5. Software quality

Here are some important areas to consider when evaluating software quality:

1. Functionality功能性: The software should meet its intended purpose and function as expected. This includes ensuring that all features work correctly and that the software operates as intended. 满足产品的功能
2. Reliability可信度: The software should be reliable and perform consistently under different conditions. This includes testing for errors, bugs, and failures, and ensuring that the software can recover from any issues.
3. Usability可用性: The software should be user-friendly and easy to use. This includes having an intuitive interface, clear documentation, and providing appropriate feedback to users.：软件应该是用户友好的，易于使用的。这包括有一个直观的界面，清晰的文档，并向用户提供适当的反馈。
4. Efficiency 有效性: The software should be efficient in its use of resources, such as memory and processing power. This includes optimizing code and minimizing resource usage to ensure the software performs well. 软件在使用资源方面应该是有效的，比如内存和处理能力。这包括优化代码和最大限度地减少资源的使用，以确保软件的良好性能。
5. Maintainability维护性: The software should be easy to maintain and modify over time. This includes using appropriate coding standards, documentation, and design patterns, as well as implementing automated testing and continuous integration processes.软件应该易于维护和长期修改。这包括使用适当的编码标准、文档和设计模式，以及实施自动化测试和持续集成流程。
6. Portability可移植性：: The software should be portable and able to run on different platforms and environments. This includes ensuring that the software is compatible with different operating systems, browsers, and devices. 软件应该是可移植的，能够在不同的平台和环境中运行。这包括确保该软件与不同的操作系统、浏览器和设备兼容。

Why is software quality relevant?

1. Customer satisfaction: High-quality software leads to a better customer experience, as it is more reliable, efficient, and user-friendly. This can lead to increased customer satisfaction and loyalty.
2. Cost-effectiveness: High-quality software is more cost-effective over the long-term, as it requires fewer resources to maintain and fix issues. This can lead to lower costs and higher profits.
3. Reputation: High-quality software can improve the reputation of the software development company, as well as the product itself. This can lead to more business opportunities and increased trust from customers.
4. Compliance遵守(legality): High-quality software is necessary to meet regulatory and legal requirements in certain industries, such as healthcare and finance.
5. Innovation: High-quality software enables software development teams to focus on innovation and new features, rather than fixing bugs and issues. This can lead to faster time-to-market and a competitive advantage in the marketplace.
6. Orginalzational certification :Certifications such as ISO/IEC 12207 or ISO/IEC 15504 provide guidelines for the software development process, including software quality. These certifications ensure that software development teams follow a consistent and rigorous 严密的process, which leads to higher quality software. By adhering to these certifications, organizations can demonstrate their commitment to quality and establish credibility with customers and stakeholders.
7. Moral/ethical codes of practice: The software development process also involves ethical considerations, such as protecting users' privacy and data, ensuring accessibility for all users, and avoiding bias in algorithms. Codes of practice, such as the ACM Code of Ethics and Professional Conduct, provide guidelines for ethical behavior in the software development industry. By following these codes of practice, software development teams can ensure that they are acting responsibly and ethically, which can improve their reputation and establish trust with customers and stakeholders.

Software quality is multi-dimensional

软件质量的多维度(subjective,objective, pratical)

- **Subjective** or “fitness for use”: as perceived by an individual user (e.g., aesthetics of GUI, missing functionality…)

主观方向来看或“适用性”：由单个用户感知（例如，GUI 的美学，缺少功能.....）

This aspect of software quality is based on the individual user's perception of the software's usefulness and how well it meets their specific needs. For example, a user may find a software's graphical user interface (GUI) aesthetically pleasing and easy to use, while another user may find it confusing or difficult to navigate.

主观或“适合使用”：软件质量的这一方面基于个人用户对软件有用性以及软件满足其特定需求的程度的看法。例如，一个用户可能会发现软件的图形用户界面 (GUI) 美观且易于使用，而另一个用户可能会发现它令人困惑或难以导航。

- **Objective** or “conformance to requirements”: can be measured as a property of the product (e.g., detailed documentation, number of bugs, compliance with regulations ….)

Objective or "conformance to requirements": This aspect of software quality is based on whether the software meets specific technical requirements, such as functionality, performance, reliability, and security. For example, a software's documentation may be measured based on its level of detail, while the number of bugs found may be used to evaluate its reliability.

客观或“符合要求”：可以作为软件的属性来衡量产品（例如，详细的文档、错误数量、法规遵从性……）软件质量的这一方面是基于软件是否满足特定的技术要求，例如功能、性能、可靠性和安全性。例如，可以根据软件的详细程度来衡量软件的文档，而可以使用发现的错误数量来评估其可靠性。

- **Practical**: what does it mean to your team and your clients?

实用性方面的考量: 对于我们的团队和客户意味着什么?

Practical: This aspect of software quality considers the software's impact on the development team and its clients. For example, the software should be easy to maintain and modify over time, and it should provide value to the clients that use it. Additionally, practical considerations include the cost and time required to develop, test, and deploy the software.

软件质量的这一方面考虑了软件对开发团队及其客户的影响。例如，软件应该易于维护和修改，并且应该为使用它的客户提供价值。此外，实际考虑因素包括开发、测试和部署软件所需的成本和时间。

通过考虑软件质量的所有三个维度，开发团队可以创建满足用户需求和期望、遵守技术要求并为客户提供价值的软件。这可以提高用户满意度、提高可靠性并取得更大的业务成功。

Quality Models: ISO/IES25010

ISO/IEC 25010 is a quality model that provides a framework for evaluating and improving software quality. The model consists of eight quality characteristics, each of which is further broken down into subcharacteristics:

ISO/IEC 25010 是一种质量模型，它提供了一个用于评估和改进软件质量的框架。该模型包含八个质量特征，每个特征又进一步细分为子特征：

1. Functional Suitability: This quality characteristic measures the extent to which the software's functionality meets the specified requirements. The subcharacteristics include accuracy, suitability, and interoperability. 功能适用性：此质量特性衡量软件功能满足指定要求的程度。子特性包括准确性、适用性和互操作性。
2. Performance Efficiency: This quality characteristic measures the software's performance in terms of speed, resource usage, and throughput. The subcharacteristics include time behavior, resource utilization, and capacity.性能效率：此质量特性衡量软件在速度、资源使用和吞吐量方面的性能。子特性包括时间行为、资源利用率和容量。
3. Compatibility: This quality characteristic measures the software's ability to operate in different environments and with other systems. The subcharacteristics include co-existence, interoperability, and replaceability. 兼容性：此质量特性衡量软件在不同环境中以及与其他系统一起运行的能力。子特性包括共存性、互操作性和可替换性。
4. Usability: This quality characteristic measures the software's ease of use and user satisfaction. The subcharacteristics include understandability, learnability, and operability.可用性：此质量特性衡量软件的易用性和用户满意度。子特性包括可理解性、可学习性和可操作性。
5. Reliability: This quality characteristic measures the software's ability to perform correctly and consistently over time. The subcharacteristics include maturity, fault tolerance, and recoverability.可靠性：此质量特性衡量软件随着时间的推移正确和一致地执行的能力。子特征包括成熟度、容错性和可恢复性。
6. Security: This quality characteristic measures the software's ability to protect data and prevent unauthorized access. The subcharacteristics include confidentiality, integrity, and availability.安全性：此质量特性衡量软件保护数据和防止未经授权访问的能力。子特性包括机密性、完整性和可用性。
7. Maintainability: This quality characteristic measures the software's ease of maintenance and modification. The subcharacteristics include modularity, reusability, and analyzability.可维护性：此质量特性衡量软件的维护和修改难易程度。子特性包括模块化、可重用性和可分析性。

8. Portability: This quality characteristic measures the software's ability to operate in different environments and on different platforms. The subcharacteristics include adaptability, installability, and replaceability.可移植性：此质量特性衡量软件在不同环境和不同平台上运行的能力。子特性包括适应性、可安装性和可替换性。

By evaluating software quality according to these characteristics and subcharacteristics, development teams can identify areas for improvement and ensure that the software meets the needs and expectations of users. Additionally, the ISO/IEC 25010 model provides a common language and framework for discussing software quality, which can improve communication and collaboration between development teams and stakeholders.

通过根据这些特性和子特性评估软件质量，开发团队可以确定需要改进的地方，并确保软件满足用户的需求和期望。此外，ISO/IEC 25010 模型提供了一种用于讨论软件质量的通用语言和框架，这可以改善开发团队和利益相关者之间的沟通和协作。

Steps Towards Software Quality:

- **Use a standard development process:**

Using a standard development process, such as Agile or Waterfall, can help ensure that the software is developed in a consistent and organized manner. This can reduce errors and improve overall quality.

- **Use a coding standard**

Using a coding standard, such as the MISRA C or C++ coding standards, can help ensure that the code is written consistently and adheres to industry best practices. This can improve code quality, readability, and maintainability.

1. Compliance with industry standards (e.g., ISO, Safety, etc.): such as ISO or safety standards, can help ensure that the software meets specific requirements for quality and safety. This can improve overall quality and help ensure that the software is safe and reliable.
2. Consistent code quality
3. Secure from start
4. Reduce development costs and accelerate time to market

- **Define and monitor metrics (defect metrics and complexity metrics)**

Defining and monitoring metrics, such as defect metrics and complexity metrics, can help identify potential issues early on in the development process. High complexity can lead to higher numbers of defects, so it's important to monitor these metrics and address any issues that arise.

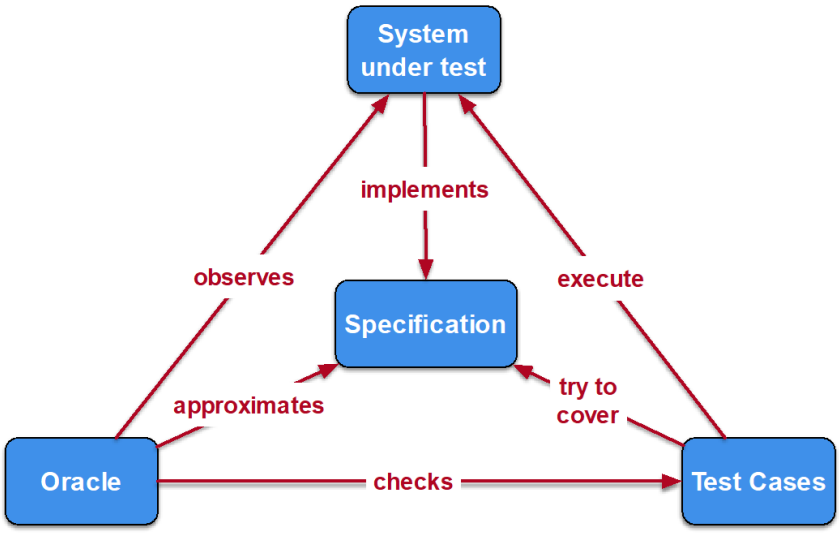
1. High complexity leads to higher number of defects

- **Identify and remove defects**

Conducting manual code reviews and using testing, such as unit testing and integration testing, can help identify and remove defects early on in the development process. This can reduce the number of defects that make it into the final product, leading to higher quality software.

1. Conduct manual reviews
2. Use Testing

Testing process: key elements and relationships



"Testing process: key elements and relationships" is a model proposed by Staats, Whalen, and Heimdahl in their paper "Programs, Tests, and Oracles: The Foundations

of Testing Revisited". The model outlines the key elements of the testing process and how they relate to one another.

The key elements of the testing process, as identified by the model, are as follows:

1. Program: The software program or system being tested.
2. Test: The set of inputs, operations, and procedures used to exercise the program or system and detect defects.
3. Oracle: The criteria or mechanism used to determine whether a test has passed or failed.
4. Test suite**测试套件**: The collection of tests used to evaluate the program or system. Test suites can be designed to cover different aspects of the program or system, such as functional requirements, non-functional requirements, and edge cases.
5. Test result: The output produced by executing a test. Test results can include information such as the inputs used, the outputs produced, and any error messages or exceptions that occurred during the test.
6. Test verdict: The outcome of a test, which indicates whether the test passed or failed. The test verdict is determined by comparing the test result to the oracle. If the test result matches the oracle, the test passes; otherwise, the test fails.

The relationships between these elements are also important. The program is the target of the testing process, and the tests are designed to detect defects in the program. The oracle is used to determine whether the tests have detected defects or not. The test suite is used to execute the tests and collect the results. The test results and verdicts are used to assess the quality of the program and identify any defects that need to be addressed.

By understanding the relationships between these key elements, software engineers can design and implement effective testing processes that improve the quality of their software. This model provides a useful framework for thinking about the testing process and identifying areas for improvement.

模型确定的测试过程的关键要素如下：

1. 程序：被测试的软件程序或系统。
2. 测试：用于运行程序或系统并检测缺陷的一组输入、操作和过程。
3. Oracle：用于确定测试是否通过的标准或机制。

4. 测试套件：用于评估程序或系统的测试集合。
5. 测试结果：执行测试产生的输出。
6. 测试判定：测试的结果，表明测试是通过还是失败。

这些元素之间的关系也很重要。程序是测试过程的目标，测试旨在检测程序中的缺陷。oracle 用于确定测试是否检测到缺陷。测试套件用于执行测试并收集结果。测试结果和判决用于评估程序的质量并识别需要解决的任何缺陷。

通过了解这些关键要素之间的关系，软件工程师可以设计和实施有效的测试流程，从而提高软件质量。该模型提供了一个有用的框架，用于思考测试过程和确定需要改进的领域。

White Box Testing:

- **Access to software ``internals'':**

1. Source code
2. Runtime state
3. Can keep track of executions.

- **White box testing exploits this to**

1. Use code to measure coverage
2. Many different ways
3. Drive generation of tests that maximise coverage

访问软件"内部"： 源代码 运行时状态 可以跟踪执行情况。 白盒测试利用此来 使用代码来测量覆盖率 许多不同的 方式 驱动生成最大化覆盖率的测试:

使用此信息的一种方法是在测试期间测量代码覆盖率。代码覆盖率是指测试套件已执行的代码的百分比。通过测量代码覆盖率，测试人员可以识别代码中尚未测试的区域，并设计额外的测试来覆盖这些区域。

白盒测试中有许多不同的方法来衡量代码覆盖率，例如语句覆盖率、分支覆盖率、路径覆盖率和循环覆盖率。这些技术中的每一种都测量代码的不同部分在测试期间执行的程度，并可用于指导其他测试的生成。

使用白盒测试技术，测试人员可以设计测试程序的内部工作并识别从外部可能看不到的缺陷。这有助于确保软件可靠、安全并按预期运行。

White box testing, also known as structural testing, is a software testing technique that focuses on examining the internal workings of a software system or program. In white box testing, the tester has access to the program's source code, architecture, and design, and uses this knowledge to design and execute tests that evaluate the correctness and completeness of the program's implementation.

White box testing techniques can be applied at different levels of the software development process, such as unit testing, integration testing, and system testing. Some common techniques used in white box testing include:

1. Statement coverage: This technique involves executing tests that cover all executable statements in the code. The goal is to ensure that every line of code is executed at least once during the testing process.
2. Branch coverage: This technique involves executing tests that cover all possible branches or decision points in the code. The goal is to ensure that all possible paths through the code are tested.
3. Path coverage: This technique involves executing tests that cover all possible paths through the code. The goal is to ensure that all possible combinations of decision points and conditions are tested.
4. Loop coverage: This technique involves executing tests that cover all possible iterations of loops in the code. The goal is to ensure that the code behaves correctly under different loop conditions and iterations.

Coverage Metrics(覆盖度量方法):

Coverage metrics are a set of quantitative measures used to evaluate the effectiveness of a testing process. These metrics are designed to measure the degree to which different parts of the code have been exercised by the tests.

Here are some examples of different coverage metrics:

1. Statement coverage(语句覆盖率): This metric measures the percentage of executable statements in the code that have been executed by the tests. A test suite that achieves 100% statement coverage executes every line of code at least once.：该指标衡量代码中已由测试执行的可执行语句的百分比。达到 100% 语句覆盖率的测试套件每行代码至少执行一次。

2. Branch coverage(分支覆盖率): This metric measures the percentage of decision points or branches in the code that have been exercised by the tests. A test suite that achieves 100% branch coverage executes every possible branch in the code.：该指标衡量代码中已被测试执行的决策点或分支的百分比。实现 100% 分支覆盖的测试套件会执行代码中的每个可能分支。
3. Def-Use or Dataflow coverage(Def-Use 或数据流覆盖率): This metric measures the percentage of dataflow paths in the code that have been exercised by the tests. A test suite that achieves 100% def-use or dataflow coverage executes every possible dataflow path in the code.此指标衡量代码中已被测试执行的数据流路径的百分比。实现 100% def-use 或数据流覆盖率的测试套件会执行代码中每个可能的数据流路径。
4. MC/DC (Modified Condition / Decision Coverage)修改条件/决策覆盖率: This metric measures the percentage of decision points in the code that have been exercised by the tests, while ensuring that every possible combination of conditions has been evaluated at least once.该指标衡量代码中已被测试执行的决策点的百分比，同时确保每个可能的条件组合至少被评估一次。
5. Mutation coverage突变覆盖率: This metric measures the effectiveness of the tests in detecting faults or defects in the code. It does this by introducing mutations, or small changes, to the code and measuring the percentage of mutations that are detected by the tests. 该指标衡量测试在检测代码中的错误或缺陷方面的有效性。它通过向代码引入突变或小的更改并测量测试检测到的突变的百分比来实现这一点。

Different software standards may prescribe different coverage metrics for testing, depending on the criticality of the software being developed. For example, the DO178-B/C standard for civilian aircraft software may require different coverage metrics depending on the criticality of the software. For non-critical software, statement coverage may be sufficient, while safety-critical software may require MC/DC coverage to ensure the safety and reliability of the software.不同的软件标准可能会规定不同的测试覆盖率指标，具体取决于所开发软件的关键程度。例如，民用飞机软件的 DO178-B/C 标准可能需要不同的覆盖指标，具体取决于软件的关键程度。对于非关键软件，声明覆盖可能就足够了，而安全关键软件可能需要 MC/DC 覆盖，以确保软件的安全性和可靠性。

Statement coverage:

Statement coverage is a code coverage metric that measures the degree to which a test suite has executed each line of code in the software being tested. The goal of statement coverage is to ensure that all executable statements in the code have been executed at least once during the testing process. 用于衡量测试套件执行被测软件中每一行代码的程度。语句覆盖率的目标是确保代码中的所有可执行语句在测试过程中至少被执行过一次。

To measure statement coverage, testers can use a tool that instruments the code and tracks which lines of code have been executed during testing. This information can then be used to generate a report that shows the percentage of statements that have been executed.

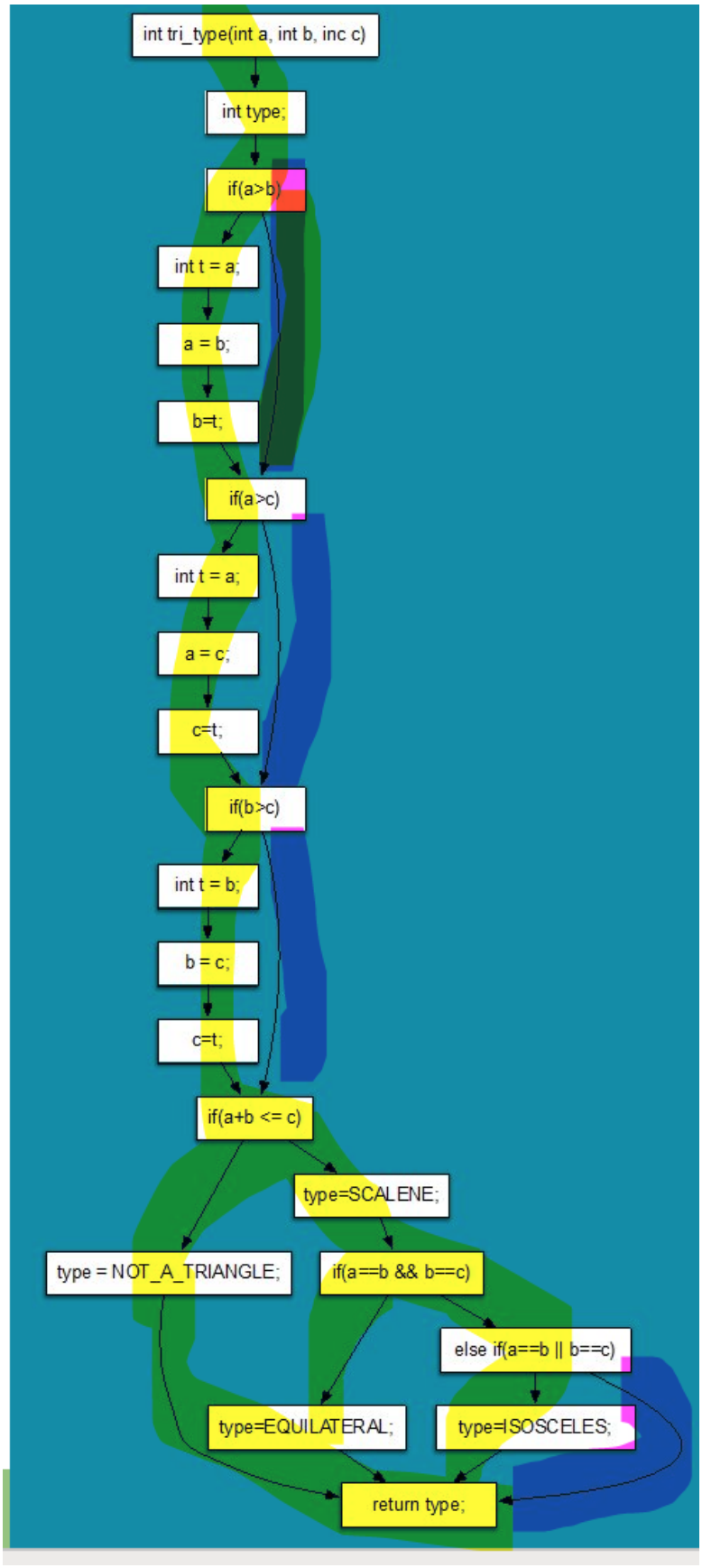
A test suite that achieves 100% statement coverage executes every line of code at least once, while a lower coverage percentage indicates that some parts of the code have not been exercised by the tests.

Statement coverage is a widely used metric in software testing, as it provides a simple and easy-to-understand measure of the effectiveness of a test suite. However, it does not guarantee that all possible behaviors of the software have been tested, as there may be multiple ways to execute the same line of code. To ensure full coverage, testers may need to use additional coverage metrics, such as branch coverage, path coverage, or MC/DC coverage.

- Test inputs should collectively have excuted each statement测试输入应该集体切除每个语句
- If a statement always exhibits a fault when executed, it will be detected 如果一条语句在执行时总是表现出故障，它将被检测出来

Computed as:

coverage = statemernts excuted/total branches


int tri_type(int a, int b, inc c)
int type;
if(a>b)
int t = a;
a = b;
b=t;
if(a>c)
int t = a;
a = c;
c=t;
if(b>c)
int t = b;
b = c;
c=t;
if(a+b <= c)
type=SCALENE;
type = NOT_A_TRIANGLE;
if(a==b && b==c)
else if(a==b || b==c)
type=EQUILATERAL;
type=ISOSCELES;
return type;

However, coverage is not strongly correlated with test suite effectiveness;

覆盖率与测试套件的有效性并不紧密相关

There are several reasons for this:

1. Coverage metrics measure only the code that has been executed by the test suite, not the code that has not been executed. Just because a test suite achieves high coverage does not mean that it has tested all possible scenarios or inputs. 覆盖率指标只衡量测试套件执行过的代码，而不是未执行的代码。一个测试套件实现了高覆盖率并不意味着它已经测试了所有可能的情况或输入。
2. Coverage metrics do not measure the quality of the tests themselves. Even if a test suite achieves high coverage, it may not be effective if the tests themselves are poorly designed or do not adequately test the software.覆盖率指标并不能衡量测试本身的质量。即使测试套件达到了高覆盖率，但如果测试本身设计得不好或没有充分测试软件，它也可能是无效的。
3. Coverage metrics do not take into account the severity of the defects that may exist in the untested code. Some defects may be more critical than others, and may require more rigorous testing to ensure their detection.覆盖率指标没有考虑到未测试代码中可能存在的缺陷的严重性。有些缺陷可能比其他缺陷更关键，可能需要更严格的测试来确保其被发现。

Testing: black box

Black box testing, also known as functional testing, is a software testing technique that focuses on evaluating the **external** behavior of a software system or program, without examining its internal workings. In black box testing, the tester is not concerned with the implementation details of the software, but rather with its functionality and how it interacts with the user and other systems.黑盒测试，也称为功能测试，是一种软件测试技术，侧重于评估软件系统或程序的外部行为，而不检查其内部工作原理。在黑盒测试中，测试人员不关心软件的实现细节，而是关心它的功能以及它如何与用户和其他系统交互。

The goal of black box testing is to ensure that the software meets its functional requirements and behaves correctly under different scenarios and inputs. Testers design test cases that cover different aspects of the software, such as functional requirements, performance, security, and usability, and execute them to detect defects and ensure that the software behaves as intended.

黑盒测试的目标是确保软件满足其功能要求并在不同场景和输入下正确运行。测试人员设计涵盖软件不同方面的测试用例，例如功能要求、性能、安全性和可用性，并执行它们以检测缺陷并确保软件按预期运行。

Black box testing techniques can be applied at different levels of the software development process, such as integration testing, system testing, and acceptance testing. Some common techniques used in black box testing include:黑盒测试技术可以应用于软件开发过程的不同级别，例如集成测试、系统测试和验收测试。黑盒测试中使用的一些常用技术包括：

1. Equivalence partitioning 等价划分: This technique involves dividing the input space into equivalence classes, where each class represents a set of inputs that are expected to behave the same way. Test cases are then designed to cover each equivalence class and ensure that the software behaves correctly for each one.这种技术涉及将输入空间划分为等价类，其中每个类代表一组预期行为相同的输入。然后设计测试用例以覆盖每个等价类，并确保软件对每个等价类都能正确运行。
2. Boundary value analysis 边界值分析: This technique involves testing the software at the boundaries of the input space, where unexpected behaviors are more likely to occur. Test cases are designed to test values that are at the upper and lower bounds of the input space, as well as values that are just inside and outside those bounds.此技术涉及在输入空间的边界处测试软件，此处更可能发生意外行为。测试用例旨在测试位于输入空间上限和下限的值，以及恰好在这些边界之内和之外的值。
3. Random testing 随机测试: This technique involves generating random inputs and executing them to test the software. Random testing can be an effective way to uncover unexpected behaviors and edge cases that may not be covered by other techniques.此技术涉及生成随机输入并执行它们以测试软件。随机测试是发现其他技术可能无法涵盖的意外行为和边缘情况的有效方法。

Black box testing can be very effective at detecting defects in the software's external behavior and ensuring that it meets its functional requirements. However, it may not be as effective at detecting defects that are related to the internal workings of the software,

such as performance issues or security vulnerabilities. To ensure full coverage, testers may need to use additional testing techniques, such as white box testing, performance testing, or security testing.黑盒测试可以非常有效地检测软件外部行为中的缺陷并确保它满足其功能要求。但是，它可能无法有效检测与软件内部工作相关的缺陷，例如性能问题或安全漏洞。为确保全面覆盖，测试人员可能需要使用额外的测试技术，例如白盒测试、性能测试或安全测试。

- **No access to "internals"**

May have access, but don't want to

In black box testing, the tester does not have access to the internal workings of the software and relies solely on the inputs and outputs of the software's interface. This means that the tester does not need to know the internal architecture or implementation details of the software.

在黑盒测试中，测试人员无法访问软件的内部工作原理，只能依赖软件界面的输入和输出。这意味着测试人员不需要知道软件的内部架构或实现细节。

- **We know the interface**

Parameters

Possible functions / methods

The tester focuses on understanding the software's interface, which includes the input parameters, possible functions or methods that can be invoked, and the expected output for different scenarios. The tester may also refer to a specification document or user requirements to guide their testing.测试人员专注于理解软件的界面，其中包括输入参数、可能调用的功能或方法，以及不同场景的预期输出。测试人员还可以参考规范文档或用户要求来指导他们的测试

- **We may have some form of specification document**

Black box testing is particularly useful when the software being tested has a complex internal structure or when the tester does not have the technical knowledge required to perform white box testing. Black box testing can also be used to test software components that are developed by different teams or organizations.当被测软件具有复杂

的内部结构或测试人员不具备执行白盒测试所需的技术知识时，黑盒测试特别有用。黑盒测试还可用于测试由不同团队或组织开发的软件组件。

Testing challengs

1. Input diversity 输入多样性: Software systems can have many different types of inputs, such as user input, network data, sensor data, or external APIs. Each input type may have different characteristics and requirements, which can make testing more complex.软件系统可以有许多不同类型的输入，例如用户输入、网络数据、传感器数据或外部 API。每种输入类型可能具有不同的特性和要求，这会使测试更加复杂。
2. Input-output relationships 输入-输出关系: The behavior of the software may be affected by the relationship between different inputs and outputs. For example, changing one input value may have a ripple effect on other parts of the system, leading to unexpected behaviors.软件的行为可能会受到不同输入和输出之间关系的影响。例如，更改一个输入值可能会对系统的其他部分产生连锁反应，从而导致意外行为。
3. Combinatorial explosion 组合爆炸: The number of possible inputs and combinations can quickly become overwhelming, making it impossible to test every possible scenario. Testers need to use techniques such as equivalence partitioning, boundary value analysis, and random testing to identify the most important and relevant inputs.可能的输入和组合的数量很快就会变得不堪重负，因此不可能测试每一种可能的场景。测试人员需要使用等价划分、边界值分析和随机测试等技术来识别最重要和相关的输入。
4. Complex interactions 复杂的交互: Software systems may have complex interactions between different components or subsystems, which can make it difficult to isolate and test individual parts of the system. Testers need to use techniques such as integration testing and system testing to ensure that all parts of the system work together as intended.软件系统可能在不同的组件或子系统之间有复杂的交互，这使得隔离和测试系统的各个部分变得困难。测试人员需要使用集成测试和系统测试等技术来确保系统的所有部分按预期协同工作

5. Changing requirements不断变化的需求: Requirements for the software may change over time, which can require testers to modify their testing approach and adapt to new scenarios and inputs.软件的需求可能会随着时间的推移而变化，这可能需要测试人员修改他们的测试方法并适应新的场景和输入。

- Many different types of input
- Lots of different ways in which input choices can affect output
- An almost infinite number of possible inputs & combinations

Equivalence Partitioning (EP) Method 等价划分

Identify tests by analysing the program interface

1. Decompose program into “functional units”
2. Identify inputs / parameters for these units
3. For each input

- Identify its limits and characteristics
- Define “partitions” - value categories
- Identify constraints between categories
- Write test specification

Equivalence partitioning is a black box testing technique that involves identifying inputs or parameters for a program, breaking them down into categories or partitions, and designing tests to ensure that each partition is tested.等价划分是一种黑盒测试技术，涉及识别程序的输入或参数，将它们分解为类别或分区，并设计测试以确保每个分区都经过测试。

Here are the steps involved in using the equivalence partitioning method:

1. Decompose program into “functional units”: Break the program down into smaller functional units that can be tested independently.将程序分解为“功能单元”：将程序分解为可以独立测试的更小的功能单元。

2. Identify inputs / parameters for these units: Identify the inputs or parameters that are needed for each functional unit.识别这些单元的输入/参数：识别每个功能单元所需的输入或参数。
3. For each input: Analyze each input or parameter and identify its limits and characteristics.对于每个输入：分析每个输入或参数并确定其限制和特征。
4. Identify its limits and characteristics: Determine the range of values that the input can take, as well as any constraints or limitations that may apply.确定其限制和特征：确定输入可以采用的值的范围，以及可能适用的任何约束或限制。
5. Define “partitions” - value categories: Group the values of the input into categories or partitions, based on their similarity or behavior. Each partition should represent a set of inputs that are expected to behave similarly.定义“分区”——值类别：根据输入值的相似性或行为，将输入值分组到类别或分区中。每个分区应该代表一组预期行为相似的输入。
6. Identify constraints between categories: Determine any constraints or dependencies between the input categories.识别类别之间的约束：确定输入类别之间的任何约束或依赖关系。
7. Write test specification: Use the input partitions to design tests that ensure each partition is tested and that the software behaves correctly for all inputs within each partition.编写测试规范：使用输入分区设计测试，确保每个分区都经过测试，并且软件对每个分区内的所有输入都能正确运行。

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

1. Possible to generate more rigorous test cases.
2. 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

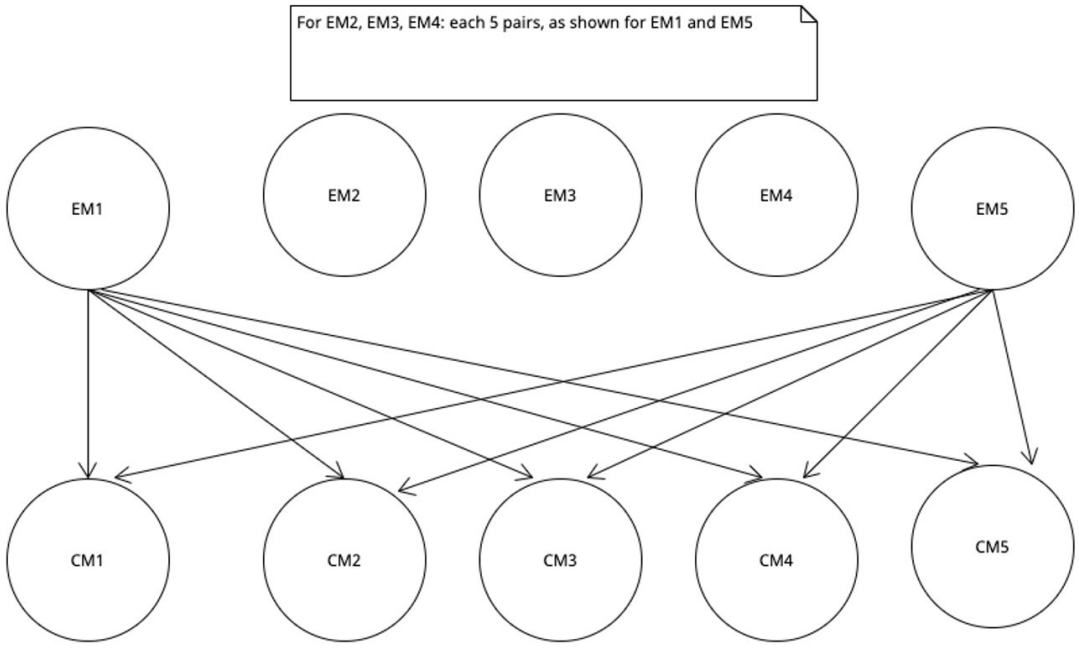
Category	Description	Partition
Valid	EM_1 valid exam mark	0 ≤ Exam mark ≤ 75
	CM_1 valid coursework mark	0 ≤ Coursework mark ≤ 25
Invalid	EM_2 invalid exam mark	Exam mark > 75
	EM_3 invalid exam mark	Exam mark < 0
	EM_4 invalid exam mark	alphabetic
	EM_5 invalid exam mark	real number
	CM_2 invalid coursework mark	Coursework mark > 25
	CM_3 invalid coursework mark	Coursework mark < 0
	CM_4invalid coursework mark	alphabetic
	CM_5 invalid coursework mark	real number

EP – 3. c Identify Constraints between Categories

• Not all categories can combine with each other

Category		Condition
valid exam mark	EM_1	0 ≤ Exam mark ≤ 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	real number
valid coursework mark	CM_1	0 ≤ Coursework mark ≤ 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	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

- Most frequently errors occur in "edge" cases

1. Test just under boundary value
2. Test just above the boundary value
3. Test the boundary value

How do we go about using this?

- Testing applied in Java unit
- Use JUnit

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

Review:

1. What is Software Quality?

Software quality refers to the degree to which a software product or system meets its functional and non-functional requirements, as well as its ability to satisfy the needs and expectations of its users and stakeholders. Software quality can be measured in various ways, including its reliability, usability, efficiency, maintainability, and security.

Software quality is important because it directly affects the performance, reliability, and user satisfaction of the software. High-quality software can help to reduce costs, increase productivity, and improve the user experience, while low-quality software can lead to errors, failures, and customer dissatisfaction.

To ensure software quality, software developers and testers must follow best practices and methodologies throughout the software development life cycle. This includes using standardized development processes, adhering to coding standards, and defining and monitoring metrics to track the quality of the software. Additionally, testers must use a combination of testing techniques, such as black box testing, white box testing, and others, to ensure that the software meets its functional and non-functional requirements and behaves correctly under different scenarios and inputs.

Overall, software quality is a critical factor in the success of any software project, and developers and testers must work together to ensure that software products and systems meet high standards of quality and reliability.

2. What are key elements and relationships for test specifications?

Test specifications are an important part of the software testing process, and they provide a detailed description of the tests that will be performed on the software system or product. Here are some of the key elements and relationships involved in test specifications:测试规范是软件测试过程的一个重要部分，它们提供了对软件系统或产品进行测试的详细描述。下面是测试规范中涉及的一些关键元素和关系：

1. Test cases: Test cases are the individual tests that will be performed on the software. Each test case should specify the inputs that will be used, the expected outputs, and any preconditions or postconditions that must be met for the test to be valid.测试用例： 测试用例是对软件进行的个别测试。每个测试用例应指定将使用的输入，预期的输出，以及任何必须满足的前提条件或后置条件，以使测试有效。

2. Test suites: Test suites are groups of related test cases that are designed to test specific aspects of the software system or product. Test suites should be organized in a logical and meaningful way, such as by feature, functionality, or priority.测试套件： 测试套件是一组相关的测试案例，旨在测试软件系统或产品的特定方面。测试套件应该以一种逻辑和有意义的方式组织起来，如按特征、功能或优先级
3. Test objectives: Test objectives are the goals and objectives of the testing process. They should be clearly defined and aligned with the overall project goals and requirements.测试目标： 测试目标是测试过程的目标和目的。它们应该被清楚地定义，并与整个项目的目标和要求保持一致。
4. Test environment: The test environment is the system or environment in which the tests will be performed. This should be specified in the test specifications to ensure that the tests are performed in a controlled and consistent manner.测试环境： 测试环境是指将进行测试的系统或环境。这应该在测试规范中明确规定，以确保测试以受控和一致的方式进行。
5. Test data: Test data is the data that will be used as input for the tests. This should be carefully selected to ensure that it covers a wide range of scenarios and inputs.测试数据： 测试数据是将被用作测试输入的数据。应该仔细选择，以确保它涵盖广泛的场景和输入。
6. Test procedures: Test procedures are the step-by-step instructions for performing the tests. They should be clear and easy to follow, and they should include any necessary setup or configuration steps.测试程序： 测试程序是执行测试的分步骤说明。它们应该是清晰的，易于遵循的，而且应该包括任何必要的设置或配置步骤。
7. Test results: Test results are the outcomes of the tests, including any defects or issues that are identified. Test results should be carefully recorded and tracked to ensure that all issues are properly addressed and resolved.测试结果： 测试结果是测试的结果，包括发现的任何缺陷或问题。测试结果应该被仔细记录和跟踪，以确保所有的问题都被适当处理和解决。

Overall, test specifications are a critical part of the software testing process, and they should be carefully designed and executed to ensure that the software system or product meets its functional and non-functional requirements and behaves correctly under different scenarios and inputs.

3. How do we carry out white-box testing?

1. Identify the code to be tested: White-box testing involves examining the code of the software system or product to identify potential issues and defects. Testers should first identify the parts of the code that need to be tested, such as specific functions, modules, or components.
2. Develop test cases: Test cases should be designed to exercise the different parts of the code and test its functionality and behavior. Test cases can be designed manually or automatically, and they should cover different scenarios and inputs.
3. Execute the tests: Once the test cases have been developed, they should be executed to verify the functionality of the code. The tests should be run in a controlled environment to ensure that the results are consistent and repeatable.
4. Evaluate the results: The results of the tests should be evaluated to determine whether the code is functioning correctly and whether any defects or issues have been identified. Any defects that are found should be reported and tracked for resolution.
5. Modify the code as necessary: If defects or issues are identified during testing, the code should be modified as necessary to correct the issues. The modified code should then be re-tested to ensure that the issues have been resolved.

White-box testing is often performed by developers or other technical experts who have knowledge of the internal workings of the software system or product. It can be an effective way to identify defects and issues that may not be detected through other testing techniques, such as black-box testing. However, it can be time-consuming and may require specialized tools and expertise to perform effectively.

4. How do we carry out black-box testing?

Black-box testing is a software testing technique that focuses on evaluating the external behavior of the software system or product, without looking at its internal workings or code. Here are the steps involved in carrying out black-box testing:

1. Identify the requirements to be tested: Black-box testing involves identifying the functional and non-functional requirements of the software system or product, and designing tests to ensure that these requirements are met. Testers should first identify the specific requirements that need to be tested, such as input validation, output correctness, or error handling.

2. Develop test cases: Test cases should be designed to exercise the different requirements of the software system or product, and to test its functionality and behavior under different scenarios and inputs. Test cases can be designed manually or automatically, and they should cover different scenarios and inputs.
3. Execute the tests: Once the test cases have been developed, they should be executed to verify the functionality of the software system or product. The tests should be run in a controlled environment to ensure that the results are consistent and repeatable.
4. Evaluate the results: The results of the tests should be evaluated to determine whether the software system or product is functioning correctly and whether any defects or issues have been identified. Any defects that are found should be reported and tracked for resolution.
5. Modify the software system or product as necessary: If defects or issues are identified during testing, the software system or product should be modified as necessary to correct the issues. The modified software system or product should then be re-tested to ensure that the issues have been resolved.

Black-box testing can be performed by testers who may not have detailed knowledge of the internal workings or code of the software system or product. It can be an effective way to identify defects and issues related to the external behavior of the software system or product, such as input validation or output correctness. However, it may not be as effective at identifying issues related to the internal workings or architecture of the software system or product, such as performance or scalability issues.

Easy Quizs:

1. What is software quality?
   a. The degree to which a software product meets its functional and non-functional requirements
   b. The ability of the software to satisfy the needs and expectations of its users and stakeholders
   c. Both a and b
   d. None of the above

2. What is white-box testing?
   a. A software testing technique that focuses on evaluating the external behavior of the software system or product
   b. A software testing technique that focuses on evaluating the internal workings of the software system or product
   c. Both a and b
   d. None of the above
3. What is black-box testing?
   a. A software testing technique that focuses on evaluating the internal workings of the software system or product
   b. A software testing technique that focuses on evaluating the external behavior of the software system or product
   c. Both a and b
   d. None of the above
4. What is equivalence partitioning?
   a. A black-box testing technique that involves breaking down inputs into categories or partitions and designing tests to ensure that each partition is tested
   b. A white-box testing technique that involves examining the code of the software system or product to identify potential issues and defects
   c. A test specification technique that involves defining the step-by-step instructions for performing the tests
   d. None of the above
5. What is statement coverage?
   a. A type of white-box testing that measures the percentage of statements in the code that have been executed during testing
   b. A type of black-box testing that measures the percentage of requirements that have been tested during testing
   c. A type of test specification that defines the inputs and expected outputs of a test case
   d. None of the above

Answers:

1. c
2. b

3. b
4. a
5. a

Harder one(maybe):

1. Which of the following is NOT a dimension of software quality?
   a. Reliability
   b. Usability
   c. Maintainability
   d. Interoperability互操作性
   e. All of the above are dimensions of software quality
2. Which of the following is NOT a type of black-box testing?
   a. Equivalence partitioning
   b. Boundary value analysis
   c. Decision table testing
   d. Statement coverage
   e. All of the above are types of black-box testing
3. Which of the following is a measure of test coverage?
   a. Cyclomatic complexity
   b. Control flow graph
   c. Def-use coverage
   d. Both a and b
   e. All of the above
4. Which of the following is an advantage of using white-box testing?
   a. It can help to identify defects and issues related to the external behavior of the software system or product
   b. It can be performed by testers who may not have detailed knowledge of the internal workings or code of the software system or product
   c. It can be an effective way to identify issues related to the internal workings or architecture of the software system or product
   d. Both a and c
   e. None of the above

5. Which of the following is a key element of test specifications?
   a. Test coverage
   b. Test cases
   c. Test data
   d. Test procedures
   e. All of the above

Answers:

1. e
2. d
3. e
4. c
5. e

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b1aafd50-d267-4594-85fe-0c8a740d8016/Code_Inspection_Checklist.pdf