CodeArts Testplan

Best Practices

Issue   01
Date   2023-11-30

HUAWEI TECHNOLOGIES CO., LTD.

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to "Vul. Response Process". For details about the policy, see the following website:https://www.huawei.com/en/psirt/vul-response-process For enterprise customers who need to obtain vulnerability information, visit:https://securitybulletin.huawei.com/enterprise/en/security-advisory

Contents

1 DevOps Agile Test

This document describes how to build test capabilities tailored to specific product development stages and characteristics to address the challenges of agile and DevOps transformation.

Agile and DevOps

Agile and DevOps transformation is driven by business goals and customer needs. As market competition becomes increasingly fierce, the time window for innovation and monetization of new business models becomes shorter and shorter. As a result, more enterprises adopt the lean entrepreneurship mode. After capturing market requirements, enterprises have to shorten the time to market (TTM) of products and launch minimum viable products (MVPs) that satisfy customer requirements as quickly as possible.

Take Huawei as an example. Before 2008, Huawei still adopted the traditional delivery method for its projects. For example, after a project is initiated at the beginning of the year, all customer requirements, including user feedback, will be collected and ranked throughout the year. In the middle of the year, the product will be released to the customer. A patch will be released two months later, and a formal version will be released at the end of the year. Traditional projects have a low speed of version delivery but high quality requirements. If the customer finds a problem in the patch, the customer can only wait for another two months. If the customer does not accept the product in this period, project efforts are wasted. Therefore, the product quality must be strictly controlled.

Today, with products gradually developing towards agility, some R&D tool platforms have been migrated to the cloud, so test tools need to be transformed correspondingly. In the past, product delivery was conducted every half a year or every two months. After transformation, delivery was conducted every one month or even every two weeks. However, the transformation was not thorough, and there were still some problems with customers' delivery processes. Transformation to platform-based and service-oriented tools has changed business models fundamentally. After requirements are migrated to the cloud, customers can quickly get involved. Functions can be developed efficiently on cloud platforms. Products are frequently iterated according to customer requirements. This delivery mode shortens the delivery period from half a year to a couple of weeks, or even to a day or two. From the perspective of requirements, great changes have taken place. Basically, we have achieved step-by-step advancing and fast trial and error.

Test Debt

From waterfall to agile development and then to DevOps, the productivity of development, testing, and delivery is increasing, leaving some issues unresolved, which affects the ongoing improvement of test capabilities and value.

- Some companies attach more importance to development than testing, and limit the career of testers. In addition, manual testers are not familiar with programming, and developers do not pay enough attention to testing. Test teams often have heavy workloads, but are always understaffed.

- Testers sometimes do not fully understand customer requirements. Besides, the department silo between testing and development leads to information transparency issues, as well as insufficient communication and collaboration. Moreover, some companies overcompromise on quality for higher efficiency, and ignore the cultivation of agile culture and values.

- Some products are highly coupled and have poor testability. Testers rely too much on black box testing, and use inappropriate test policy and methods. The test environment deployment takes a long time and is frequently updated.

Focus of Testing: Quality of Service Value

Testing is a quality activity, which means its top priority is quality. Testing is also an engineering activity to obtain the maximum value with limited time, manpower, and resources. Although quality has multiple dimensions, it should have a focus: quality of service value, that is, quality of product value presented to customers. Testers should focus on the service value and determine the weights and priorities of quality in multiple dimensions, such as function, security, performance, usability, and compatibility. Testers should not test every aspect and related points for every testing project.

For example, for online payment functions, the focus of testing should be security; for online shopping functions, the focus should be usability; for large-scale flash sales and promotion activities, the focus should be performance. Therefore, testing must aim at the service value of products, determine product objectives, formulate key quality points and related test policy, and implement the policy in practice. Then, testers need to provide feedback on poor functions and test the improved functions again to check whether the overall quality meets the expected results.

Conventional Security and Elastic Security

Conventionally, we would try to find out and remove all insecure factors, which is an ideal way of testing. In actual work, however, it is impossible to identify all insecure factors in the entire system, which involves various aspects such as capabilities and architectures.

Therefore, elastic security is developed based on this. That is, the insecure factors are displayed as much as possible through scenario simulation. Based on this insecure scenario, a quick repair solution is provided to compensate for the insecure factors, which is not perceived by users. From the perspective of a product, both its commercial and quality goals can be achieved. This is called elastic security. Even if an error occurs, vulnerabilities can be quickly fixed or self-repaired in time to achieve the normal working purpose.

Shift-Left Testing and Shift-Right Testing

Shift-left testing is to push test actions toward the early stages of a project. For example, during behavior-driven development (BDD), test cases are designed based on scenario requirements to match the design. During consumer-driven contract (CDC) testing, services are coupled with each other. CDC can be used to decouple services from each other to prevent problems.

Shift-right testing is to push test actions toward the later stages of a project. Generally, tests are performed only before the software package release. Shift-right testing requires continuously testing from version release to production and online operation.

There are also some practices in these two aspects. For example, online dialing tests are performed to proactively monitor user behavior, quickly capture problems from the behavior track, and proactively push the problems to related owners for them to pay attention to and solve the problems. Therefore, the online process can be fed back to developers through some test methods to let them know the overall performance of the current product. Then the developers can quickly respond to the product.

Test Strategies in Different Stages of Product Development

Is it necessary to build the all automated testing capabilities as soon as a team is built? The following describes how to build automated testing capabilities from the perspective of the software maturity period.

In the initial stage of software exploration, the product is in an uncertain state. The front-end style and overall layout, as well as the back-end APIs, change frequently. Because the life cycle of automated testing cases is short, it is not cost-effective to create some automated testing cases. In this period, the product features can be controlled and only a few tests are performed. Therefore, manual tests can be performed instead of automated testing. In this way, the product can quickly identify errors and users can use the product.

In the product expansion stage when users recognize the product, the number of users and requirements will increase. In this case, automation must be considered because the full verification cost of each iteration in this stage increases and the delivery speed also increases. It is impossible to perform all manual tests during each round of go-live. In this case, automated testing cases are required for old modules.

At the product extraction stage, product requirements and benefits have reached the saturation stage. In this case, product requirements must be strictly controlled, and the responsibilities of automated testing cases must be guarded. No change is allowed to introduce extra risks or major feature changes, which may cause attacks on mature users.

Impact of Team Scale on Test Construction

If the team has fewer than five members and the team is in the exploration stage, the quality activities can be limited to the self-organization stage of the test. In this case, only some basic test management activities are performed, defects are managed, and regression tests are performed. In this stage, a test management process and a mechanism are established, and automated testing are not involved.

With the further expansion of the project, the number of team members gradually increases to 5 to 10. At this time, the test workload suddenly increases, and dedicated testers may be involved. The testers will talk with developers, convert the requirements into automated testing cases, establish continuous integration, and gradually evolve some test methods. At this stage, some automation attempts have been made.

As the team size increases, if one person cannot handle the workload, more testers will be recruited and a dedicated test team will be set up. This team will shift from automated testing to test automation and involve more management work. During the management process, we will interconnect some products, including developing dedicated tools to implement the overall automation capability, not only automation execution.

After the preceding evolution cycles, the test team has a lot of test automation experience. In this case, the cloud-oriented transformation can be performed. Currently, many teams are performing DevOps transformation. The most concerned aspect is to set up a DevOps full-function team. What are these people doing before transformation? What is the original test team with 10 to 15 members? In this stage, the team needs to transform special test capabilities into service-oriented capabilities. Test specialists will provide training at the early team stage, including test engineering construction, early test case formulation, standard template development, and special capability training for non-functional tests. All teams review the test process, including the test policy, test plans, and test cases. Then, they check the improvements in the process of the entire team. Last but not least, the special test teams are transformed from all aspects to servitization to achieve automation transformation.

Automated Testing and Test Automation

This part introduces the concept of test automation.

Test automation aims to reduce manual tests and operations. Test automation includes not only automated testing, but also all other activities that can reduce manpower input, such as automated test environment creation, deployment of tested systems, monitoring, and data analysis. In many cases, automated testing is only the execution part of the test. For example, some manual test methods during test execution are changed to automated testing. However, test automation is not only an execution part, but also includes obtaining and generating test data from the environment, executing automated testing, and generating results. If there is any problem, it will be automatically pushed to related personnel and the corresponding organization will solve the problem. Test reports are automatically generated so that testers can directly obtain the test results.

2 Tailoring a Test Plan

This chapter describes how to determine test objectives and scope, formulate test policy and plans, set up test teams, and prepare test tools and environments.

At the beginning, the team needs to formulate a test plan to guide the test activities of the testers in the entire test period. A test plan describes the objectives, objects, scope, policy, activities, methods, resources, and progress of the test. It also determines the test items, features, tasks, executors, and possible risks. It can effectively prevent the risks and ensure the smooth implementation of the plan.

Significance of Making a Test Plan

- Ensures that test activities are carried out based on test objectives and serve specific test objectives.
- Determines the test features, requirement list, and test scope of the test object.
- Selects test policy and methods suitable for the team's technical capabilities and tool portfolio.
- Identifies possible risk factors during test activities as early as possible so that you can resolve them in time.
- Properly estimates the test workload, personnel, and resource requirements, and prepares the plan for each test item.
- Helps testers break down test activities and tasks and orchestrate personal work plans.
- Guides test execution activities, and corrects and remedies execution deviations in time.
- Provides related documents to report and communicate with stakeholders.

Time for Making a Test Plan

Test activities include test plan development, test design, and test execution. Test plan development ranks first and is carried out in the early stages of the test period. The test plan may be carried out at multiple time points based on the development mode and team organization mode used by different projects. For example:

- A system test plan can be made after the test department receives the test demand for a customer-oriented mobile app before its first version is released.

- In an internal software project, a test plan can be made before the product enters the development stage when the product initiation requirements are confirmed, the architect has designed the product architecture solution and design solution, and the developers begin to make a development plan.

- For commercial products that require routine special security tests, a test plan can be made after the security test department evaluates feature changes of new products based on historical security test plans and test reports.

- Different types of test plans can be made by experts of function tests, performance tests, security tests, and more after the test team leader makes the overall test plan.

It is recommended that the test plan be started as early as possible to guide and standardize the product quality activities in the early stages and improve the product testability. The testers should guide the quality elements in the product architecture and design from the perspective of attackers, which complies with the idea of shift-left testing.

However, in the early stages of the product, the granularity of the test plan is coarser. There is a lack of test cases at the executable level and available test environments. Therefore, the test plan needs to be refined as the project progresses. The test plan is not constant. As the test project is carried out, the test plan is gradually detailed and contains more and more information. During the refinement and improvement of the test plan, the initial test objectives, scope, design, and policy should be reviewed.

Test Plan Makers

- Owners of test projects and test teams
- Owners of security, performance, and reliability tests
- Experienced test engineers and test architects

Test Plan Content

According to ISO and IEEE standards related to test documents, the content to be included in a test plan can vary from project to project. The content depends on the project and team scale. The test plans can be simplified for small teams.

- Objectives

  Describe why a test is performed and what test objectives need to be achieved. The test objectives are the beginning of a test plan. The test needs to focus on the service value of the product. An overall product test plan should integrate the function, security, performance, usability, compatibility, and scalability into the test objectives based on the service attributes of the product. For example, financial products have high requirements on security.

- Test scope

  Describe the name, version, features, requirements, environments, and test items of the tested system (test objects).

- Test policy

  Specify the test types, scenarios, and methods, and strategically describe how to perform the test.

- Test solution

  Describe the test solution, such as the integration procedure and sequence, test procedure and sequence, test methods, test tools, and test case design and execution methods.

- Test environments

  Describe the names, specifications, quantities, versions, and accounts of the hardware, software, and test tools required for the test, as well as the management policy for preparing, reserving, restoring, and releasing the test environments.

- Testers

  Describe the number of testers, work division, and responsibilities, such as test architects, test development engineers, performance test engineers, and test environment management personnel.

- Test schedule

  Describe the planned start time and end time of a test, overall test schedule, and key phased progress check points. The test schedule should be combined with the development plan. The constraints and dependencies between tasks and resources such as the test solution, environments, and personnel must be considered.

- Test entry conditions

  Specify the entry conditions for starting a test. For example, the functions specified in the product specifications have been implemented, and the basic process and entry test cases have been passed. This prevents the test plan from being affected by the lack of basic test conditions.

- Test release standards and deliverables

  Specify the conditions to be met for completing a test, the criteria for passing or failing the test, and the deliverables to be generated after the test, for example, a test report whose content is specified.

- Risks

  Analyze the potential risks in the current project operation and the measures to mitigate and resolve the risks. Examples of risks: human resource availability risks, personnel skill and domain knowledge risks, and development-to-test time risks.

Test Plan Review

After a test plan is created, key stakeholders, such as the project manager, test manager, product manager, architect, and O&M manager, should be invited to review the correctness, comprehensiveness, and feasibility of the test plan.

3 Typical Test Design Techniques

Test design plays an important role in test activities. Test objects, scenarios, types, and environments are analyzed based on the test plan. Proper test techniques are selected based on the test policy to design test cases. Test design techniques include scenario analysis, equivalence partitioning, boundary value analysis, cause and effect graphing, decision table testing, orthogonal array testing, and more. Flexible use of test design techniques can help reduce redundant test cases, improve test coverage, maintainability, and reuse of test cases, reduce invalid test execution workload, and improve test effects.

Equivalence Partitioning

Equivalence partitioning is a testing technique that divides the input data of a system into equivalence classes of equivalent data. In each equivalence class, selecting all input data for testing is equivalent to selecting only one piece of input data for testing. If one piece of input data fails to detect any system error, all other input data in the equivalence class cannot detect any system error. Equivalence partitioning is used to select one piece of data from each equivalence class as the test input to improve the coverage of test scenarios and reduce invalid test workload.
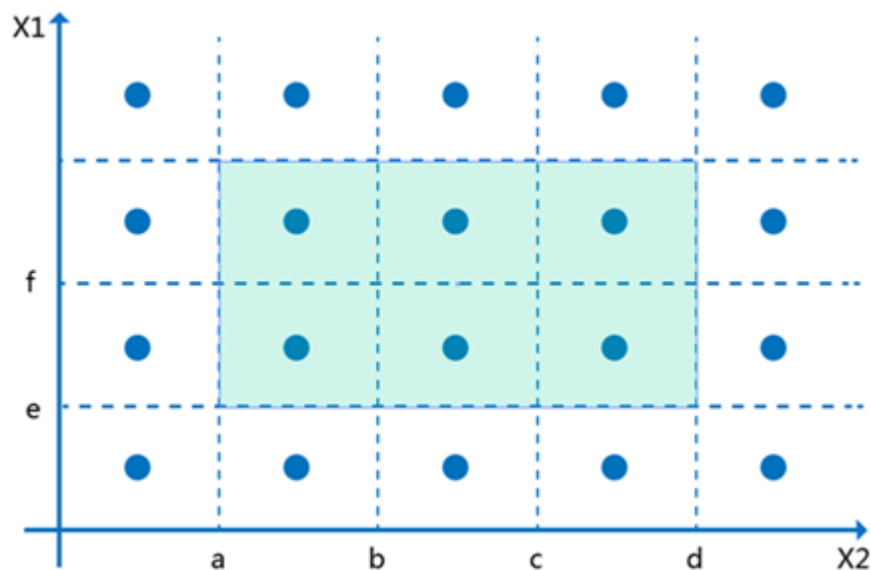
Equivalence partitioning includes valid equivalence classes and invalid equivalence classes. A valid equivalence class summarizes a set of reasonable and meaningful input data, that is, a regular path test input. It can be used to check whether the program implements the functions and performance specified in the specification. In contrast, an invalid equivalence class is a set of unreasonable or meaningless input data.

- Example 1: If the input condition is a Boolean value, a valid equivalence class and an invalid equivalence class can be obtained.

  Input condition: whether to back up data.

  Valid equivalence class: yes (TRUE).

  Invalid equivalence class: no (FALSE).

- Example 2: If the input condition specifies the range of the input value, a valid equivalence class and an invalid equivalence class can be obtained.

  Input condition: a number greater than 1 and less than 3.

  Valid equivalence class: 2.

  Invalid equivalence class: 0 and 4.

- Example 3: If the rules that the input data must comply with are specified, a valid equivalence class that complies with all rules and several invalid equivalence classes that violate the rules from different perspectives can be obtained.

  Input condition: a positive integer.

  Valid equivalence class: 1.

  Invalid equivalence class: 0, –10, 10.1...

The equivalence partitioning technique focuses on the proper division of input values. When designing test cases, consider both types of equivalence classes. Ensure that not only reasonable data can be received, but also input in unexpected scenarios can be processed. When the equivalence partitioning technique is used for test design, list the possible inputs of the features to be analyzed, and divide and classify the equivalence classes. Each equivalence class after analysis is regarded as a test case. Tools such as tables or mind maps can be used to assist in equivalence partitioning. Equivalence partitioning is often used in conjunction with other techniques, such as boundary value analysis.

- For example, if the input data is months, which are integers ranging from 1 to 12, then 12 valid equivalence classes and one invalid equivalence class can be obtained, as shown in the following table.

Input Condition	Valid Equivalence Class	No.	Invalid Equivalence Class	No.
Month	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	0001	13	1001

The following test cases can be created based on the equivalence classes.

No.	Test Case	Equivalence Class No.
0001	Input of a correct month	0001
0002	Input of an incorrect month	1001

Boundary Value Analysis

According to experience, a large number of errors occur on the boundary of the input or output range. Boundary value analysis is to select test data at and near the boundary of the divided equivalence class area and to design test cases. The following methods and principles can be followed:

- When the input condition specifies the value range

  Select the boundary values and the values that are just beyond the boundaries as the test input. For example, if the input value is an integer ranging from 0 to 100, design test cases for values 0 and 100, and also for –1, 1, 99, and 101.

- When the input condition specifies the number of values

  Select the maximum number, minimum number, minimum number – 1, and maximum number + 1 as the test input. For example, if the number of attachments to be uploaded ranges from 1 to 10, design test cases for values 1, 10, 0, and 11.

- When the input and output are ordered sets

  Select the first and last elements of the set as the test input. For example, if the input is an ordered array and the array value ranges from 1 to 7, indicating Monday to Sunday, select 1 and 7 as the test input.

- If a group of values (n) of the input data are specified and the program needs to process each input value separately, n valid equivalence classes and one invalid equivalence class can be established.

- Analyze the specifications and find out other possible boundary conditions.
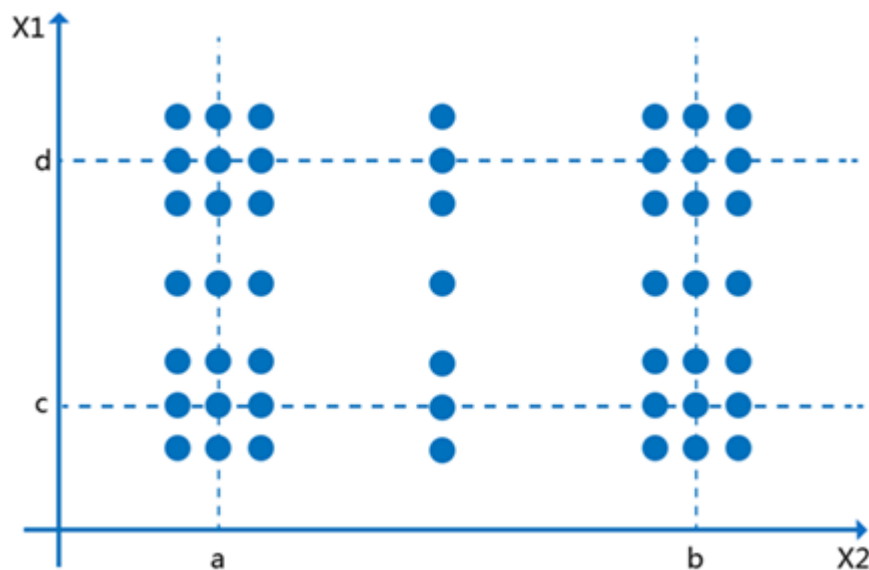
After the preceding steps are performed, a large number of test items may be generated, which may be repeated and can be combined. Boundary value analysis is often used together with equivalence partitioning. Generally, boundary values are selected in equivalence classes.

4 Testing Pyramid and Continuous Automated Testing

In agile and DevOps development modes, products must be able to be released at any time. This document describes how to use the testing pyramid and continuous integration and continuous delivery (CI/CD) automated testing to implement efficient test feedback and ensure the quality of products released at any time.

Testing Pyramid

The test automation pyramid was first proposed by Mike Cohn in 2009 in his book *Succeeding with Agile: Software Development Using Scrum*. It was first proposed as a three-layer pyramid, which consists of the GUI tests, service tests, and unit tests from top to bottom. With the development of agile tests, there are some variants of the test automation pyramid. In practice, the service tests can also be regarded as API tests.

The triangle structure indicates that the recommended investment proportion for automation at each layer increases from top to bottom.
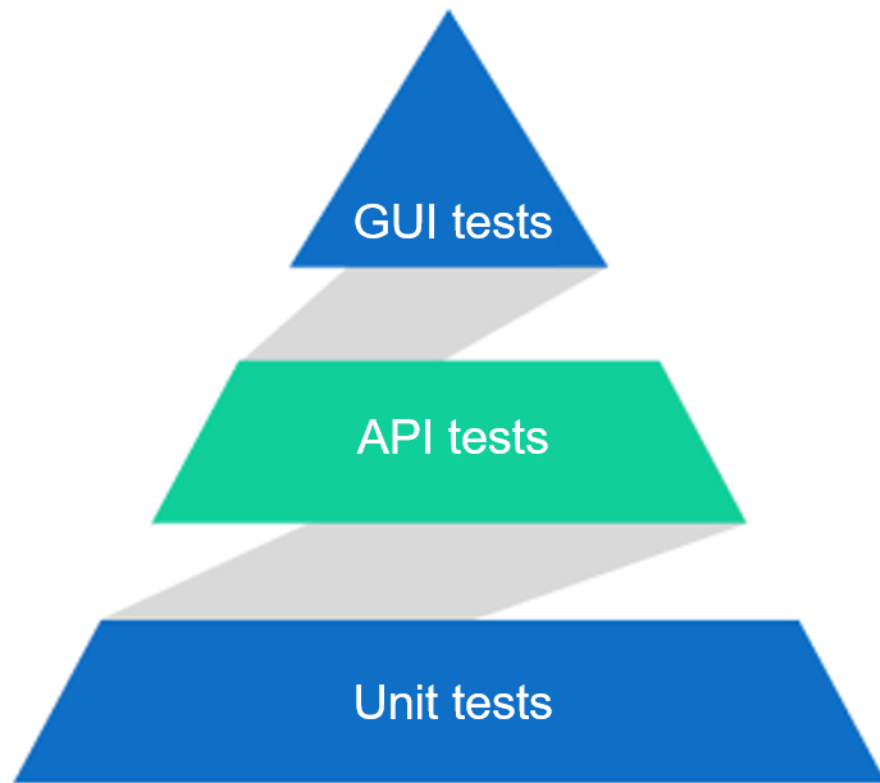
Martin Fowler has a comment on the testing pyramid, "Most importantly such tests are very brittle. An enhancement to the system can easily end up breaking lots of such tests, which then have to be re-recorded. You can reduce this problem by abandoning record-playback tools, but that makes the tests harder to write. Even with good practices on writing them, end-to-end tests are more prone to non-determinism problems, which can undermine trust in them. In short, tests that run end-to-end through the UI are: brittle, expensive to write, and time consuming to run. So the pyramid argues that you should do much more automated testing through unit tests than you should through traditional GUI based testing."

The test technologies involved in each layer of the testing pyramid have their own advantages and limitations. The upper-layer GUI tests are brittle (instability), time-consuming (execution efficiency), and the distance between the problem symptom location (UI) and the root cause location (code) is too long. The testing pyramid focuses on the test quality instead of quantity. It is recommended that the bottom-layer test investment be increased.

- A higher layer leads to lower running efficiency, which slows down the build-feedback cycle of continuous integration.
- A higher layer leads to higher development complexity, which slows down delivery progress when the team capabilities are limited.
- End-to-end testing is more likely to bring uncertainty in test results.

● A lower layer leads to stronger unit isolation, which makes it easier to locate and analyze problems.

In principle, the unit test needs to be undertaken by the developers. In many teams, the developers are insufficient, so they have to focus on function implementation first. In addition, due to a lack of investment and experience in the unit test, many teams do not perform the unit test or just do it reactively. Some have proposed the anti-patterns of the pyramid structure—the **Ice-Cream Cone** and the **Cupcake**.

The Ice-Cream Cone pattern, proposed by Alister Scott in 2012, inverts the ratio of GUI tests and unit tests in the testing pyramid to form an inverted pyramid. In addition, a large number of manual tests are added above GUI tests. Although anti-patterns are not recommended, they are indeed used frequently. At the early stages, most teams did not perform automated testing and completely relied on manual tests. Later, the teams started from function test automation and generated some GUI automated testing cases. This way of building automated testing from outside to inside leads to such an anti-pattern. This pattern has some problems in terms of test efficiency and test case maintainability. However, it is an inevitable path for many teams to build test automation capabilities. When problems are accumulated to a certain level, this pattern needs to gradually evolve to the testing pyramid.
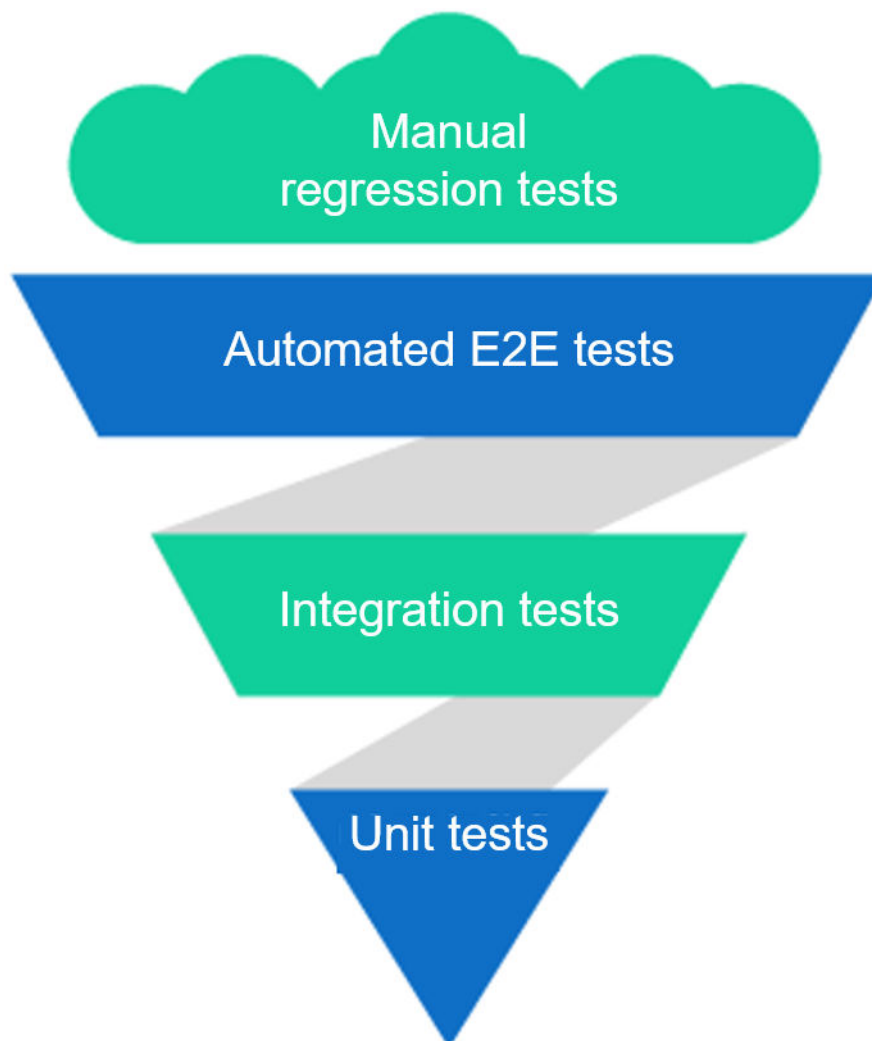
Continuous Automated Testing

Continuous automated testing is to run automated testing during CI/CD and quickly report failures. It is first derived from the idea that developers obtain quick feedback through unit tests in the development environment. Continuous automated testing gradually matures with the development of CI/CD. Today, developers are required to update product and fix online problems more and more quickly. If manual tests are still used frequently or development and testing are completely separated, it is difficult to ensure that the test quality assurance activities are completed in a short time window. Therefore, automated testing needs to be embedded in the CI/CD process to ensure the quality of deliverables.
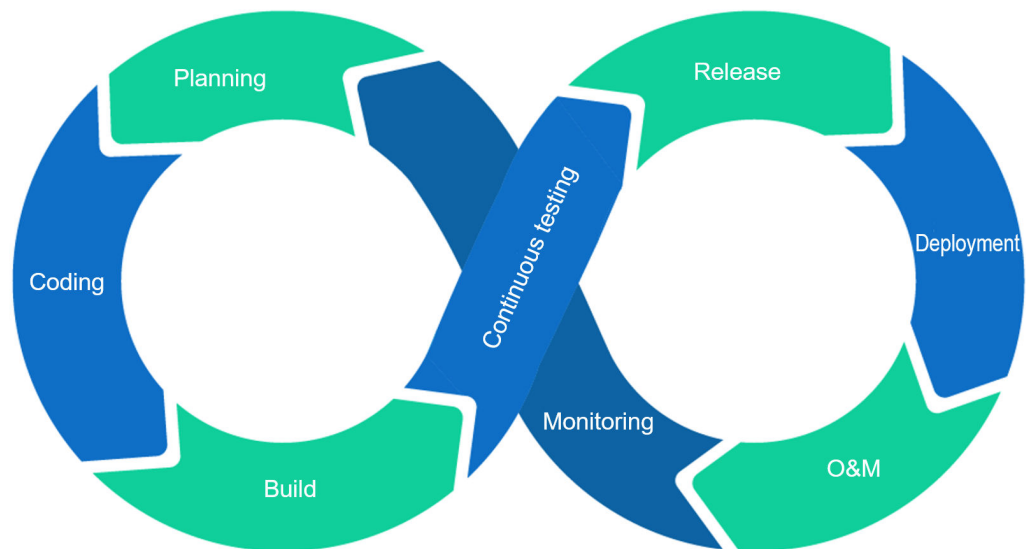
Continuous testing means that test activities are incorporated into the continuous integration, feedback, and improvement cycle. Continuous testing runs through the entire software delivery cycle. Continuous testing advocates early, frequent, and automated testing.

"Continuous" is reflected in the whole process of evolving deliverables from small granularity to finished software in the agile and DevOps processes, from white-box code tests to component module tests, API tests, E2E function tests, and even online tests in the production environment after delivery. Each stage maps the layers of the testing pyramid from bottom to top. The lower-layer tests are performed in the early stages, and the upper-layer tests are performed in the later stages. This is similar to each stage of the automobile manufacturing pipeline. After the assembly in each stage is complete, necessary checks are performed before the next stage starts. During software DevOps development, the pipeline carries the assembly, check, and test processes.
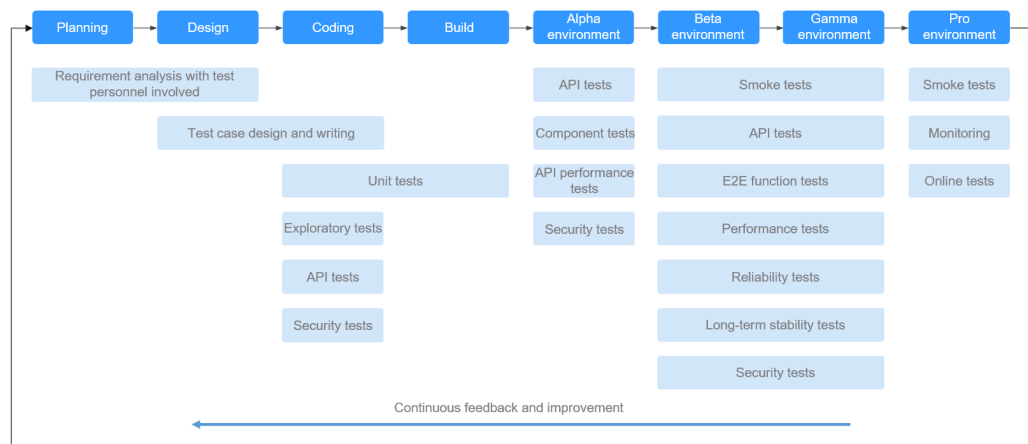
- Shift-left testing and shift-right testing

  During continuous testing, test activities can go beyond the time, role, and department restrictions of traditional tests, and are developed into coherent and continuous quality assurance activities.

  Shift-left testing emphasizes that the test activities are carried out as early as possible. The testers participate in the activities in the early stages of a software project. The related test cases are defined before the function development to detect quality problems in advance. The developers participate in the tests.

  Shift-right testing emphasizes that test monitoring is performed in the production environment, and that user feedback is obtained in real time to continuously improve user satisfaction and product quality.

- Test automation

  To achieve fast flow and quick feedback, continuous testing needs to be automated to improve efficiency. Therefore, automated unit tests, API tests, and E2E tests are embedded into the DevOps pipeline. Automated testing improves test feedback efficiency and reduces errors caused by human factors. Test automation not only automates the execution of test case scripts, but also implements all other activities that can reduce manpower input, such as automatic environment creation, deployment, monitoring, and data analysis.

5 Defect Handling Process and Precautions

Product defect handling is not just about ticket submission by testers and defect fixing by developers. It requires clear, comprehensive, and traceable defect tickets, as well as a process that covers defect detection, reproduction, confirmation, rectification, self-verification, regression tests, and closure.

Collaboration Between Development and Testers

During product testing, the testers record defect tickets, transfer them to the developers for handling, and track the defect handling and closure. Defect tickets are an important information carrier for development and testers to communicate with each other. They may encounter the following problems:

- The developers complain that the defect description submitted by the testers is not detailed. For example, the reproduction procedure and the software version of the problem are not provided, which increases the communication cost.
- The developers do not reproduce the problem mentioned in the defect ticket in the local development environment and directly transfer the defect to the testers.
- After a defect is rectified, the developers do not notify the testers. As a result, the defect is not reviewed in time.
- After a defect is found during testing, the functions of related peripheral functions are not tested. Therefore, potential problems are ignored. Besides, the developers do not conduct related research.
- Developers disagree with the severity of defects marked by testers.

Defect Handling Process

The development and testers are the owners of the software product quality. They have the same goal and willingness in terms of product quality assurance. The only difference lies in the work activities they are engaged in. The development and implementation of the defect handling process should aim to achieve mutual trust and efficient collaboration, instead of being used as an excuse for inaction and a trigger for contradictions. The following describes a complete defect handling process, which can be used as a reference in operations.
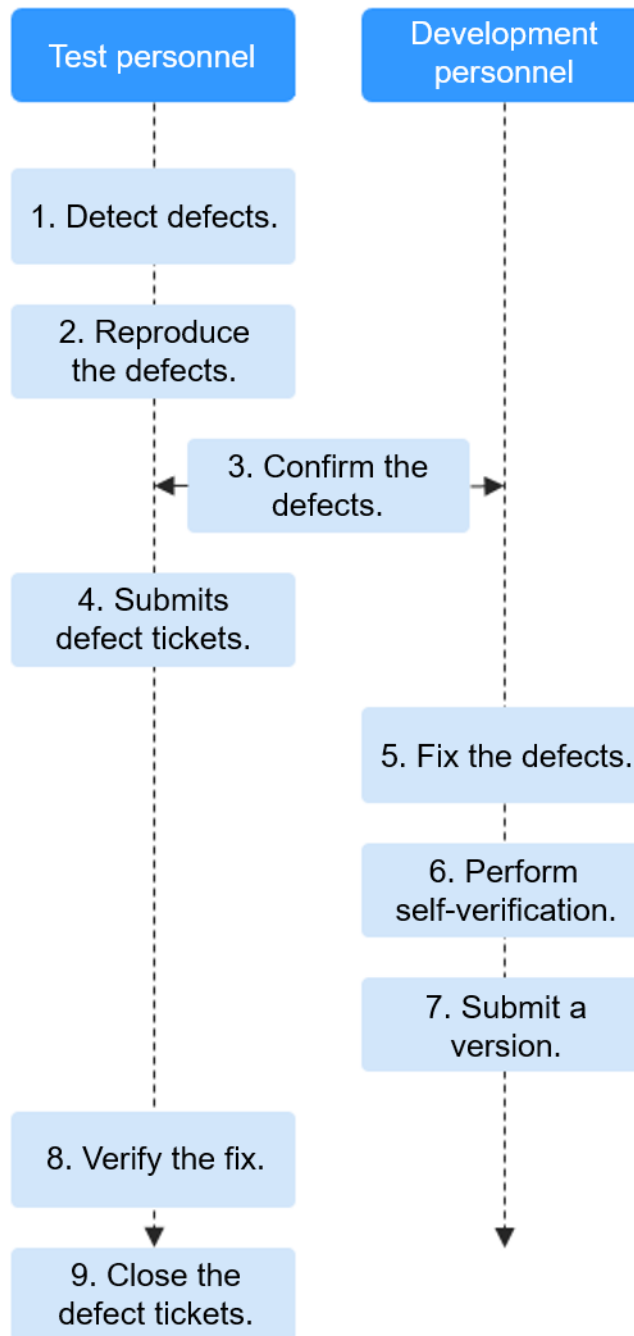
1. Detect defects.

   In software development and testing, as code and modules are overlaid and invoked layer by layer, an underlying defect may cause multiple problems. Testers should not jump to conclusions about the problem that is found at first and its causes. Instead, a logical and systematic analysis is required.

   – First, exploratory analysis is required to check whether there are other problems in addition to the first problem, and whether these problems exist at the same time or have certain dependencies and sequence. Therefore, more test procedures are required.

For example, if an IT system cannot be logged in using a mobile number and verification code, the testers need to analyze other login modes such as using a mobile number and password; logging in to another system using a mobile number and verification code; logging in to the system using an app, a browser, or another device; using a mobile number of another carrier.

– Second, infer the causes of the problem and verify the causes. Do not regard test procedures as the causes. Instead, analyze the data changes caused by the test procedures as the causes to check whether similar problems may occur in other scenarios. If the problem occurs occasionally, analyze the cause and contact developers to locate and demarcate the problem.

– Finally, sort out the conditions, operation procedures, and symptoms of the problem.

2. Reproduce the defects.

If the defects do not recur, it is difficult for developers to locate them. Generally, the testers are responsible for ensuring that the defects can be reproduced. If the defects occur occasionally and are difficult to be reproduced, it indicates that the root causes of the defects are deep. In this case, contact the developers for help. To reproduce the defects, the testers must:

– First, the testers who have detected the defects should change the input data or combination, and also change the test environment to reproduce the defects according to defect occurrence conditions and operation procedures.

– Second, other personnel (such as developers) should reproduce the defects based on the text and screenshot descriptions.

3. Confirm the defects.

Before submitting defect tickets, the testers should confirm with the developers, including whether the found problems are defects rather than optimization points or new requirements, whether the problems are repeated, whether the defects can be reproduced, whether the problem logs need to be supplemented, whether the defect severity is correct, whether the defects block testing, and when the defects will be resolved.

The time for development and testers to confirm defects is not limited. It is advised to confirm information as soon as possible from the time when a defect is found to the time when developers start to rectify the defect. The defect tickets can also be submitted to the module owner for unified confirmation and feedback.

4. Submit defect tickets.

The submitted defect tickets must be clear, comprehensive, manageable, and traceable. A dedicated defect management system is required for defect tickets. It is recommended that the defect management system be the same as the requirement and development task management systems to facilitate unified management and planning. Generally, a defect ticket contains the severity, type, problem description, root cause analysis, handling suggestions, test suggestions, associated test cases, environment information description, logs required for fault locating by developers, and screenshots.

5. Fix the defects.

After receiving defect tickets, the developers preliminarily analyze the workload and arrange the schedule. In addition to fixing the problems described in the defect ticket, the developers need to further test the scenarios that may be associated, perform in-depth tests, find possible in-depth problems, and solve the problems. After the defects are fixed, the developers need to describe the root causes, occurrence conditions, and solutions of the problems in the defect tickets. Some defect management systems can also associate defect tickets with code submission records to help track, collect, and trace defect tickets.

6. Perform self-verification.

   After problem rectification, the testers need to create an individual build and deploy it in the test environment in addition to verifying that the problems are rectified in the local development environment. Ensure that the test environment is the same as the environment used by the testers or the environment where the problems are found to eliminate environment differences. The self-verification is successful only after no problem is found in the further test in the test environment. In DevOps tools, individual-level pipelines can be used to automate the entire process of individual build packaging and environment deployment, improving self-verification efficiency.

7. Submit a version.

   After the code is fixed and reviewed, it should be released to the code branch of the target version.

8. Verify the fix.

   Testers deploy code branches that contain defect rectification in the test environment to check whether the problems are completely rectified. If the problems are not fixed or new problems are introduced during the fixing, record the problems in defect tickets and send them back to the developers for further analysis and fixing.

9. Close the defect tickets.

   The defect tickets can be closed only after the regression test verifies that the problems are resolved and no new problem is generated. Generally, a defect ticket can be closed only in three cases: normal closure of a problem, closure of a non-problem, and closure of a repeated problem. Some descriptions and pictures can be added to the defect tickets to record the version in which the problems have been resolved.

## Customizing a Defect Handling Process in CodeArts TestPlan

Step 1 Determine the defect status, for example, **New**, **Developing**, **Resolved**, **Testing**, **Rejected**, and **Closed**. These statuses have been preset in the CodeArts TestPlan bug template. You can also add new statuses.
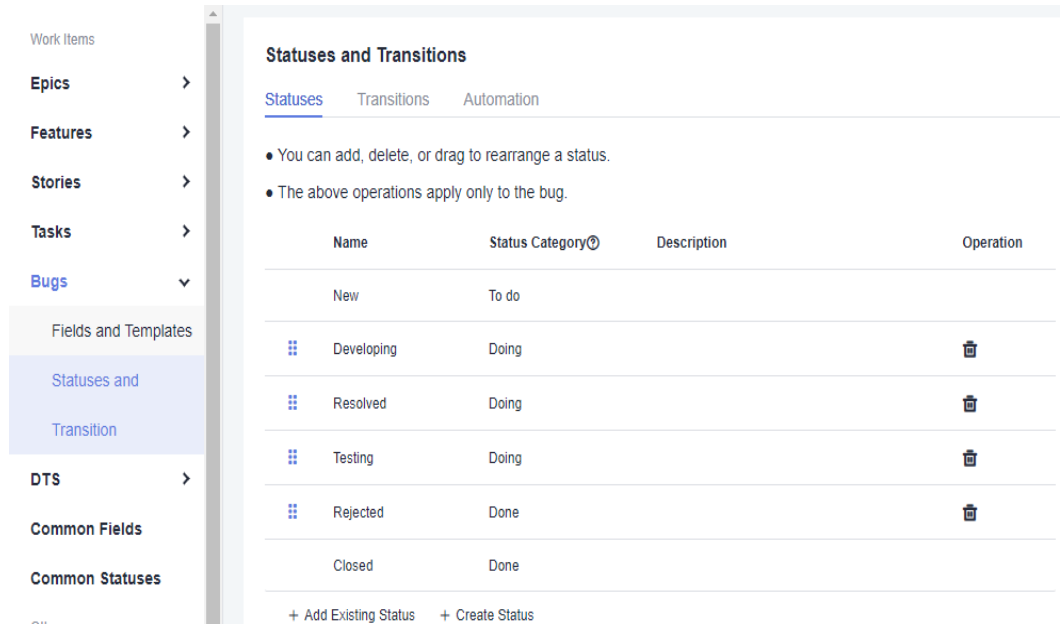
Step 2 Set the transition direction of the defect status.
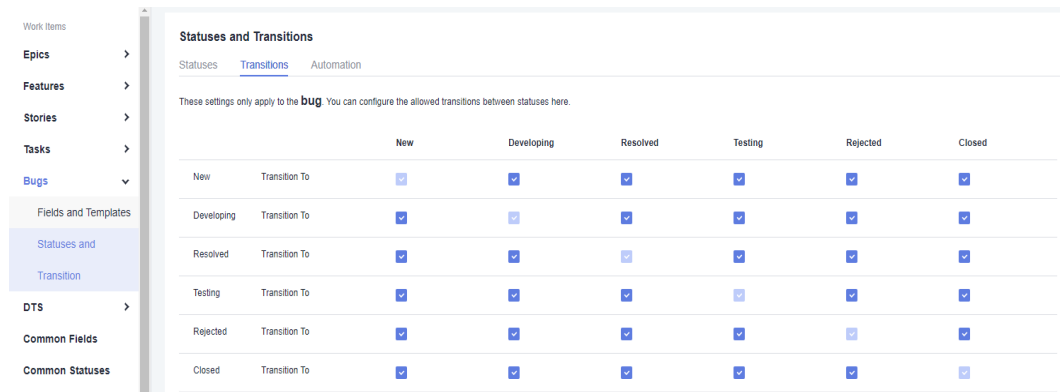
Step 3 Set defect fields and templates to instruct test and developers to fill in information.
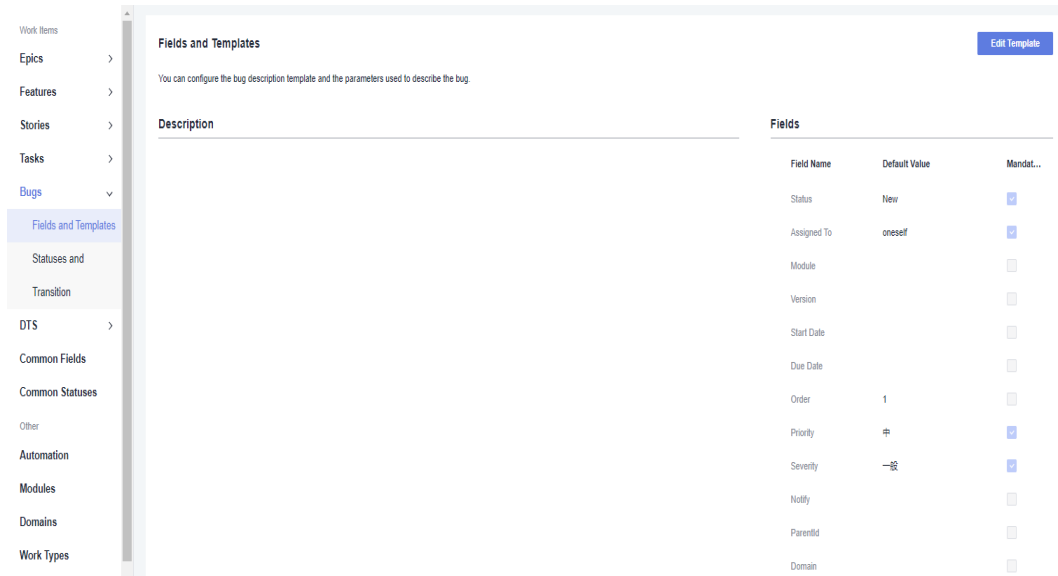
----End

6 Writing a Test Report

A test report summarizes the test process and result, that is, the completion of the test plan, analyzes findings, and provides the product quality basis for related personnel to make decisions on acceptance and delivery. Generally, a test report includes the test overview; test scope and function list; test policy and method description; test metric statistics and analysis; test risk analysis and disclosure; quality evaluation and release suggestions.

Test Overview

A test report briefs the test activity, clarifies target audience, reference test standards, test background and requirements, summarizes test object analysis, test requirements, test content, and test process, and draws test conclusions.

The test conclusions must be prepared based on the focuses of the target audience of the report, that is, the target stakeholders. Product managers should focus on risk disclosure and product quality conclusions. Test managers should focus on test costs and test outputs. Developers should focus on defect results and product quality information.

Test Scope and Function List

- A test report describes the functions, application scenarios, benefits, and functions of test objects.
- A test report describes the test scope specified in the test plan, including the name, version, features, requirements, environment, and test items of the tested system (test objects).

Test Strategy and Method Description

- A test report reviews the test policy and solution, such as the test types, scenarios, and methods. It strategically describes how to perform the test. It also introduces the solution used in the test, such as the integration procedure and sequence, test procedure and sequence, test method, test tool, and test case design and execution method.
- A test report describes the test environment, such as the name, specifications, quantity, version, and account of the hardware, software, and test tools used in the test.

- A test report summarizes the test period and tester input, that is, the planned start time and end time of the test, overall test progress, key progress check points, number of testers, work division, and labor hours.

Test Metric Statistics and Analysis

- Statistics on key test metrics: quantifiable metrics of the tested system quality, such as the speed, throughput, temperature, time, and resource usage.

- Defect statistics and analysis: total number of defects, defects by severity, defect resolution rate, defect repeat rate, number of pending defect tickets, defect distribution by module, and defect source distribution. Techniques such as defect orthogonal analysis and four-quadrant defect analysis can be used.

- Test execution statistics: number and proportion of designed test cases, number of executed test cases, pass rate of test case execution, number of regression tests, manpower input for test execution, and test execution period.

- Statistics on test adequacy and test capability: coverage of requirements and functional features, test execution completion rate, code test coverage rate, test automation rate, and defect hit rate of test cases.

Test Risk Analysis and Disclosure

Based on the test process and result, a report analyzes whether the product has quality risks, and lists the risks, risk basis, risk level, and risk mitigation suggestions. Risks do not mean that the quality does not meet requirements. The risk handling strategy depends on the risk occurrence probability and the estimated loss after the occurrence. If the product of the two is very low, the risk can be accepted.

Risks are an important basis for major stakeholders to determine the overall product quality and whether the release conditions are met. Risks must be filled in logically.

Quality Evaluation and Release Suggestions

A test report provides objective quality ratings and evaluations based on company or industry standards, as well as quality results and risk analysis for reference.

It provides release suggestions such as "Release", "Delay", or "Partial release" based on the quality evaluation. The release suggestions can be specific to features. Features with high risks are not recommended for release or can be released with restrictions.

7 E-Commerce Platform Test Driven by API Automation Test Cases and Keywords

Application Scenarios

With increasingly complex functions of cloud platforms, the same pre-steps or test logic is often used during test case design. If these steps are written in each test case, the workload is heavy and the maintenance is difficult. Through the auto API test function of CodeArts TestPlan, you can create test projects, compile test cases, and run automatic execution of test case scripts. URL test steps can be set as API keywords. The keyword library manages API keywords, combined keywords, and system keywords, and makes them easy-to-use, understandable, maintainable, and reusable in different test scenarios, such as component test and system test.

This section demonstrates the test steps of product management function of an e-commerce platform.

Procedure

Querying product information of an e-commerce platform.

Step 1 Log in to the CodeArts TestPlan homepage, search for your target project, and click the project name to access the project.

Step 2 In the navigation pane, choose **Testing** > **Testing Case**.

Step 3 Click the **Auto API Test** tab and click **Create** on the right.

Step 4 Enter the case name, configure other information as required, and click **Save and Write Script**. The **Scripting** page is displayed.
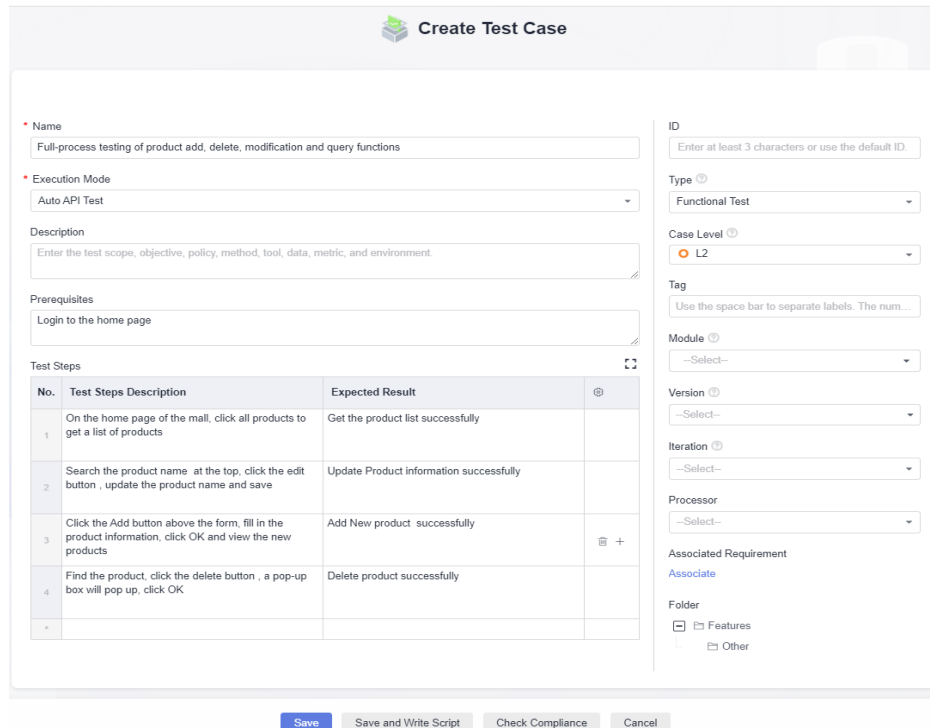
Step 5 Create a user login API. (Before managing product information, you need to log in to the e-shop home page.) On the **Scripting** page, select the **Pre-steps** tab and click **URL Request** to generate a test step.
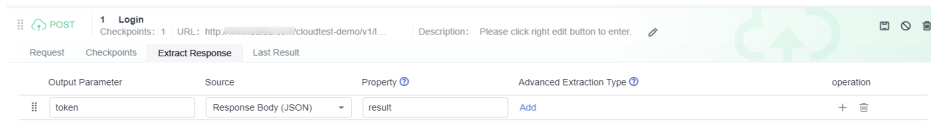
Select **POST** as the request method, enter the request URL of the tested service, and set request parameters (username and password).
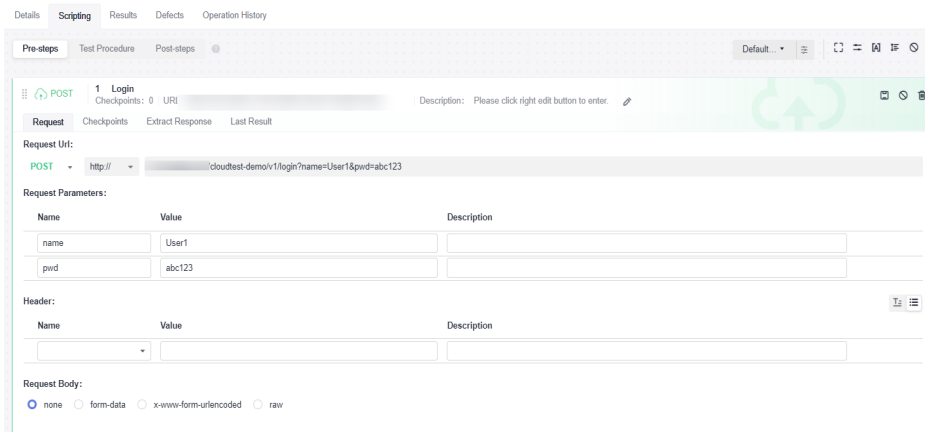
Click the **Checkpoints** tab and set the checkpoints based on the response code.
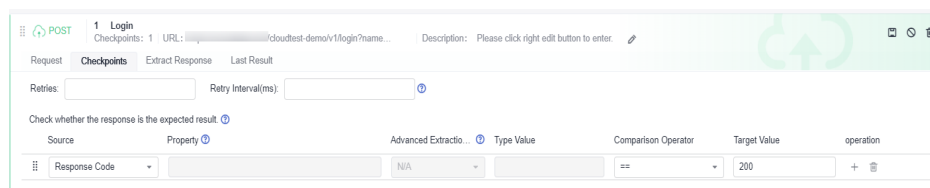
Click the **Extract Response** tab and set the response extraction parameters to extract parameters for subsequent test steps.

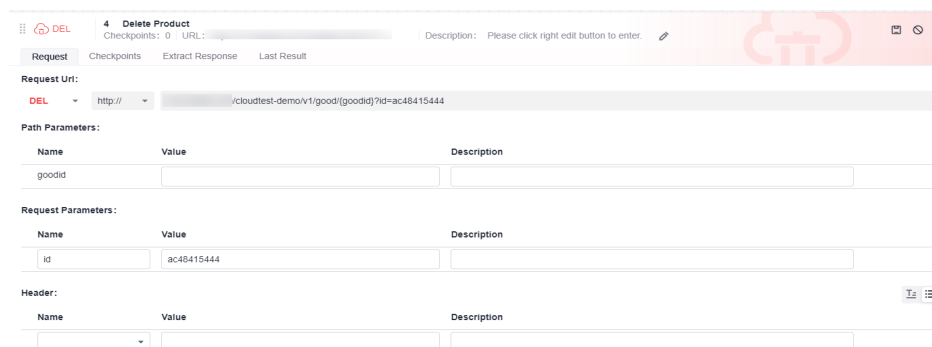**Step 6** Create an API for adding new products. On the **Scripting** page, select the **Test Procedure** tab and click **URL Request** to generate a test step.

Set the request method to **POST**, enter the request URL of the service to be tested, set the request parameters (inventory, description, icon, ID, category, name, price, and title) and request header parameters (using the response parameters extracted in the pre-steps).
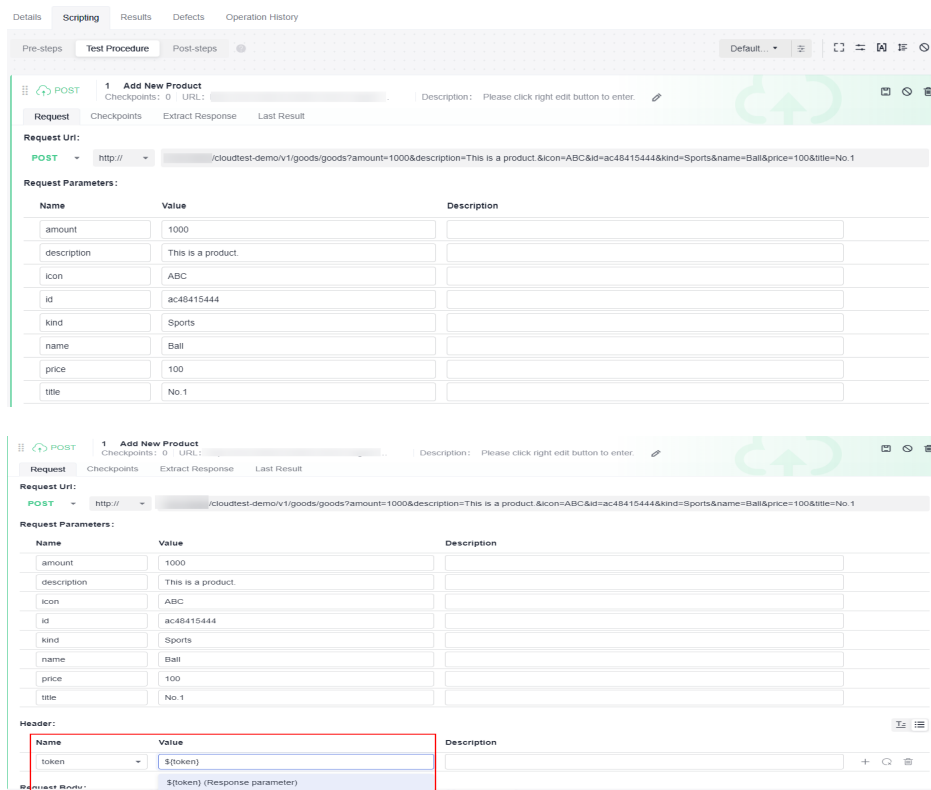
Step 7 Create an API for querying specified product information. On the **Scripting** page, select the **Test Procedure** tab and click **URL Request** to generate a test step.

Set the request method to **GET**, enter the request URL of the tested service, and set the request parameters (the product ID).
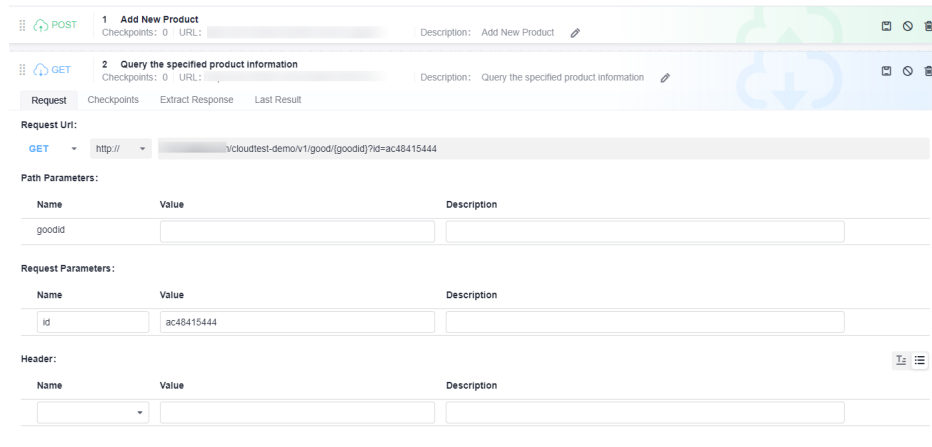
Step 8 Create an API for updating products. On the **Scripting** page, select the **Test Procedure** tab and click **URL Request** to generate a test step.

Select the request method to **PUT**, enter the request URL of the tested service, and set the request parameters (for updating the product information).
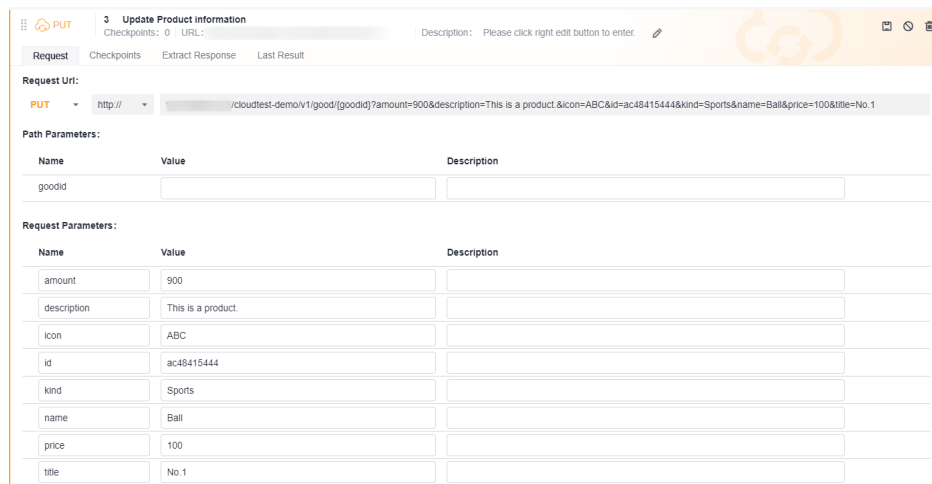

Step 9 Create an API for deleting products. On the **Scripting** page, select the **Test Procedure** tab and click **URL Request** to generate a test step.

Select the request method to **DEL**, enter the request URL of the tested service, and set the request parameters (ID of the product to be deleted).

Step 10 Create an API for querying the product list. On the **Scripting** page, select the **Test Procedure** tab and click **URL Request** to generate a test step.

Set the request method to **GET** and enter the request URL of the tested service.
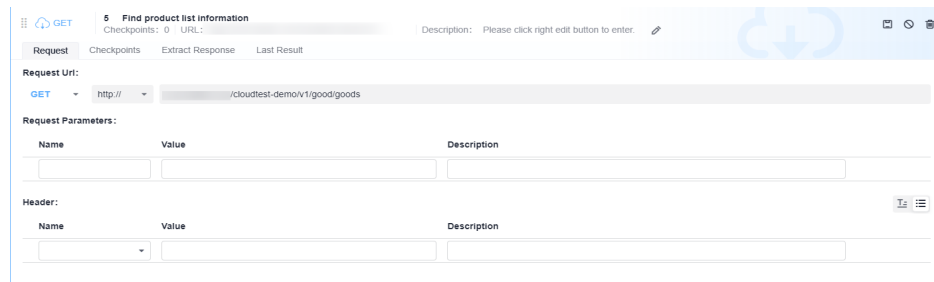
Step 11 After the script is edited, click **Save** and execute the test case. After the execution is complete, view the execution result on the **Results** tab page.

Step 12 Set test steps that may be reused in future script editing as API keywords.

Click the 💾 icon on the right of the URL request name. On the page that is displayed, set **API Keyword** and **Description**, and select the directory where the keyword is to be stored.
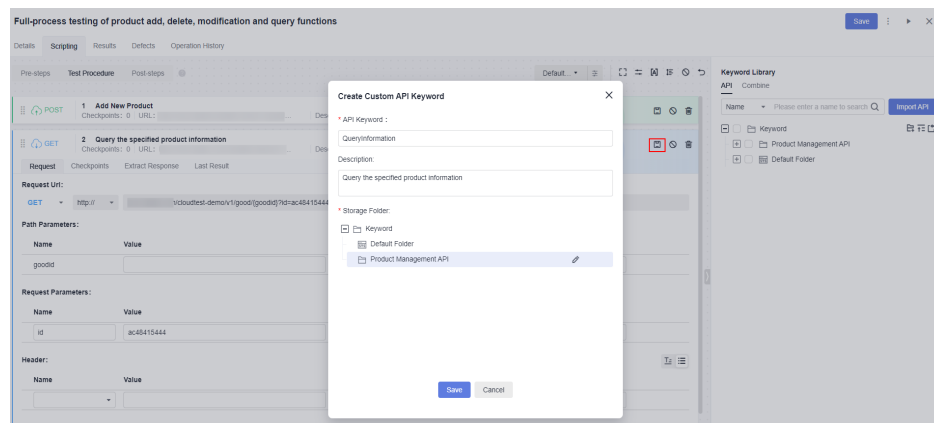
----End

Combined Keywords for a Full Process

When designing test cases, you may often use the same pre-steps or test logic. If these steps are written in each test case, the workload is heavy and the maintenance is difficult. Combined keywords encapsulate multiple test steps as common test logic. This test logic can be reused when the combined keywords are invoked by other test cases. The process of adding, deleting, modifying, and querying products can be set as a basic combined keyword for reuse on the e-commerce platform.
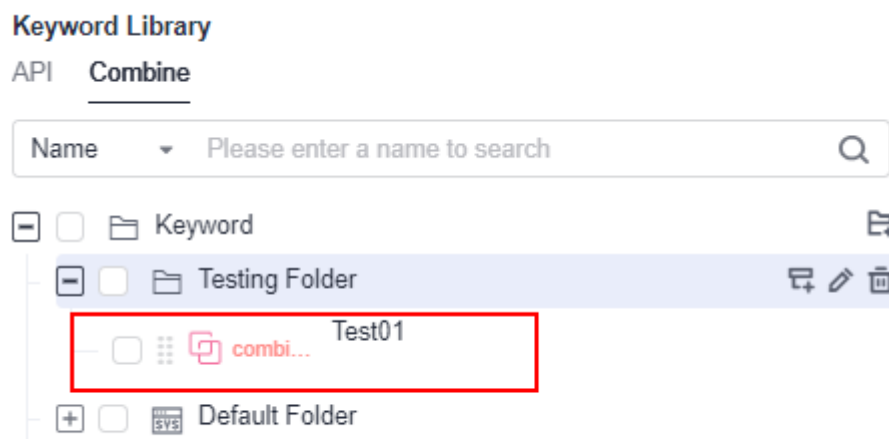
● **Scenario 1**

Step 1 Click [A] in the upper right corner of the **Scripting** page.

Step 2 Set **Name** and **Description**, select the directory where the keywords are to be stored, and set request parameters as required. Select the added **URL Request** and click **Save**.
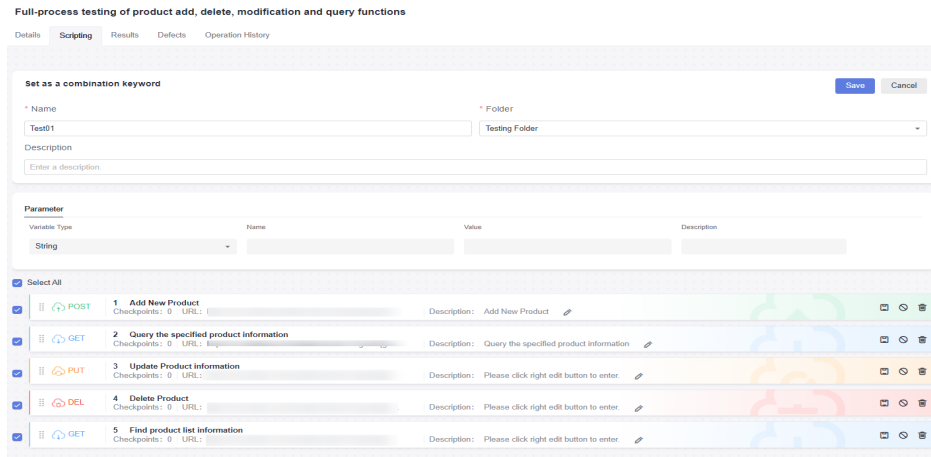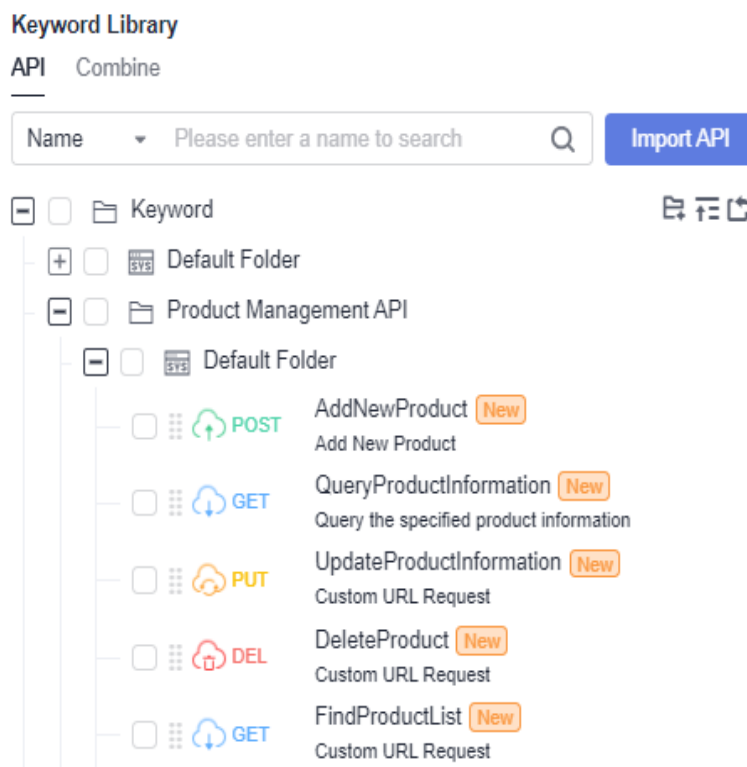

Step 3 Choose **Keyword Library** > **Combine**, and view the stored combined keywords.

----End
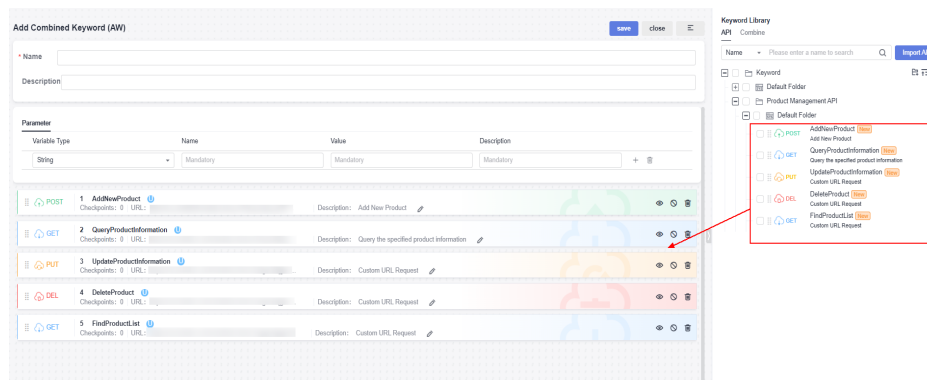
- **Scenario 2**

Step 1 Store the preceding test steps to the API of the keyword library by referring to **the steps of setting API keywords**.

Step 2 On the **Keyword Library** > **Combine** tab page, click ⬚ next to the folder where the keywords are to be saved. Set **Name** and **Description**.

Step 3 Click the **API** tab, select the folder where the keyword to be added is located, and click ✛ on the right of the keyword to be added or hover the cursor over the keyword area and drag the keyword to the test step area.

Step 4 Click **Save**.

----End