软件质量与保证测试

软件质量与保证测试

- 1. 考前注意
- 2. 第一部分
  - 2.1 总体介绍
    - 2.1.1 QA（quality assurance）和test（QC）的区别?
    - 2.1.2 灰盒测试 gray-box testing
    - 2.1.3 Acceptance test（用户验收测试）和 system test（系统测试）粒度granularity区别
    - 2.1.4 测试的分类
    - 2.1.5 回归测试和冒烟测试
    - 2.1.6 α测试和β测试
    - 2.1.7 静态测试 & 动态测试
    - 2.1.8 Traceability可追溯性
    - 2.1.9 Coverage 覆盖范围
  - 2.2 测试计划
    - 2.2.1 迭代软件开发
    - 2.2.2 需求阶段QA的核心工作
    - 2.2.3 全功能提交团队
    - 2.2.4 EC & AC
    - 2.2.5 软件度量的指标
    - 2.2.6 圈复杂度
    - 2.2.7 缺陷-瑞利分布
  - 2.3 测试用例
    - 2.3.1 概述 Overview
    - 2.3.2 理论 Theory
    - 2.3.3 分类原则 Partitioning Principle
    - 2.3.4 BVA Boundary Value Analysis 边界值分析
    - 2.3.5 Dependency Between Parameters
  - 2.4 测试执行
    - 2.4.1 Entry criteria & exit criteria
    - 2.4.2 Suspension & Resumption criteria
    - 2.4.3 “successful” test execution
    - 2.4.4 effective defect principles
  - 2.5 测试提交
    - 2.5.1 提交什么
    - 2.5.2 Burn Down Chart 燃尽图
- 3. 第三部分
  - 3.1 unit test
    - 3.1.1 Why we need unit test
    - 3.1.2 Classic pattern of unit test
    - 3.1.3 How to perform the unit test
    - 3.1.4 Difference between Unit test and white box test
- 4. 白盒测试三种测试方法
  - 4.1 名词解释
- 5. 历年卷 40%雷同
  - 5.1 2023-2024
    - 5.1.1 Q1 解释概念
    - 5.1.2 Q2 简答题
    - 5.1.3 Q3 Unit test 和 white box test 的区别与联系
    - 5.1.4 Q4 为什么要尽早地开展 test
    - 5.1.5 Q5 设计等价类

5.1.6 Q6 我们一般认为 statement coverage < branch coverage < condition coverage，给出一个反例否定这一关系。

1. 考前注意

Important

要动脑的题：

1. 白盒测试、写测试用例、CFD（control flow diagram）
2. 给定场景设计测试用例：电梯、计算器、遥控器、方向盘（转向灯）、开关
   - 划分等价类
   - 边界值测试用例（不用太纠结）

Important

要抄的题

- 区别
  - 不一定在一张PPT上
  - 核心区别
- 罗列
  - 不要多写
- 特点
  - 不同分类体系下的划分？联系和区别
- 词
  - 写准

Caution

考试范围：第二部分无题目，第一部分多

	系统级	单元级
功能性	第一部分 概念与框架	第三部分 单元测试
非功能性	第二部分 自动化测试与非功能性测试	

Table 1

unit test第一个判断会改变第二个判断的值

总+测试计划+报告

第三部分要抄的东西：unit test

2. 第一部分

2.1 总体介绍

单元测试（unit test）&白盒测试（White-box testing）？

> 单元测试——技术
>
> 白盒测试——方法

2.1.1 QA（quality assurance）和test（QC）的区别?

> QA是目标，test是实现目标最重要的方法

Quality Assurance (QA) and Testing are closely related but distinct concepts in the software development process. While both aim to ensure the quality of the software, they differ in scope, focus, and approach.

1. **Definitions**:

- **Quality Assurance (QA)**: QA is a comprehensive process that spans across the entire software development lifecycle. Its goal is to improve the quality of the development process by ensuring adherence to best practices, standards, and procedures, ultimately preventing defects. QA focuses on preventing defects before they occur by ensuring that processes are followed correctly throughout the project.
- **Testing**: Testing is a specific part of QA that involves the actual process of checking and verifying the software to ensure it behaves as expected. It is about identifying defects in the software product by executing it under controlled conditions. Testing is focused on confirming that the software meets the specified requirements and behaves as intended.

2. **Focus**:

- QA: QA focuses on process improvement and the adherence to standards. It aims at ensuring that the entire development lifecycle follows appropriate quality standards and methodologies to prevent errors from occurring in the first place.
- Testing: Testing focuses on validation. It is concerned with checking whether the software product functions correctly, meets requirements, and is free of defects. Testing is about finding flaws and verifying that the software behaves as expected.

3. **Scope**:

- **QA**: QA has a broader scope and includes:
  - Establishing quality standards and processes.
  - Evaluating and improving development methods and tools.
  - Monitoring and managing the software development process.
  - Conducting process reviews to ensure the team follows best practices.
- **Testing**: Testing has a narrower scope and is concentrated on the actual verification of the software. Testing activities include:
  - Writing test plans and test cases.
  - Executing manual or automated tests.
  - Identifying and logging defects.
  - Reporting issues found during testing.

4. **Goals**:

- QA: The goal of QA is to prevent problems from occurring by ensuring that the entire development process adheres to predefined quality standards.
- Testing: The goal of testing is to verify that the software meets expectations and requirements by finding and fixing defects, ensuring the software functions correctly.

5. **Roles and Activities**:

- QA: QA teams are typically involved throughout the entire software development lifecycle, from requirements gathering, design reviews, and quality management during development, to final quality audits upon delivery. They may also be involved in process improvement, quality training, and ensuring compliance with standards.
- Testing: Testing teams are focused on activities such as writing test cases, executing tests, tracking defects, and improving testing efficiency through automation. They primarily focus on functional testing, performance testing, security testing, and verifying the software's behavior.

6. **Prevention vs. Detection**:

- QA: QA is a preventive activity aimed at reducing the likelihood of defects by improving the development process and establishing quality practices.
- Testing: Testing is a detection activity aimed at finding defects by executing the software and verifying its behavior against the requirements.

Summary:

- QA is a broader concept that ensures the quality of the entire development process, focusing on process improvement to prevent defects.
- Testing is a crucial component of QA, focusing on validating the quality of the software product by finding and fixing defects.

In short, QA focuse on the objective, and Testing is the most important means to achieve the objective.

2.1.2 灰盒测试 gray-box testing

In software engineering, the term **"gray-box testing"灰盒测试** is often considered to lack a strict definition because it is not a completely distinct or formal testing methodology. Instead, it is a hybrid approach that lies somewhere between black-box testing and white-box testing. Here are several reasons why "gray-box testing" is often said to lack a strict meaning:

1. **Ambiguous Definition**:
    - Black-box testing focuses on testing the software's inputs and outputs, without considering the internal implementation details of the software. Testers design test cases based solely on functional requirements and specifications.
    - White-box testing focuses on testing the internal logic and structure of the software. Testers need to have knowledge of the source code and test paths, branches, and internal structures.

Gray-box testing does not have a clear, universally accepted definition. It typically refers to testing in which the tester has partial knowledge of the internal architecture or code (such as API or database structure) but does not need to fully understand the code itself. This definition is somewhat vague and lacks clear boundaries, which is why it is not considered a strict or formal testing type.

2. **Fusion of Black-box and White-box Testing 黑盒与白盒测试的融合**:
   Gray-box testing essentially combines features of both black-box and white-box testing. Testers generally design test cases based on some internal information of the system, such as architecture or APIs, but without fully understanding the source code. Thus, gray-box testing cannot be strictly categorized as either white-box or black-box testing; it is a blend of both.
3. **Execution-Level Hybrid**:
   - In practice, gray-box testing is often carried out by testers who have some knowledge of the internal system design, like APIs, databases, or system architecture, while still focusing on the external functionality of the software. This makes it a compromise between having no knowledge of the system and having full knowledge of the code.
   - However, strictly speaking, black-box testing and white-box testing have well-defined boundaries: black-box testing disregards internal details, while white-box testing requires full understanding of internal implementation. Gray-box testing cannot clearly specify "how much internal information is enough," which makes it harder to define strictly.
4. **Confusion with Different Definitions**:
   Different organizations or industries may use the term "gray-box testing" with varying interpretations, adding to the confusion. In some contexts, gray-box testing may refer to having some understanding of external interfaces or database structures, while in other cases, it may imply a deeper understanding of system architecture. This lack of consistency leads to a blurred line of what constitutes gray-box testing.
5. **Flexibility in Testing Approach**:
   Since gray-box testing emphasizes partial knowledge of the system’s internal structure while combining aspects of black-box and white-box testing, it is highly flexible in practical scenarios. Due to the absence of a standardized definition, many testing teams view it more as a methodological compromise or flexible application rather than a strictly defined, formal testing type.

2.1.3 Acceptance test（用户验收测试）和 system test（系统测试）粒度 granularity区别

From a granularity perspective, UAT (User Acceptance Testing) and System Testing are not of the same granularity. They differ in terms of scope, focus, and depth. Here are the main differences:

1. **Scope of Testing**:
   - UAT: The scope of user acceptance testing is generally smaller and focuses primarily on validating whether the software meets the user’s needs and business objectives. UAT is usually performed by end users or customers, who verify if the system works as expected in real-world scenarios and if it meets business requirements. UAT focuses on the business processes and whether the system can function properly in a production-like environment.

   - System Testing: System testing has a broader scope and covers the validation of the entire system's functionality and performance. It includes testing all functional components and verifying that the various parts of the system interact and integrate correctly. System testing is typically done by professional testers and aims to ensure that the system meets all technical and functional specifications, including functional, performance, security, and compatibility testing.
2. **Focus of Testing**:
   - UAT: The focus of UAT is primarily on requirements and user experience, testing whether the system meets the expectations of the end users. UAT is based on requirement documents and business workflows, and users participate to ensure the system meets their practical needs, ease of use, and whether key functions work correctly.
   - System Testing: The focus of system testing is on the overall functionality and technical implementation of the system. It ensures that all technical requirements (e.g., performance, interfaces, error handling, security) are met. System testing involves validating both individual functions and the integration of those functions, testing the system under various conditions.
3. **Depth of Testing**:
   - UAT: The depth of UAT is generally shallow, focusing mainly on high-level business workflows and whether the system meets the user's expectations. Testers usually do not get involved with the technical details of the code but instead focus on usability, business process flow, and overall functionality.
   - System Testing: System testing has a much deeper depth, covering a wide range of tests. It includes unit testing, integration testing, regression testing, and other levels of testing, examining the system's technical details to ensure all components work together in various scenarios.
4. **Testing Environment**:
   - UAT: User acceptance testing is typically conducted in the real-world operational environment or a simulation of it. This allows the end users to test the system in conditions that closely resemble their actual work environment.
   - System Testing: System testing is typically performed in a dedicated test environment that simulates various use cases. This environment is often separate from the production environment and is designed to test the system under different conditions before deployment.
5. **Test Executers**:
   - UAT: UAT is usually executed by end users or business representatives, who test the system based on their own practical needs and business processes.
   - System Testing: System testing is typically carried out by test engineers or QA teams, who design and execute tests based on requirement and design documentation to verify the technical implementation of the system.

2.1.4 测试的分类

1. **Functionality Testing**
    - Corresponding McCall’s Factor: Correctness
    - Explanation: Functionality testing primarily verifies if the software performs the functions it is intended to do and meets the required business objectives. The “Correctness” factor in McCall's model assesses whether the software behaves as expected and fulfills the specified requirements.
2. **Performance Testing**
    - Corresponding McCall’s Factor: Efficiency
    - Explanation: Performance testing focuses on the software’s responsiveness, throughput, and resource usage. It ensures that the system performs well under high load. The “Efficiency” factor in McCall's model evaluates how well the software uses resources, including processing speed, memory consumption, and response time.
3. **Security Testing**
    - Corresponding McCall’s Factor: Integrity
    - Explanation: Security testing ensures that the system is protected from unauthorized access, data breaches, and other security threats. The “Integrity” factor assesses the software’s ability to protect data, ensuring that it remains accurate, consistent, and free from unauthorized modifications.
4. **Availability Testing**
    - Corresponding McCall’s Factor: Reliability
    - Explanation: Availability testing validates whether the system can continue operating in the event of failures and whether it has recovery mechanisms in place. The “Reliability” factor in McCall's model evaluates the software's ability to perform consistently over time and recover from faults.
5. **Compatibility Testing**
    - Corresponding McCall’s Factor: Portability
    - Explanation: Compatibility testing verifies whether the software can run on different operating systems, devices, browsers, or network environments. The “Portability” factor measures the software’s ability to be transferred or adapted to different platforms or environments.
6. **Consistency Testing**
    - Corresponding McCall’s Factor: Maintainability
    - Explanation: Consistency testing ensures that the software remains consistent across versions, including interfaces, data formats, and behavior. The “Maintainability” factor in McCall's model assesses how easy it is to modify and maintain the software, and consistency in behavior and code helps with ongoing maintenance.

2.1.5 回归测试和冒烟测试

1. **Regression Testing**

Definition: Regression testing is a type of testing conducted to verify that new changes, enhancements, or bug fixes have not introduced any unintended side effects or bugs into the existing functionality of the software. It ensures that previously working features continue to work as expected after changes are made to the codebase.

Purpose: The main purpose of regression testing is to ensure that modifications to the software (whether adding new features, fixing bugs, or making improvements) do not break or negatively impact existing functionality.

When It Is Done: Regression testing is performed after changes such as code modifications, bug fixes, new features, or updates to the software. It is typically done frequently during the software development lifecycle.

Scope: Regression testing covers a broad scope, including re-testing old features, modules, or functionalities to ensure they still work as expected after the new changes. It may involve testing the entire system or specific modules that could be impacted by the changes.

Example: After adding a new feature for user authentication, regression testing would check if other features, such as user registration and profile management, still work as intended.

2. **Smoke Testing**

Definition: Smoke testing, also known as "build verification testing," is a type of preliminary testing conducted to check whether the most critical and basic functionalities of the software work correctly after a new build or version is deployed. The goal is to verify if the software is stable enough for further, more detailed testing.

Purpose: The main purpose of smoke testing is to quickly assess whether a software build is stable enough to proceed with more extensive testing. It checks if the core features are working and if there are any obvious defects that would make further testing impossible or unproductive.

When It Is Done: Smoke testing is usually performed immediately after receiving a new build or software version. It serves as an initial check before more comprehensive functional, integration, or system tests are conducted.

Scope: Smoke testing has a very narrow scope compared to regression testing. It typically focuses on verifying that essential functionalities (such as launching the application, logging in, performing basic operations) are working. Smoke tests usually do not go into deep functionality or detail but ensure that critical parts of the system are functioning.

Example: After deploying a new version of a web application, smoke testing would check if users can log in, view basic pages, and perform simple tasks (such as navigating to the homepage) without errors.

2.1.6 α测试和β测试

Alpha Testing and Beta Testing are two different phases of software testing that are performed before a software product is officially released to the public. They are both important in ensuring the quality and reliability of the software, but they have distinct purposes, approaches, and target audiences. Below is a detailed comparison of Alpha Testing and Beta Testing:

1. **Alpha Testing**

Definition: Alpha testing is the first phase of testing conducted by the development team within the organization before releasing the software to external users. It is performed in a controlled environment, usually by the internal QA team or sometimes the development team itself.
Purpose: The main goal of alpha testing is to identify bugs and issues that are critical to the

product’s functionality. It focuses on catching errors early in the development process and ensuring that the software is stable enough to be used by external testers.
Testers: Alpha testing is performed by internal testers, which can include developers, quality assurance (QA) teams, and sometimes a small group of selected end-users within the company.
Environment: The testing environment for alpha testing is typically the development or staging environment, which is controlled by the organization.
Focus: It generally focuses on both functional and non-functional aspects of the application, checking for correctness, usability, performance, and security issues.
Duration: Alpha testing lasts for a longer period compared to beta testing. The duration can range from a few weeks to several months, depending on the complexity of the software.
Outcome: The outcome of alpha testing is the identification of critical issues that need to be addressed before the software is released to a wider audience.

2. **Beta Testing**

Definition: Beta testing is the second phase of testing, conducted by a selected group of external users outside of the organization. It is done after alpha testing has been completed and is aimed at gathering feedback from real-world users in a real production environment.
Purpose: The primary goal of beta testing is to identify any remaining issues, bugs, or usability problems that were not caught during alpha testing. It allows the software to be tested by actual end-users to see how it behaves in different environments and use cases.
Testers: Beta testing is typically conducted by a group of external users, also known as "beta testers," who can be either voluntary or invited by the company. These testers are often selected based on their knowledge, technical skills, or experience with similar software.
Environment: The software is tested in real-world environments, meaning users install and use the software in their personal or professional settings.
Focus: Beta testing focuses on user feedback, usability, and identifying any minor or subtle bugs that were not caught earlier. It also helps evaluate the overall user experience (UX) and how the software performs in diverse environments.
Duration: Beta testing typically lasts for a shorter period compared to alpha testing. The duration can range from a few weeks to a month, depending on the product and the scope of testing.
Outcome: The outcome of beta testing is feedback from real users, including bug reports, suggestions for improvements, and insights into how the software performs in real-world scenarios.

Similarities Between Alpha and Beta Testing:
Both are conducted before the software is officially released to the public.
Both aim to improve the quality and stability of the software by identifying bugs and issues.
Both require systematic reporting of issues found during testing, which are then prioritized and addressed.

2.1.7 静态测试 & 动态测试

Aspect	Static Testing	Dynamic Testing
Test Object	Software artifacts like code, documentation, and requirements.	Executed code and its behavior during runtime.
Code Execution	Does not require code execution.	Requires code execution to evaluate behavior.

Aspect	Static Testing	Dynamic Testing
Main Purpose	Detect defects early, ensure design and code quality, and verify requirements.	Validate software functionality, performance, and behavior in real scenarios.
Testing Process	Performed via reviews, inspections, and static analysis tools.	Performed by executing code, checking outputs, and observing behavior.
Tools Used	Static analysis tools, review checklists, inspection tools.	Automated testing tools, manual testing, performance monitoring tools.
Focus	Code structure, design defects, coding standards violations.	Functional correctness, performance, security, and runtime issues.
Advantages	Early identification of issues, improves code quality, can catch issues that dynamic testing may miss.	Validates the actual behavior, detects runtime issues, ensures the software meets functional requirements.
Disadvantages	Cannot detect runtime issues like performance and memory leaks.	Requires more resources and may not detect structural or design flaws.

Table 2

2.1.8 Traceability可追溯性

Definition: Traceability in software testing refers to the ability to link or map test cases to specific requirements, design elements, or code components. It ensures that every requirement has a corresponding test case and that the test results can be traced back to the original requirement or feature being tested.

Purpose: The primary purpose of traceability is to ensure requirements coverage and help in tracking the relationship between tests, requirements, and defects. It provides transparency and enables stakeholders to see how testing aligns with business goals and functional requirements.

Key Aspects:
Requirements Traceability: Ensures that each requirement or user story has an associated test case that validates its implementation.
Design and Code Traceability: Helps map tests to specific design elements and source code components to confirm that the implemented code matches the design and that every code component is tested.
Test Case Traceability: Ensures that test cases trace back to requirements, and also tracks how test results, defects, and changes in the code or requirements affect the overall testing process.

Benefits:
Helps ensure complete test coverage by mapping tests to all requirements.
Supports test documentation and auditability, making it easier to track the progress of testing.
Provides valuable insights into testing gaps and missing test cases.

Example: A requirement for a login feature could be traced to specific test cases that test valid login attempts, invalid login attempts, password recovery, etc. This traceability ensures that the login feature has been tested in all required scenarios.

2.1.9 Coverage 覆盖范围

Definition: Coverage refers to the extent to which the software has been tested. In testing, coverage can measure how much of the application, its code, or its requirements have been exercised by the tests. It is often used as a metric to evaluate the thoroughness of testing.

Purpose: The main purpose of coverage is to quantify testing efforts and identify untested areas. High coverage helps ensure that more of the software's functionality has been tested, which in turn increases confidence in the software's quality.

Types of Coverage:
1， Requirement Coverage: This type of coverage measures whether each requirement has been tested by one or more test cases. It ensures that all business requirements are tested.
2， Code Coverage: This type of coverage focuses on measuring how much of the code has been executed during testing. Common metrics include:
3， Test Case Coverage: This type of coverage measures the percentage of test cases executed relative to the total number of test cases.

Benefits:
Provides a clear indication of the areas tested and the areas that might be untested or insufficiently tested.
Helps identify dead code (code that isn't executed) and areas where additional tests are needed.
Offers a quantitative measure of testing completeness, which is important for high-assurance systems (e.g., medical or financial applications).

Example: If a system has 100 lines of code, and 80 lines are executed during testing, then the line coverage would be 80%. However, if only 10 lines are tested, the coverage would be 10%.

2.2 测试计划

2.2.1 迭代软件开发

In each iteration of the development cycle, which typically includes requirements gathering, design, coding, testing, and feedback, both goals and conditions may evolve. Here's how they interact:

Goals Influenced by Conditions
Resource Constraints: Development teams often face resource constraints, such as limited time, personnel, or technical expertise. In this case, the team might have to adjust their goals to ensure that they can realistically achieve the desired outcome given the available resources. For example, a complex feature may need to be simplified or split across multiple iterations due to time limitations.

Technical Limitations: Sometimes, the technical feasibility of certain goals might be restricted by the current technology stack. A goal like implementing a sophisticated AI feature might be adjusted to a simpler version, or the team might decide to delay the use of certain technologies until they have the necessary resources or expertise.

Requirement Changes: As customer feedback or market demands evolve, the initial goals might change. For example, the product owner might change the priorities mid-iteration, requiring the team to focus on different features or rework the existing ones.

Conditions Influenced by Goals
Goals Driving Condition Improvement: Setting ambitious goals can push teams to improve conditions. For example, if a feature requires better system performance, the team might advocate for upgrading the infrastructure, improving the codebase, or adding more team members to meet the objectives.

Gradual Improvement of Conditions: Over time, as goals are met in each iteration, the development team can improve the conditions. For instance, lessons learned from previous iterations might lead to better code quality, enhanced testing practices, or more effective team collaboration. These incremental improvements can make the next iteration smoother and more efficient.

2.2.2 需求阶段QA的核心工作

In the Requirements Analysis phase, the main tasks of the QA team are to answer two key questions:

1. Does every feature have an appropriate testing method (testing technique)?
   This question is crucial to ensure that each functional requirement is covered by a suitable testing approach. The QA team needs to:

Define testing strategies: Identify the testing techniques that will be used to validate each feature. These strategies might include unit testing, integration testing, functional testing, performance testing, etc., based on the nature of the feature.

Create a Test Case Inventory: A high-level outline or inventory of test cases that covers all the features of the software. This inventory should list all the test cases needed for each feature and align them with the corresponding requirements.

Determine Test Methods: Specify the particular testing techniques used for each feature, such as:

Manual Testing: For user-centric, exploratory, or complex functionality.
Automated Testing: For repetitive or regression testing.
Performance Testing: For load or stress testing specific system functionalities.
Test Data Sources: Identify where the test data will come from, which is critical for both manual and automated tests. The data could be:

Synthetic data: Data created specifically for testing purposes.
Real-world data: Data from existing systems, anonymized for privacy concerns.
Simulated data: Data that mimics real-world scenarios.
Ensuring that each function has a defined testing method prevents gaps in test coverage and ensures thorough validation of all features.

2. Are all team members on the same baseline in terms of requirements understanding?
   This question is aimed at aligning the whole team, including developers, testers, business analysts, and product owners, on a shared understanding of the requirements. The QA team needs to:

Facilitate requirement clarification: Ensure that all team members fully understand the requirements. This might involve conducting requirement review sessions or discussions with stakeholders.

Document the baseline: Ensure that the requirements are documented clearly and consistently, often in the form of user stories, use cases, or requirement specifications. This serves as the baseline for the entire team.

Review and validate requirements: Perform a thorough review of the requirements to identify any ambiguities, inconsistencies, or gaps that could affect the testing or development process.

Agreement on Acceptance Criteria: Make sure that the acceptance criteria (the conditions under which the software will be accepted as "done") are clear, precise, and understood by everyone. This ensures that testing will be focused on the right areas.

By answering these two questions, the QA team ensures that:

Testing efforts are well-planned, comprehensive, and aligned with the project's needs.
There is a shared understanding of the requirements, which minimizes the risk of misunderstandings and defects during development and testing.

2.2.3 全功能提交团队

A Full-Function Commit Team is a type of software development team structure typically used in Agile development and continuous integration environments. This team consists of members from various functional areas who are capable of working independently at different stages to ensure that all features and functionalities of the software are implemented, tested, and deployed, and continuously integrated into the main codebase.

Key Features:
Cross-Functional Team Composition:
A full-function commit team is typically composed of developers, testers (QA), product managers, business analysts, and operations engineers, among other roles.
Team members are responsible not only for their specific areas but also for collaborating to ensure that every phase of the project, from requirement analysis to product release, progresses smoothly.
For example, developers write the code, testers handle the testing, operations manage deployment and infrastructure support, and product managers and business analysts define and validate requirements.

Agile and Continuous Delivery:
The team adopts Agile methodologies, such as Scrum or Kanban, for iterative development with short development cycles.
Continuous integration and continuous delivery (CI/CD) practices are followed, meaning that each team member is responsible for code commits, testing, building, and deploying the application.
Self-Organizing and High Collaboration:

Members of the team work collaboratively and have a high level of autonomy in solving problems.
Team members can independently address technical challenges while driving the project forward together.
The team is flexible enough to adapt quickly to changing requirements or priorities and to make necessary feature commits to keep the project on track.

Feature Completeness:
Each cycle (such as every sprint or iteration) delivers a complete, functional commit. This means that it’s not just about the development of a feature but also about testing and preparing that feature for deployment.
Each commit is a complete, integrated feature, tested, and ready for deployment, ensuring that the software is always in a deployable state.

Advantages:
Increased Efficiency: With cross-functional roles, communication costs are minimized, and team members can collaborate directly, avoiding delays due to inter-department communication.
Faster Feedback: As development and testing occur within the same team, issues can be detected and resolved more quickly.
Reduced Dependencies: Tight collaboration within the team allows for parallel work in development, testing, and deployment, reducing dependencies on external departments or resources.

Example:
In a typical full-function commit team, the roles might include:
Developers: Responsible for implementing the functional requirements through coding.
QA Testers: Write and execute test cases, ensuring that features meet requirements and are free of defects.
Product Managers: Define requirements and ensure that the features align with market or customer needs.
Operations Engineers: Ensure smooth deployment of the code, manage environment configurations, and monitor system performance.

2.2.4 EC & AC

Exit Criteria (EC)
Exit Criteria are the conditions or set of requirements that must be met before a testing phase, process, or project can be considered complete. They are used to determine when testing or validation activities can be concluded and when the project can progress to the next phase or be delivered.

Key Aspects of Exit Criteria:
Completion of Test Execution: The tests defined in the test plan (e.g., functional, performance, regression, etc.) have been executed.
Test Coverage: All critical and high-priority requirements or features have been tested.
Defect Closure: All defects identified during the testing phase have been addressed, either fixed or accepted with risk mitigation.
Pass Percentage: A certain percentage of tests must pass (often 90% or higher), or specific thresholds related to test pass/fail rates must be met.
Approval from Stakeholders: Relevant stakeholders (e.g., product managers, business owners) have reviewed and approved the testing results.

Exit Criteria help to determine if the current phase is complete and if the product is ready for the next steps, such as deployment, release, or further stages in the lifecycle.

Example:
All planned test cases have been executed.
95% of high-priority defects have been fixed or mitigated.
Stakeholders have approved the test results, and the product is ready for production deployment.

Admission Criteria (AC)
Admission Criteria refer to the conditions that must be met before a testing phase or process can begin. These criteria ensure that the required inputs, resources, and preconditions are in place to start the testing activities effectively.

Key Aspects of Admission Criteria:
Test Planning: The test plan, test cases, and test data should be ready and approved.
Environment Setup: The test environment (hardware, software, network, etc.) must be set up and configured correctly to support testing.
Requirement Stability: The requirements or specifications for the features being tested should be stable and clear.
Availability of Resources: Required resources, such as testing tools, team members, and access to systems or environments, should be available.
No Critical Blockers: There should be no critical issues or blockers preventing the start of testing.

Admission Criteria help ensure that the testing process is initiated in a controlled and well-prepared manner, with all necessary conditions in place.

Example:
The test environment has been successfully set up and validated.
All test cases have been written and reviewed.
All required software and hardware resources are available.
No critical issues are preventing the start of testing.

2.2.5 软件度量的指标

Here is the translation and explanation of the software quality factors and their corresponding metrics:

1. **Lines of Code (LOC)**
   Translation: Lines of Code (LOC)
   Explanation: LOC is a fundamental unit for measuring the size of a software application. It counts the total number of lines of code in a program, including both executable code and comments. While it can give an indication of software size and complexity, it does not account for code quality or functionality, which makes it an imperfect metric for measuring software quality on its own.
2. **Function Points (FP)**
   Translation: Function Points (FP)
   Explanation: Function Points are used to evaluate the size of software based on its functional requirements from the user's perspective. Unlike LOC, which measures the physical size of the code, function points focus on the software's functionality. They are calculated based on factors like the number of inputs, outputs, user interactions, files, and interfaces. Function Points are particularly useful for measuring software size in a more abstract, user-oriented way, which is independent of the programming language or technology used.
3. **Defect Density**
   Translation: Defect Density
   Explanation: Defect density is a measure of the number of defects (bugs or issues) found per unit of software size. It is often calculated as the number of defects per thousand lines of code (KLOC) or per function point. This metric is important because it helps identify the quality of the codebase. A higher defect density may indicate that the software is error-prone, while a lower defect density suggests a more stable and well-tested system.
4. **Reliability**
   Translation: Reliability
   Explanation: Reliability refers to the software's ability to function without failure under specified conditions for a specified period of time. It is often measured using metrics like Mean Time To Failure (MTTF), which calculates the average time between failures in a system.

A highly reliable system ensures minimal downtime and consistent performance, which is essential for critical applications like banking, healthcare, and aerospace.

5. **Maintainability**
   Translation: Maintainability
   Explanation: Maintainability is the ease with which software can be modified to correct defects, fulfill new requirements, or adapt to changes in the environment. High maintainability is crucial for long-term success because software systems often need updates or adjustments. Metrics for maintainability may include code modularity, complexity, and how well-documented the code is. Tools like Cyclomatic Complexity can also help assess maintainability.
6. **Testability**
   Translation: Testability
   Explanation: Testability is the degree to which a software system can be tested effectively. This includes factors such as the availability of test cases, ease of automation, and how well the software can be verified against its specifications. Testability can be improved by designing software with clear interfaces, consistent behavior, and appropriate levels of abstraction. Automation coverage is a common metric for testability.
7. **Schedule Variance**
   Translation: Schedule Variance
   Explanation: Schedule variance measures the difference between the planned project timeline and the actual progress. It is a critical metric for tracking project performance. Positive schedule variance indicates that the project is ahead of schedule, while negative variance suggests that the project is delayed. Keeping schedule variance under control is essential for meeting deadlines and avoiding scope creep.
8. **Critical Path Completion Rate**
   Translation: Critical Path Completion Rate
   Explanation: This metric tracks the percentage of critical tasks completed on time. The critical path in project management refers to the sequence of tasks that determine the minimum project duration. If tasks on the critical path are delayed, the entire project will be delayed. Monitoring the completion rate helps ensure that essential tasks are completed according to the project schedule.
9. **Cost Variance**
   Translation: Cost Variance
   Explanation: Cost variance measures the difference between the actual cost incurred in a project and the budgeted cost. Positive cost variance indicates that the project is under budget, while negative variance shows overspending. Managing cost variance is crucial for maintaining financial control over a project and ensuring its profitability.
10. **Return on Investment (ROI)**
    Translation: Return on Investment (ROI)
    Explanation: ROI is a measure of the profitability of an investment. It compares the benefits (returns) gained from a software project to the costs incurred during its development. A high ROI indicates that the project is delivering good value for the money spent, while a low ROI suggests that the project may not be financially viable or sustainable.
11. **Productivity**
    Translation: Productivity
    Explanation: Productivity is often measured as the amount of output (usually in terms of function points) produced per unit of input (typically measured in developer hours or work hours). A higher productivity rate indicates that the team is producing more functionality with

fewer resources, suggesting better efficiency and performance. Productivity can be influenced by factors like team experience, tool support, and project management practices.

12. **Review Efficiency**
    Translation: Review Efficiency
    Explanation: Review efficiency is the ratio of the number of issues or defects found during the review process to the total number of items checked. A higher ratio indicates that the review process is effective at identifying potential problems early, which helps improve software quality. Efficient reviews can significantly reduce the number of defects in the final product.
13. **Team Morale**
    Translation: Team Morale
    Explanation: Team morale is a measure of the overall motivation, satisfaction, and enthusiasm of the team members. High morale often leads to better productivity, collaboration, and quality. Team morale can be gauged through surveys, interviews, or informal feedback sessions. Factors like workload, leadership, work-life balance, and recognition play a significant role in influencing morale.
14. **Communication Effectiveness**
    Translation: Communication Effectiveness
    Explanation: Communication effectiveness refers to how accurately and promptly information is exchanged within the team and between stakeholders. Effective communication ensures that all team members are aligned, reducing misunderstandings and errors. It also plays a critical role in decision-making, problem-solving, and maintaining project focus.
15. **User Feedback**
    Translation: User Feedback
    Explanation: User feedback is direct input from users about their experience with the software. It includes comments, suggestions, complaints, and feature requests. Regular collection of user feedback helps in improving the product by aligning it with user needs and expectations. This feedback is crucial for continuous improvement and ensuring that the software meets the target audience's requirements.
16. **Net Promoter Score (NPS)**
    Translation: Net Promoter Score (NPS)
    Explanation: NPS is a metric used to measure customer loyalty. It is calculated based on the likelihood of users recommending the software to others. Users are typically asked how likely they are to recommend the product on a scale from 0 to 10. Based on their score, users are categorized as promoters, passives, or detractors. A high NPS indicates strong user loyalty and satisfaction, which can lead to increased customer retention and word-of-mouth promotion.

These metrics are essential for evaluating and improving software quality, ensuring that projects meet both functional and non-functional requirements while delivering value to users and stakeholders.

2.2.6 圈复杂度

Cyclomatic Complexity is a software metric used to measure the complexity of a program’s control flow. It quantifies the number of linearly independent paths through the source code, providing an indication of how complex a program is and how difficult it might be to understand, test, or maintain. The term was introduced by Thomas J. McCabe in 1976.

Explanation:

Cyclomatic complexity is based on the structure of the control flow graph (CFG) of a program. In this graph:

- Nodes represent the statements or instructions in the program.
- Edges represent the flow of control between those statements.

The cyclomatic complexity (V) can be calculated using the following formula:

V = E - N + 2P

- E is the number of edges in the control flow graph.
- N is the number of nodes in the control flow graph.
- P is the number of connected components (typically 1 for a single program).

Interpretation:

- Low Complexity (1-10): The program is relatively simple and easy to understand and test.
- Medium Complexity (11-20): The program is moderately complex, and there may be some difficulty in understanding or testing it.
- High Complexity (21 or more): The program is highly complex, which might indicate that it's difficult to understand, maintain, and test. It could be prone to errors and bugs.

Benefits of Cyclomatic Complexity:

1. Predictability: It can be used to predict how hard the program will be to test and maintain. A high cyclomatic complexity usually correlates with a greater number of bugs and increased maintenance difficulty.
2. Test Coverage: It helps determine the minimum number of test cases needed to cover all independent paths in the program.
3. Code Quality: Reducing cyclomatic complexity can improve the readability and maintainability of the code, making it easier for developers to understand and modify.

2.2.7 缺陷-瑞利分布

In software testing, the daily number of newly discovered defects often follows a **Rayleigh distribution** in terms of its temporal variation. This phenomenon can be explained from a statistical perspective, as the Rayleigh distribution is commonly used to describe processes influenced by multiple factors, such as the gradual discovery of defects during software testing.

Overview of Rayleigh Distribution:
The Rayleigh distribution is a continuous probability distribution often used to describe the amplitude of random variables with symmetric and normal distributions, particularly when multiple independent, identically distributed random variables are involved. In the context of software defects, the Rayleigh distribution can model the defect discovery process, especially when the rate of discovery increases over time.

Why the Daily Defect Discovery Follows a Rayleigh Distribution:
Early Stages – Low Discovery Rate: In the initial phases of testing, the number of defects discovered is low because testers may not fully understand the system's complexity or the system has not been sufficiently tested.
Middle Stages – Increasing Discovery Rate: As testing progresses, testers gain more experience and a better understanding of the system, and the test coverage expands. This results in an accelerated rate of defect discovery.

Later Stages – Stabilization or Decrease: After most of the obvious defects have been found and addressed, testers may start discovering marginal or hard-to-detect defects. The rate of defect discovery slows down and stabilizes, eventually reaching a plateau or decreasing.

Role of Rayleigh Distribution in Defect Discovery:
Modeling and Prediction: The Rayleigh distribution effectively describes the dynamic process of defect discovery and can be used to predict the number of defects that might be discovered in the near future. This helps test managers make decisions on resource allocation and planning adjustments during testing.
Quantifying Testing Progress: By analyzing the distribution of defect discoveries, the QA team can assess the progress of testing and the completeness of test coverage. The Rayleigh distribution shape reflects the natural process: fewer defects are found early on, followed by a more efficient discovery process, and eventually a saturation point is reached.

2.3 测试用例

2.3.1 概述 Overview

Exhaustively testing all possible inputs to any nontrivial software component is generally impossible due to the enormous number of variations. One approach to create a test suite with high coverage and a low number of variations is known as combinatorial testing. One common strategy, known as pairwise, tests a set of variations where every possible pair of parameters appears at least once. This method can be extended to use higher orders of combinations (3-wise, 4-wise, etc) for increased coverage at the expense of a larger test suite.

Software test techniques exist to reduce the number of tests to be run whilst still providing sufficient coverage of the system under test

There is no single technique or approach that is completely effective in software testing, so by increasing the diversity of methods used in testing and considering different perspectives, we are more likely to be successful in both exposing potential issues as well as increasing the effectiveness of our testing.

The value or effectiveness of using any test technique ultimately depends on the individual test's system and domain knowledge and ability to apply the applicable technique in the correct situation.

2.3.2 理论 Theory

The single fault assumption in reliability theory states that failures are rarely the result of the simultaneous occurrence of two (or more) faults.The primary purpose of ECP and BVA testing is to expose single fault.

在实际系统中，输入域往往不是单一的，有多个输入域，每个输入域到底该如何组合取值呢。等价类和边界值分析都基于一个十分重要的假设，这个假设在可靠性理论中被称作“单故障”假设，即失效问题通常不会由两个（或多个）故障同时引发。因此，构造测试用例的方式是：仅让一个变量取不同的值，而让其他所有变量都取正常值.

2.3.3 分类原则 Partitioning Principle

If an input condition specifies a range of values, this defines three classes:
within range: a valid input equivalence class
too large: an invalid input equivalence class
too small: a invalid input equivalence class

If an input condition specifies an enumerated set of values (e.g. “car”, “truck”, etc.):
One valid equivalence class for each value in the group.
One invalid equivalence class: all values not in the enumerated set (i.e. everything else).

If an input condition specifies a “must be”, situation (e.g. “first character of the identifier must be a letter”), this leads to:
One valid equivalence class (e.g. the first character is a letter).
One invalid equivalence class (e.g. the first character is not a letter).

Finally, if there is any reason to believe that elements in an equivalence class are not handled in an identical manner by the implementation software, split the equivalence class into smaller classes.

2.3.4 BVA Boundary Value Analysis 边界值分析

This BVA technique believes and extends the concept that the density of defect is more towards the boundaries. This is done due to the following reasons：
a) Usually the programmers are not able to decide whether they have to use <= operator or < operator when trying to make comparisons.
b) Different terminating conditions of For-loops, While loops and Repeat loops may cause defects to move around the boundary conditions.
c) The requirements themselves may not be clearly understood, especially around the boundaries, thus causing even the correctly coded program to not perform the correct way.

2.3.5 Dependency Between Parameters

Inadequate when there are dependencies between the variables ("Equivalence Class" Testing is better)

Bounds may not be always obvious (e.g. an upper bound for the triangle problem)
Bounds may not be appropriate (e.g. for Boolean values; “Decision Table” Testing is better)

No bounds in cases of enumerated variables (e.g. in bank transactions “deposit”, “withdrawal”, “query”)

2.4 测试执行

2.4.1 Entry criteria & exit criteria

进入标准和退出标准

Simply, Entry and Exit criteria are the conditions which when satisfies; the test team starts (enters) the testing process and stops (exits) it.
Entry/Exit Criteria depends on time, milestones and project deadlines.

Entry criteria examples：
All test hardware platforms must have been successfully installed configured and Functioning properly.
All standard software tools including testing tools must have been successfully installed and functioning properly.
All documentation and design of the architecture must be made available.
All personnel involved in the system test effort must be trained in tools to be used during testing process.
A separate QA environment (with its own web server database and Application server instance) must be available.
Proper test data is available.

Exit criteria examples：
Application must provide the required services.
Ensure all application documentation has been completed and is up to date.
100 of all Priority 1 and priority 2 bugs must be resolved.

2.4.2 Suspension & Resumption criteria

Suspension and Resumption criteria are the conditions which when satisfies; the test team temporarily suspends (suspension) the testing process and resumes (resumption) it.

Suspension/Resumption Criteria in a Software Testing depends on working of build and Whether the build is working properly or not.

Suspension criteria examples：

hardware / software not available at the time indicated in the project schedule
the build contains many serious defects which seriously prevent or limit testing progress
Assigned test resources are not available when needed by the test team

Resumption criteria examples：
If testing is suspended, resumption will only occur when the problem(s) that caused the suspension have been resolved. When a critical defect is the cause of the suspension, the “FIX” must be verified by the testing team before testing is resumed

2.4.3 “successful” test execution

What is a “successful” test execution?

- A successful execution is one which tells us the result of a test. Whether this result is the one we want or not.
- Honesty is the simplest and the most important.

Planning is essential for success

- The execution itself, needs to be scheduled and environment variables scheduled. The wider Test Process planning plays a part. Analysis and requirements are the foundations on which success is built. Assembling a sound mechanism for recording results and outputs. It is from these, that debugging of defects will be based.
- Examples in test execution plan：
  Business Days Setup for running the test cases in case its batch application.
  Priority should be defined for the test cases.
  Sequencing of the test cases for the execution.
  Schedule, Resource allocation.
  Backup plan if testing of module is suspended.

Control risk

- Priority for test case
- High priority test first

Test environment

- A huge range of platforms, operating system and other interface combinations are available. Each combination will have an effect on the SUT. This means that the test case suite may have to be ran many times. However we can never test in every possible environment. A scope will need establishing for which combination to test in. A huge range of platforms, operating system and other interface combinations are available. Each combination will have an effect on the SUT.

Automated or manual

Log results

- Once the test suite has been exercised we need an audit trail of the execution. We need to log all the relevant details. Again this process can be automated or manually on paper.
- Likely information to log includes
  Time date
  Who ran test
  Environment - Platforms, operating systems
  Overall results - Percentage of tests passed and other metrics.
- At the individual test case level useful information includes:
  Requirement tested
  Input
  Output
  Predicted outcome
  Actual outcome
  Pass/Fail - according to pass/fail criteria.

2.4.4 effective defect principles

How to write an effective defect

Principles

- Condense - Say it clearly but briefly
- Accurate - Is it a defect or could it be user error, misunderstanding, etc.?
- Neutralize - Just the facts. No zingers. No humor. No emotion.
- Precise -Explicitly, what is the problem?
- Re-create - What are the essentials in triggering/re-creating this problem? (environment, steps, conditions)
- Impact - What is the impact to the customer? What is the impact to test? Sell the defect.
- Evidence - What documentation will prove the existence of the error?

2.5 测试提交

2.5.1 提交什么

In Scrum software development, at the end of each Sprint, a Potentially Shippable Product Increment (PSPI) is created. This increment is the most important outcome of the Sprint and must meet specific criteria to ensure it's potentially ready for release. Here's what the Potentially Shippable Product Increment typically includes:

1. **Completed and Fully Developed Features**
   The Product Increment must include all the user stories or features planned for that Sprint. These user stories should be completed, meaning all the tasks within them have been finished and accepted by the Product Owner (PO).
   The features are considered "done" according to the team's Definition of Done (DoD). This

includes not only the development of the feature but also any associated tasks like testing, documentation, and code reviews.

2. **Fully Tested**
   The increment must pass all types of testing defined by the team, such as unit tests, integration tests, and user acceptance tests (UAT).
   Testing must include both manual testing (if applicable) and automated tests to ensure that the new code works as expected and does not introduce regressions.
   The tests should be completed with no major defects outstanding. If defects are found, they should be documented and ideally resolved within the same Sprint.
3. **Integrated with Existing System**
   The new features or improvements are integrated with the existing system. This ensures that the new code works well with the previously developed features and doesn't break anything in the existing product.
   Any conflicts with previous functionality should be addressed and resolved, meaning that the product should be functionally stable at the end of the Sprint.
4. **Buildable and Deployable**
   The Product Increment must be in a buildable state (meaning no major build or integration issues) and deployable to a production-like environment. It should be possible to deploy the product increment at any time without requiring major changes.
   This implies that all code, configurations, and dependencies are packaged in a way that the development team or operations team can push it to production if needed.
5. **Meets Product Owner's Acceptance Criteria**
   The increment must meet the acceptance criteria defined by the Product Owner for each user story or feature. The PO will review the functionality and verify that it aligns with the business needs and the goals of the Sprint.
   The increment should also be aligned with the overall product vision and roadmap, providing value to the end users or stakeholders.
6. **Documentation (if applicable)**
   While Scrum emphasizes working software over comprehensive documentation, any necessary documentation (like API documentation, deployment guides, etc.) should be updated as part of the increment.
   The goal is to ensure that the product can be maintained, operated, and enhanced by both the development team and future teams working on it.
7. **Ready for Release (if necessary)**
   The term Potentially Shippable means that the increment is ready to be released to the customer or end users, if the decision is made to do so. However, this doesn't necessarily mean it will be released at the end of every Sprint.
   In some cases, the product may not be released immediately, but it must be in a state where it could be deployed to production without further work required.

Key Characteristics of a Potentially Shippable Product Increment:

- Quality: It must meet the agreed-upon quality standards, defined by the team's Definition of Done (DoD).
- Functional: The product must be functional, meaning all the committed features and user stories must work as expected.
- Stable: The increment should not introduce instability to the product.

- Value-Driven: The increment must deliver value to the customer, as defined in the Sprint goal and user stories.

2.5.2 Burn Down Chart 燃尽图

Burn Down Chart in Software Development Process and Its Application

A Burn Down Chart is a visual tool used in Agile project management, particularly in Scrum, to track the progress of a project or sprint. It is a graphical representation that shows the remaining work (in terms of effort, time, or tasks) over time, helping teams monitor their progress towards completing a specific goal. The chart typically tracks the completion of work items, such as user stories or tasks, during a sprint or iteration.

Structure of a Burn Down Chart
A typical Burn Down Chart has the following components:

1. X-Axis (Horizontal): Represents time, usually broken down into days or iterations (e.g., sprint days).
2. Y-Axis (Vertical): Represents the amount of work remaining, which could be in the form of story points, hours, or tasks.
3. Ideal Line: A straight line that starts at the total amount of work (at the beginning of the sprint) and gradually decreases to zero as time progresses, indicating the ideal progress toward completing all tasks.
4. Actual Line: A line showing the actual remaining work at the end of each day or iteration. This line fluctuates based on the progress of the team in completing the tasks.

How It Works

- At the beginning of the sprint or project, all tasks or user stories are estimated and plotted on the Y-axis.
- As work progresses, the Actual Line is updated daily to reflect the remaining work. Ideally, the actual work completed should follow the trajectory of the Ideal Line.
- The chart visually displays whether the team is ahead, on track, or falling behind in terms of completing the required work.

Applications of the Burn Down Chart

1. Tracking Progress:
   The primary application of the Burn Down Chart is to track the progress of the team in completing the sprint or project. It gives a quick visual representation of how much work is remaining versus how much has been completed. If the Actual Line is close to the Ideal Line, it indicates that the project is on track.
2. Identifying Potential Issues Early:
   A Burn Down Chart can quickly reveal whether a team is falling behind schedule. For example, if the Actual Line is above the Ideal Line, it means the team is not completing tasks as expected. This early warning allows the team to take corrective actions, such as re-prioritizing tasks or allocating additional resources.
3. Forecasting Completion:
   By tracking the burn rate (the rate at which work is completed), the Burn Down Chart can help forecast whether the team will complete the work by the end of the sprint or iteration. If the Actual Line is not on target, adjustments can be made, such as adding more resources or reducing the scope of the sprint.

4. Improving Team Performance:
   The Burn Down Chart serves as a feedback tool. It helps the team see their progress and the pace at which they are completing work. Over time, teams can use this data to improve their estimation skills, productivity, and sprint planning.
5. Transparency and Communication:
   Burn Down Charts promote transparency in Agile teams. It is a useful tool for stakeholders and team members alike, as it allows everyone to see the current state of the project. It improves communication and collaboration by providing a shared understanding of progress.
6. Retrospective Discussions:
   During sprint retrospectives, teams can review the Burn Down Chart to reflect on the sprint’s progress and any issues encountered. The chart can help identify patterns in the team's workflow, such as whether tasks were under-estimated or external factors impeded progress.
7. Handling Scope Changes:
   If scope changes occur (e.g., new features are added or removed), the Burn Down Chart can be updated to reflect the new work remaining. This helps the team assess the impact of these changes on the sprint’s timeline.

Benefits of Using a Burn Down Chart:

- Simple and Visual: It provides a simple and quick visual reference to understand the project’s progress.
- Helps with Focus: The chart highlights how much work is left, helping the team focus on completing tasks and avoiding distractions.
- Motivational: Seeing the line steadily go down as work is completed can serve as a motivational tool for the team.
- Facilitates Agile Practices: It aligns well with Agile practices by encouraging incremental progress and continuous improvement.

Limitations of Burn Down Charts:

- Does Not Show Quality: While it tracks the amount of work remaining, it doesn’t account for the quality of the work. A project could be "burning down" tasks quickly, but if the quality isn’t high, this could be problematic in the long run.
- May Lead to Oversimplification: It is easy to misinterpret the chart if it is not used correctly. For example, a flat or erratic Actual Line could be seen as a problem, but it may be due to external factors like dependencies or delays that are not captured in the chart.
- Does Not Capture Dependencies or Blockers: The Burn Down Chart primarily tracks work completed but does not show blockers or dependencies between tasks, which could also impact progress.

3. 第三部分

3.1 unit test

3.1.1 Why we need unit test

The value of unit testing can be summarized in the following four aspects:

1. **Bring the Test as Early as Possible**
   Unit testing encourages writing and running test cases early in the development process. The value of this approach lies in:

- Early Detection of Defects: By writing unit tests early in the development phase, developers can validate functionality as they implement it, allowing defects to be detected early, thus reducing the cost of later fixes.
- Improved Development Efficiency: With early feedback, developers can quickly identify issues, improving development efficiency and avoiding the complexity of debugging and fixing problems later in the process.

2. **Make the White Box Test Possible**
   Unit testing facilitates white-box testing, which tests the internal logic of a program, particularly the details of code structure and implementation:

- Transparency and Control: Unit testing allows developers to closely examine each part of the code, ensuring the correctness of the internal implementation. It helps to cover all code paths and test branches and conditions.
- Validating Design and Implementation: It not only verifies that the functionality works as expected but also helps validate the design and ensures that the implementation aligns with the intended logic, thereby assisting developers in improving and optimizing the code structure.

3. **Lower the Threshold of Test as Possible**
   Unit testing lowers the barrier to testing, making it easier for developers to write and run tests:

- Automation and Ease of Use: Unit tests are typically automated and can be executed easily, allowing developers to run tests and view results without requiring complex configurations or testing tools. This simplifies the testing process.
- Encouraging Developer Participation: Because unit tests are automated and provide rapid feedback, developers are more likely to engage in testing, which increases the overall focus on quality control within the team.

4. **Make the Integration as Specific as Possible**
   Unit testing helps achieve more specific and granular integration testing:

- Early Verification of Integration Points: Through unit testing, developers can validate the functionality of individual modules before integrating them, ensuring that no new errors are introduced during integration.
- Efficient Collaboration Between Modules: Unit testing ensures that each module is independently verified, ensuring that interfaces and interactions between modules are correct. This leads to smoother and more stable system integration.

3.1.2 Classic pattern of unit test

1. **Unit Testing in General Development**
   In traditional development processes, unit tests are often written after the code has been developed. Developers typically write tests after implementing features or completing modules.
   Characteristics:

- Timing of Writing Tests: Tests are written after code is implemented, typically once the feature or module is complete.
- Test Coverage: Test cases are written to validate critical functionality that the developer deems necessary. It often involves validating whether the code works as expected.

- Purpose of Testing: The primary goal is to find defects by verifying that the code behaves as expected and to catch bugs.
- Separation of Testing and Development: Testing is a separate phase that comes after development, potentially leading to delays between development and testing.

2. **Unit Testing in TDD (Test-Driven Development)**
   In TDD, unit tests are an integral part of the development process. Developers write test cases before writing the code, starting with a failing test and then writing code to make it pass.
   Characteristics:

- Timing of Writing Tests: Tests are written before the implementation code. Developers first write a failing test and then write code to make the test pass.
- Test-Driven Development: The key principle is "test first, code second." Testing is not an afterthought but a driving force in the development process. Each piece of functionality is developed around one or more tests.
- Purpose of Testing: The goal is not only to validate functionality but to drive design. The tests help developers design clean and simple interfaces and architectures.
- Rapid Feedback and Iteration: Since tests are written immediately after code, developers receive immediate feedback, enabling faster bug fixes and reducing defects.
- High Test Coverage: TDD encourages incremental development, with each change accompanied by a new test. This typically leads to higher code test coverage.

3. **Unit Testing in XP (Extreme Programming)**
   XP emphasizes continuous integration, rapid feedback, simplicity, and collaboration. Unit testing is central to the development process in XP, and tests are written and executed continuously throughout the development cycle.
   Characteristics:

- Timing of Writing Tests: Like TDD, unit tests are written continuously throughout development. Tests are often written before code and executed frequently.
- Purpose of Testing: The goal of testing in XP is to ensure that each part of the system is working correctly and remains functional as the system evolves. Tests are used not only to find bugs but also to ensure high code quality in a collaborative environment.
- Frequent Refactoring: XP encourages frequent refactoring of code. Unit tests provide a safety net, allowing developers to refactor without fear of breaking functionality.
- Collective Code Ownership: In XP, anyone on the team can modify any part of the codebase. Unit tests ensure that changes do not introduce new defects, and they help maintain the stability of the code.
- Continuous Integration and Automation: XP emphasizes the use of continuous integration and automated unit testing, ensuring that all features are always tested as they are integrated into the codebase.

3.1.3 How to perform the unit test

Unit test best practice:

1. Ensure the independence of tests, with each test method focusing on a single function.
2. Adopt white-box testing methods for fine-grained control and use data-driven approaches to drive different test cases.

3. Use metadata for tagging to ensure the readability and maintainability of test code.
4. Use stubs to isolate the external environment.
5. Integrate unit tests into the CI/CD pipeline to achieve automation.
6. Use assertions and implement comprehensive test logs.

3.1.4 Difference between Unit test and white box test

Connections

- Both unit testing and white-box testing focus the code level quality and require a deep understanding of the software's internal structure and logic.
- Unit test provides the operational skeleton frame for white-box testing .
- Both types of testing are generally conducted by developers who have access to the source code and understand how the software should work internally.

Differences

- Unit testing focuses on the smallest testable parts of an application, while white-box testing can encompass broader sections of the code, including multiple units and their interactions.
- Unit tests are based on functional requirements and specifications, whereas white-box tests are based on the implementation details.
- Unit testing isolates the unit from other parts of the system, often using mocks and stubs; white-box testing may not isolate units and can include interaction between units.

4. 白盒测试三种测试方法

语句覆盖（statement coverage）

> 程序中的每个语句至少能被执行一次

判定/**分支覆盖**（branch coverage）

> 满足判定覆盖一定满足语句覆盖 BC≥ SC
>
> 判定里面是个黑盒

条件覆盖（condition coverage）

> 一个判定往往覆盖多个条件，此时判定里面是个白盒
>
> （理论上）条件覆盖是一个比判定覆盖更强的标准 CC “≥” BC

条件判定覆盖（CDC condition decision coverage）

- 结合了条件覆盖和判定覆盖

多重条件覆盖（MCC multiple condition coverage）

-

> MCC ＞ CDC

等价类：以简洁为准

4.1 名词解释

粒度 granularity

白盒/灰盒/黑盒测试

单元测试 unit test：独立

集成测试 integration test：模块间集成，不是系统间

回归测试

冒烟测试smoke test：和准入准出有关

EC(Exit criteria), AC(Accept criteria)

unit test 测试方法，白盒测试

为什么需要 unit test?

单元测试和白盒测试的区别?

质量保证与软件测试之间的区别?

5. 历年卷 40%雷同

5.1 2023-2024

5.1.1 Q1 解释概念

1. Beta Testing
2. Alpha Testing
3. Smoke Test
4. Regression Test

5.1.2 Q2 简答题

1. QA 在需求分析时主要干什么
2. What factors determine the priority of a bug.
3. 3个 key elements of a testing plan.
4. QA 和 testing 区别
5. V model, waterfall 的 unit test 对应哪个 stage，这个 stage 的 inputs 和 outputs
6. McCall 质量三角形中，correctness, efficiency, reliability 分别对应哪种 test

5.1.3 Q3 Unit test 和 white box test 的区别与联系

5.1.4 Q4 为什么要尽早地开展 test

5.1.5 Q5 设计等价类

一盏灯，T (on / off)，C (color，有三种颜色），A (brightness，三档亮度)

画一个XY坐标轴的图，和设计测试用例（填表）

ID	等价类	initial state	operation	expect result
1				
2				
…				

Table 3

5.1.6 Q6 我们一般认为 statement coverage < branch coverage < condition coverage，给出一个反例否定这一关系。

> 以下通过一个简单的代码示例来给出反例否定“statement coverage < branch coverage < condition coverage”这一通常认为的关系：
>
> 考虑如下一段简单的 `if - else` 结构的代码（使用Python语言示例，其他语言原理类似）：
> `python def check_value(x):     if x > 5 and x < 10:         return "In range"     else:         return "Out of range"`
>
> 分析
>
> 1. **语句覆盖（Statement Coverage）**： 要达到语句覆盖，只需要执行代码中的每一条可执行语句至少一次就可以了。对于上述代码，我们只需要选择一个合适的 `x` 值，比如 `x = 7`，这样代码中 `if` 语句体里面的 `return "In range"` 会被执行到，同时 `else` 分支里的 `return "Out of range"` 不会执行，但整体的代码语句都能覆盖到（因为代码中就这两条不同的返回语句属于可执行语句部分），所以语句覆盖的标准就达到了。
> 2. 2. **分支覆盖（Branch Coverage）**： 分支覆盖要求程序中每一个分支（比如 `if` 语句的 `true` 分支和 `false` 分支）都至少被执行一次。同样对于上述代码，我们可以选择 `x = 7` 执行 `if` 语句的 `true` 分支（使得进入 `if` 语句体里面执行），再选择 `x = 3` 来执行 `if` 语句的 `false` 分支（进入 `else` 语句体执行），这样就达成了分支覆盖。
> 3. **条件覆盖（Condition Coverage）**： 条件覆盖要求使得判定中的每一个条件的所有可能结果至少出现一次。在我们的代码里，判定条件是 `x > 5 and x < 10`，这里有两个条件分别是 `x > 5` 和 `x < 10` 。理论上要达到条件覆盖，需要考虑这些条件分别取 `true` 和 `false` 的各种组合情况。 但是存在这样一种情况，如果我们只进行语句覆盖时采用的 `x = 7` 这个测试用例，从条件覆盖角度看，对于条件 `x > 5` 结果是 `true`，对于条件 `x < 10` 结果也是 `true`，也就是部分条件的结果情况已经被覆盖到了，可此时并没有达成完整的条件覆盖（因为没有出现 `x > 5` 为 `false` 或者 `x < 10` 为 `false` 等其他组合情况），然而语句覆盖却已经达成了。 所以在这个例子中出现了语句覆盖达到了，但条件覆盖并没有完整达到的情况，打破了一般所认为的“statement coverage < branch coverage < condition coverage”的严格顺序关系，形成了反例。 综上所述，通过这样一个简单代码示例可以说明这种常规认知的关系并不绝对成立，是可以有反例情况出现的。