements: **Phil Crosby** embodied a different tone. He **defined quality as “conformance to requirements”** . His opinion was that quality can be achieved by the disciplined specification of these requirements, by setting goals, educating employees about the goals, and planning the product in such a way that defects would be avoided.

符合要求: 菲尔·克罗斯比表达了一种不同的声音。他将质量定义为“符合要求”。他的观点是, 通过严格规范这些要求, 设定目标, 教育员工了解目标, 并以避免缺陷的方式规划产品, 可以实现质量。

# Quality Models: ISO/IES25010

- **Functional Suitability**

- Functional Completeness
- Functional Correctness
- Functional Appropriateness

- **Performance Efficiency**

- Time Behaviour
- Resource Utilisation
- Capacity

- **Compatibility**

- Co-existence
- Interoperability

- **Usability**

- Appropriateness
- Realisability
- Learnability
- Operability
- User Error Protection
- User Interface Aesthetics
- Accessibility

- **Reliability**

- Maturity
- Availability
- Fault Tolerance
- Recoverability

- **Security**

- Confidentiality
- Integrity
- Non-repudiation
- Authenticity
- Accountability

- **Maintainability**

- Modularity
- Reusability
- Analysability
- Modifiability
- Testability

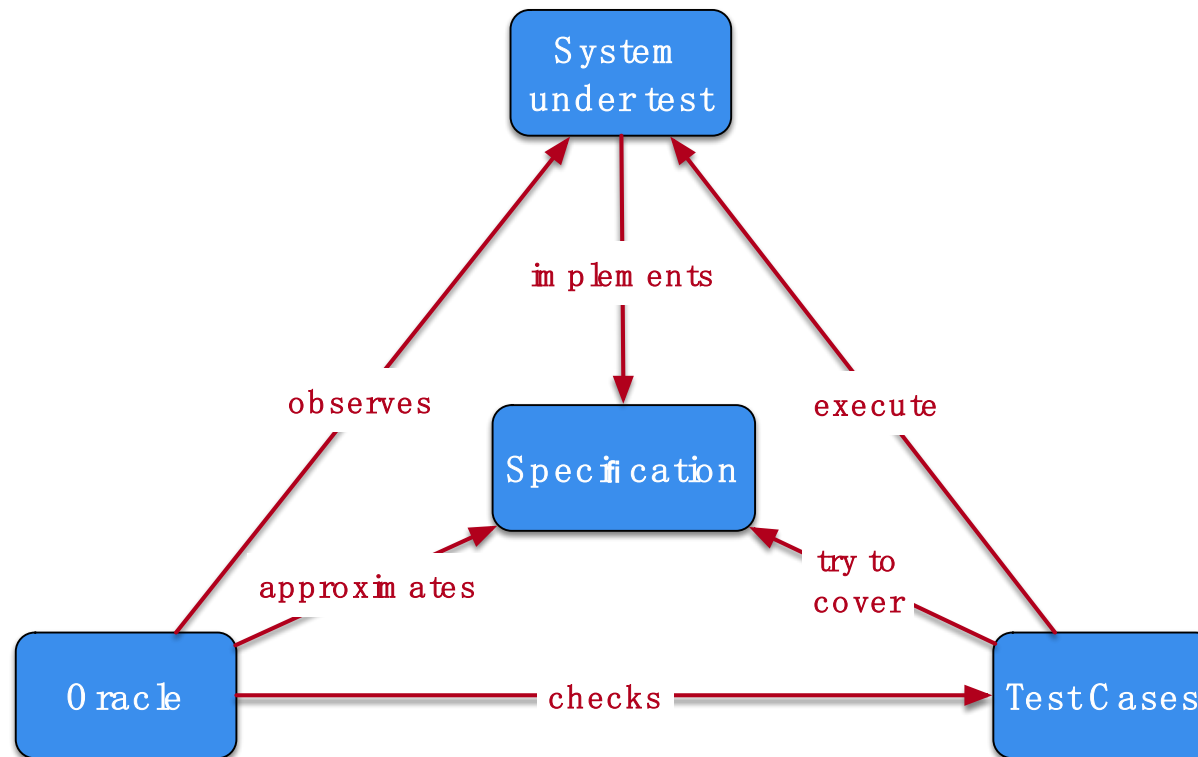
- **Portability**

- Adaptability
- Installability
- Replaceability

# Steps Towards Software Quality - 提高软件质量

- Use a **standard** development **process**
- Use a **coding** standard
  - Compliance with **industry standards** (e.g., ISO, Safety, etc.)
  - Consistent code **quality** 代码质量
  - **Secure** from start 一开始安全
  - **Reduce** development **costs** and **accelerate** time to **market** 降低开发成本并加快上市时间
- **Define and monitor metrics** (defect metrics and complexity metrics) 定义和监控指标 (缺陷指标和复杂度指标)
  - High complexity leads to higher number of defects 高复杂度会导致更多的缺陷
- Identify and remove defects 辨识并清除缺陷
  - Conduct manual reviews 手动审查
  - Use Testing 测试

# Testing process: key elements and relationships



From: M. Staats, M. W. Whalen, and M. P. E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 391–400. IEEE, 2011.

**System under Test (SUT)** This is the system (or unit/function) **being tested**. It seeks **to implement the specification**. The SUT can either be a **white-box system**, where we have complete access to the source code and the run-time state (e.g. the call-stack), or a **black-box system**, where we only have access to the external interface or API (depending on the type of system). It can also be **a mixture of the two**; for example, library routines might be provided in the form of closed source components, whilst the source code for the main core of the system is available for analysis. **The system might be reactive** where the input / output **behaviour at one stage is affected by previous inputs** (e.g. a GUI), or it **might process inputs in a single batch and return to its initial state**. **This matters from a testing perspective, because in the reactive case, the test inputs have to be formulated as sequences**. The system might be *deterministic*, where it always returns the same answer for a given input. It might however also be *non-deterministic*, where the same input can elicit different outputs (perhaps because of randomised internal behaviour, or other factors beyond control such as thread-scheduling). 通常很重要确保SUT是“生产”系统的一个孤立版本。  
**It is commonly important to ensure that the SUT is an isolated version of the “live” system.**

**Specification** A specification represents **the idealised behaviour of the system under test**. Depending on the development context, this might be embodied as a **comprehensive, rigorously maintained document** (e.g. a set of UML diagrams or a Z specification). Alternatively, if developed in an agile context, it might be a partial intuitive description captured in a selection of user stories, test cases, and documented as comments in the source code. The nature of the specification has an obvious bearing on testing. **If a concrete, reliable specification document exists** and there is a shared understanding of what the system is supposed to do, this can be **used as the basis** for a systematic test-generation process. If this **is not** the case, then testing becomes a **more ad-hoc** and **dependent upon** the intuition and **experience** of the tester.

**Test cases** The test cases correspond to the **executions of the SUT**. In practical terms a test case corresponds to an input (or a sequence of inputs) to the system.

Test cases should ideally cumulatively execute every distinctive facet of software behaviour. **An ideal test set (collection of test cases) should be capable of exposing any deviation that the SUT makes from the specification.** If it can be shown to do this, the test set is deemed to be *adequate*. 一个理想的测试集（测试用例的集合）应该能够暴露出SUT与规范的任何偏差。如果可以证明它能够做到这一点，那么测试集被认为是足够的。

**Test Oracle** Executing the test cases alone will not determine whether the SUT conforms to the specification or not. This decision – **whether or not the output of a test is correct or not – is made by a test oracle.** In practice, an oracle might be an assertion in the source code that is checked during the test execution, or it might be the human user, **deciding whether or not the behaviour is acceptable.**

**Test oracles** are notoriously **difficult to produce**. There is in practice rarely an explicit, comprehensive, up to date specification that can be used as a reference. A successful software has usually been developed over the course of decades by a multitude of developers, which means that, **ultimately, there is rarely a definitive record of how exactly the system should behave.** What's more, there may be tens of thousands of test cases, each of which might produces complex outputs, which can make the task of manual validation of the outputs prohibitively **time consuming**. These issues are collectively referred to as the **oracle problem**. 预期问题



**被测系统 (SUT)** 这是正在测试的系统（或单元/功能）。它旨在**实现规范**。SUT可以是**白盒**系统，其中我们完全可以访问源代码和运行时状态（例如调用堆栈），也可以是**黑盒**系统，其中我们只能访问外部接口或API（取决于系统的类型）。它也可能是**两者的混合**；例如，库例程可能以闭源组件的形式提供，而系统的主要核心源代码则可供分析使用。系统可能是反应性的，其中一阶段的输入/输出行为受到先前输入的影响（例如GUI），也可能是一次性处理输入并返回其初始状态的。从测试的角度来看，这很重要，因为在反应性情况下，测试输入必须被制定为序列。系统可能是确定性的，即对于给定的输入，它始终返回相同的答案。然而，它也可能是非确定性的，即相同的输入可能会引发不同的输出（也许是因为随机的内部行为，或者其他无法控制的因素，比如线程调度）。通常很重要确保SUT是“生产”系统的一个孤立版本。

**规范** 规范代表被测系统的理想行为。根据开发环境的不同，这可能是作为**全面、严格维护的文档**（例如一组UML图或z规范）体现的，也可能是在敏捷环境中开发的，它可能是一部分直观描述，以用户故事、测试用例的形式捕获，并以注释的形式记录在源代码中。规范的性质对测试有明显的影响。**如果存在具体、可靠的规范文档，并且对系统应该做什么有共享的理解，那么这可以作为系统测试生成过程的基础。**如果情况不是这样，那么测试就变得更加临时和依赖于测试人员的**直觉和经验**。

**测试用例** 测试用例**对应于SUT的执行**。在实际情况下，一个测试用例对应于对系统的输入（或一系列输入）。测试用例应该理想地累积执行软件行为的每个独特方面。一个**理想的**测试集（测试用例的集合）应该能够**暴露出SUT与规范的任何偏差**。如果可以证明它能够做到这一点，那么测试集被认为是**足够的**。

**测试预期** 仅执行测试用例不足以确定SUT是否符合规范。**这个决定——一个测试的输出是否正确——由一个测试预期做出。**在实践中，测试预期可能是源代码中的断言，在测试执行期间进行检查，也可能是人类用户，决定行为是否可接受。

测试预期往往很难产生。实际上很少有明确的、全面的、最新的规范可以用作参考。一个成功的软件通常是在几十年的时间里由众多开发人员开发而成的，这意味着，最终，很少有确切记录系统应该如何行为的定义。此外，可能有成千上万个测试用例，每个测试用例可能产生复杂的输出，这可能会使手动验证输出的任务变得非常耗时。

**这些问题被统称为预期问题。**

# White Box Testing-白盒测试

- Access to software “**internals**”:
  - Source code源代码
  - Runtime state运行状态 | ↓跟踪执行过程
  - Can keep track of executions.
- White box testing exploits this to
  - **Use code to measure coverage**衡量覆盖率
    - Many different ways
  - Drive generation of tests that maximise coverage驱动生成最大程度覆盖率的测试

```
1  int tri_type(int a, int b, int c) {
2      int type;
3      if (a > b)
4          { int t = a; a = b; b = t; }
5      if (a > c)
6          { int t = a; a = c; c = t; }
7      if (b > c)
8          { int t = b; b = c; c = t; }
9      if (a + b <= c)
10         type = NOT_A_TRIANGLE;
11     else {
12         type = SCALENE;
13         if (a == b && b == c)
14             type = EQUILATERAL;
15         else if (a == b || b == c)
16             type = ISOSCELES;
17     }
18     return type;
19 }
```

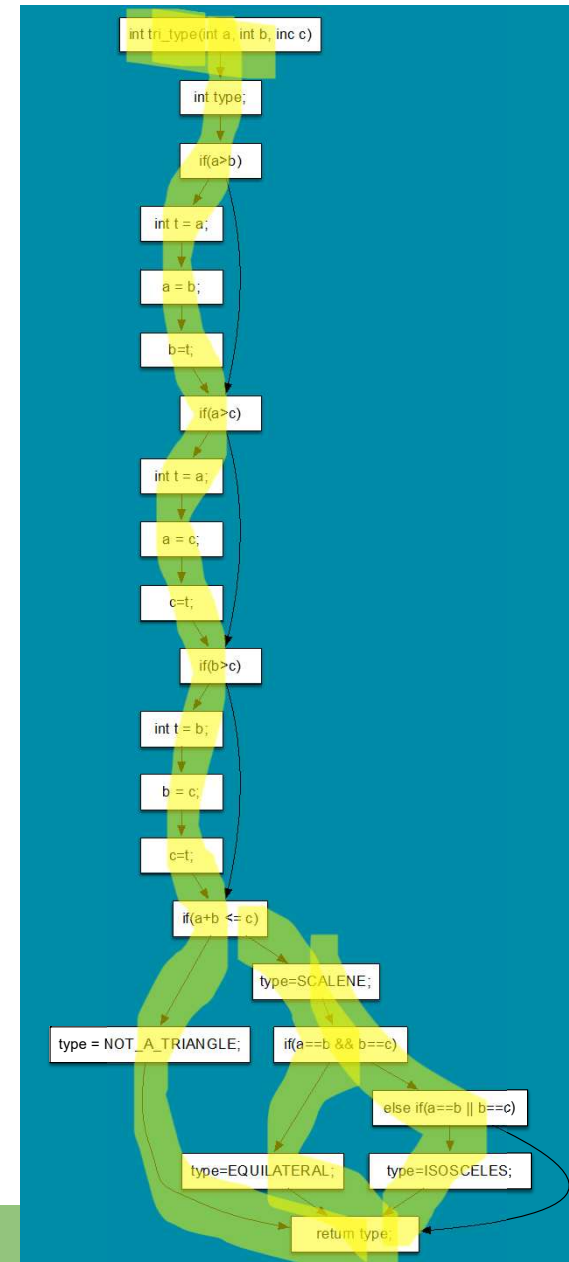
# White-Box Testing

- Coverage Metrics:
  - **Statement** coverage :The proportion of **executable statements** in the program that have been executed. 语句覆盖率：程序中已执行的可执行语句的比例 | ↓ 分支覆盖率：已执行的源代码中所有逻辑分支的比例
  - **Branch** coverage : The proportion of all of the **logic-branches** in the source code (e.g. outcomes of **IF, WHILE, or FOR** statements) to have been executed.
  - **Def-Use or Dataflow coverage** : The source code is analysed to extract the **def-use relations**, which relate statements at which a variable is defined (i.e. instantiated and given a value) to subsequent statements using that definition. The test-goal is to **cover all of the possible def-use relations**. 测试目标是覆盖所有可能的定义-使用关系。
  - MC/DC (Modified Condition / Decision Coverage)
  - Mutation coverage...
- Prescribed metrics, e.g., DO178-B/C standard for civilian aircraft software
  - non-critical - statement coverage
  - safety-critical - MC/DC coverage

## Statement Coverage-语句覆盖率

- Test inputs should collectively have executed each statement共同执行每个语句
- If a statement always exhibits a **fault** when executed, **it will be detected**
- Computed as:

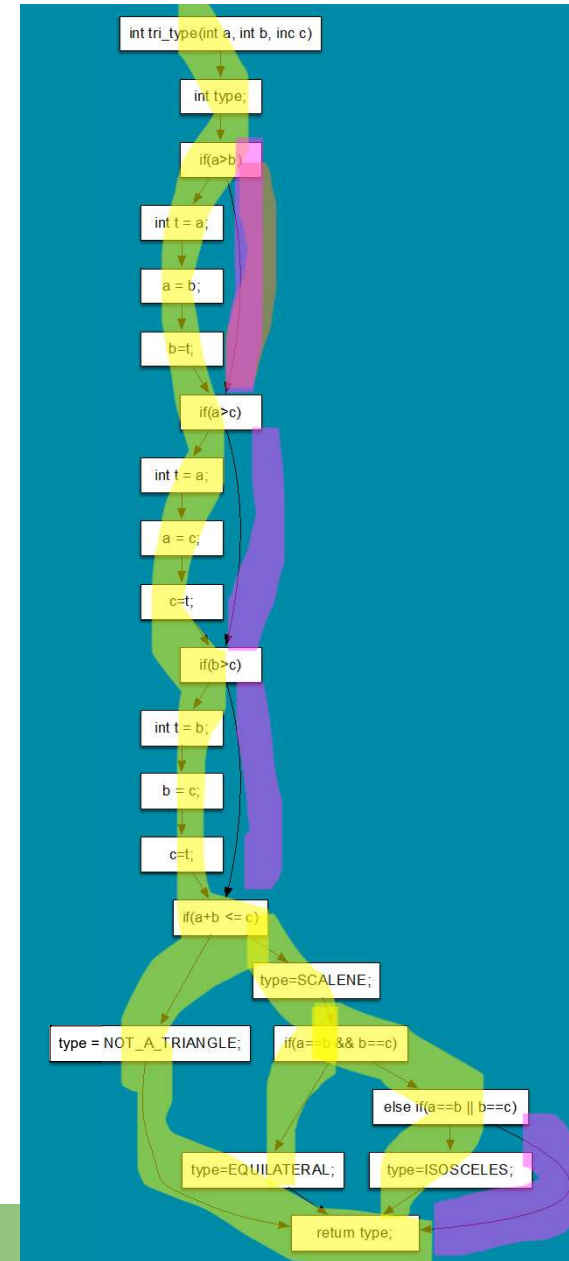
$$\text{Coverage} = \frac{|\text{Statements executed}| / \text{可执行语句}}{|\text{Total statements}| / \text{总语句}}$$



## Branch Coverage-分支覆盖率

- Test inputs should collectively have executed each branch 共同执行每个分支
- Subsumes statement coverage 包含语句覆盖率
- Computed as:

$$\text{Coverage} = \frac{|\text{Branches executed}|}{|\text{Total branches}|}$$



# Black Box Testing- 黑盒测试

- No access to “internals”
  - May have access, but **don't want to**
- We know the **interface**接口
- Parameters参数
  - Possible functions / methods
- We may have some form of **specification document**规范文档

## Equivalence Partitioning (EP) Method等价划分(EP)方法

Identify tests by **analysing the program interface**

1. Decompose program into “**functional units**”
2. Identify **inputs / parameters** for these units
3. For each input
  - a) Identify its **limits and characteristics**
  - b) Define “partitions” - **value** categories
  - c) Identify **constraints** between categories
  - d) Write **test specification**

## Testing Challenges – 测试挑战

- Many different types of input不同类型的输入
- Lots of different ways in which input choices can affect output影响有许多不同的方式
- An **almost infinite** number of possible inputs & combinations几乎无限数量的可能输入和组合

## Example – Generate Grading Component

*The component is passed an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it generates a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark which is calculated as the sum of the exam and c/w marks, as follows:*

*greater than or equal to 70 - 'A'*

*greater than or equal to 50, but less than 70 - 'B'*

*greater than or equal to 30, but less than 50 - 'C'*

*less than 30 - 'D'*


*Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers.*

## EP – 1. Decompose into Functional Units

- Dividing into **smaller units** is good practice
  - Possible to generate **more rigorous** test cases.
  - **Easier to debug** if faults are found.
- E.g.: dividing a large Java application into its core modules / packages
- Already a functional unit for the Grading Component example



## EP – 2. Identify Inputs and Outputs

- For some systems this is straightforward
  - E.g., the Triangle program:
    - Input: 3 numbers,
    - Output: 1 String
  - E.g., Grading Component
    - Input: 2 integers: exam mark and coursework mark
    - Output: 1 String for grade
- For others less so. Consider the following:
  - A phone app.
  - A web-page with a flash component.

## EP – 3.a Identify Categories

Category	Description
Valid	valid exam mark
	valid coursework mark
	valid total mark
Invalid	invalid exam mark
	invalid coursework mark
	Invalid total mark

## EP: 3.b Define “Partitions” - value categories

- Significant value ranges / value-characteristics of an input

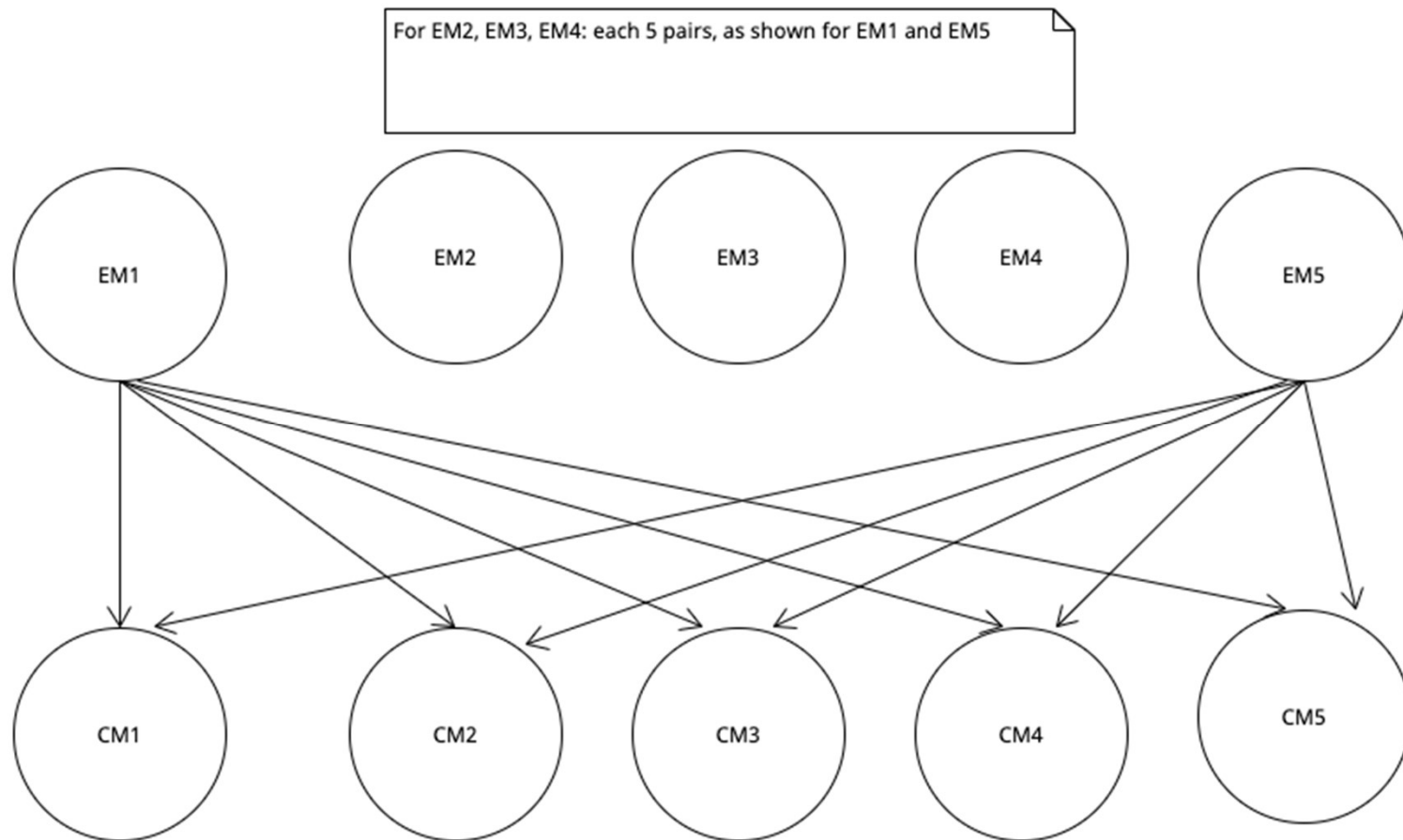
Category	Description	Partition
Valid	EM_1 valid exam mark	$0 \leq \text{Exam mark} \leq 75$
	CM_1 valid coursework mark	$0 \leq \text{Coursework mark} \leq 25$
Invalid	EM_2 invalid exam mark	Exam mark > 75
	EM_3 invalid exam mark	Exam mark < 0
	EM_4 invalid exam mark	alphabetic
	EM_5 invalid exam mark	Other real number (outside of EM_1)
	CM_2 invalid coursework mark	Coursework mark > 25
	CM_3 invalid coursework mark	Coursework mark < 0
	CM_4 invalid coursework mark	alphabetic
	CM_5 invalid coursework mark	Other real number (outside of CM_1)

## EP – 3. c Identify Constraints between Categories

- Not all categories can combine with each other

Category		Condition
valid exam mark	EM_1	$0 \leq \text{Exam mark} \leq 75$
invalid exam mark	EM_2	Exam mark > 75
invalid exam mark	EM_3	Exam mark < 0
invalid exam mark	EM_4	alphabetic
invalid exam mark	EM_5	Other real number
valid coursework mark	CM_1	$0 \leq \text{Coursework mark} \leq 25$
invalid coursework mark	CM_2	Coursework mark > 25
invalid coursework mark	CM_3	Coursework mark < 0
invalid coursework mark	CM_4	alphabetic
invalid coursework mark	CM_5	Other real number

## EP – 3. d Write Test Specifications



## Example: Inputs and Expected Outputs

The test cases corresponding to partitions derived from the input exam mark are:

Test Case	1	2	3
Input (exam mark)	44	-10	93
Input (c/w mark)	15	15	15
total mark (as calculated)	59	5	108
Partition tested (of exam mark)	$0 \leq e \leq 75$	$e < 0$	$e > 75$
Exp. Output	'B'	'FM'	'FM'

# Boundary Values – 边界值

- Most **frequently errors** occur in **"edge" cases**
  - Test just under boundary value
  - Test just above the boundary value
  - **Test the boundary value**

# How do we go about using this?

- Testing applied in **Java**: Use **JUnit**
  - uses “Assertions” to test the code
  - Allow us to **state what *should* be the case**
  - If assertions do not hold, JUnit’s logging mechanisms reports failures
  - Various types of assertion are available, e.g., assertEquals( expected, actual ); assertTrue( condition ); assertFalse( condition ); assertThat ( value, matchingFunction )



## Review

### What is **Software Quality**?

软件质量是指软件产品或系统在满足特定需求和预期功能的同时，其整体表现和属性的水平。它涵盖了多个方面，包括功能性、可靠性、性能、可维护性、可用性、安全性等。软件质量的好坏直接影响用户体验、系统的稳定性以及组织的声誉和成本。

主观上：Subjective or “fitness for use”: **as perceived by an individual user**

客观上：can be measured as a property of the product (e.g., detailed documentation, number of bugs, compliance with regulations .... )

### What are **key elements** and **relationships** for **test specifications**?

P6-9

### How do we **carry out white-box** testing?

1. Access to software “**internals**”
2. White box testing exploits this to Use code to **measure coverage**衡量覆盖率 (drive generation of tests that maximise coverage)

### How do we **carry out black-box** testing? No access to “internals”(May have access, but don’t want to)

1. We know the interface接口(Parameters参数 or Possible functions / methods)
2. We may have some form of specification document规范文档