Software Engineering Fundamentals
Project No. 1 – Software Testing
PHPUnit

Domingo, Lenar
Fortaleza, Keanu Sonn
Lactaotao, Benny Gil
Morados, Lou Diamond
Octavo, Sean Drei
Samilin, Mark Francis
Santiago, Denzel Khyle

9374 CS 313 Software Engineering
November 25, 2024

Table of Contents

PART 1: Testing Concepts

1.1 The objectives of software testing

Software testing relies on a disciplined approach of evaluating software to determine whether it complies with the requirements and performs those functions under specified conditions. Testing is mostly focused on detecting and measuring the quality of software and most importantly, assessing whether the software achieved its intended purpose. It is one of the activities in the software development lifecycle that helps prevent defects, enhances the reliability of systems, and assures the users' contentment. Testing is divided into levels that include the unit, integration, system, and acceptance tests, which all address particular stages of the application. These levels work together to confirm that even the individual parts of the system and the whole system work well. Furthermore, testing can also be carried out differently, for example black-box testing which concentrates on how the system appears from outside and white-box testing which investigates the inner workings of the program [1].

Software Testing is an essential quality assurance activity in the Software Development life-cycle. Its objectives however, are more complex than the simple finding of errors; they embody a series of important objectives that significantly contribute to improved quality of the software [2].

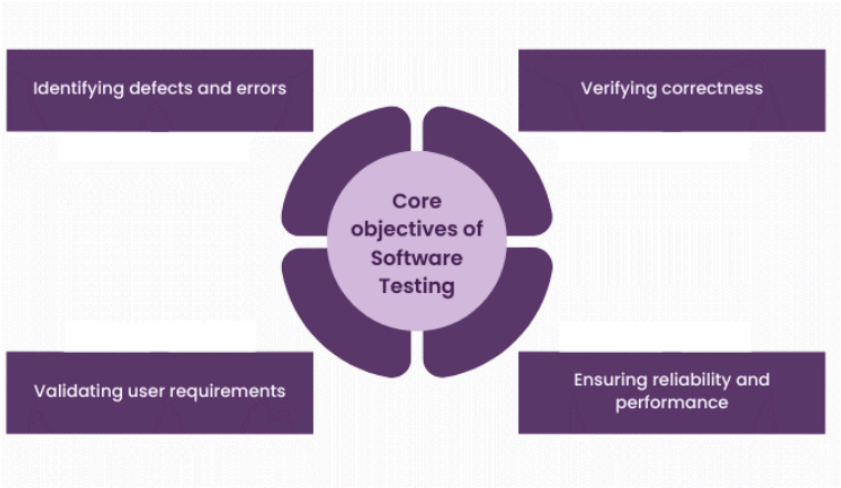


Figure 1: Core Objectives of software Testing [1]

- **Identifying defects and errors**

Identifying defects and errors is to discover even the smallest defect or failure within the code and functionality of the software. Testers, with the aid of the application, attempt to ensure that all possible defects are found, recorded, and repaired prior to the application being launched or released.

- **Verifying correctness**

Verifying Correctness is also confirming that the software works correctly, and performs all of the expected functions. It seeks to verify that all the features, modules and components work correctly and consistently as stipulated by the specifications.

- **Validating User Requirements**

Validating User Requirements is to ensure that it meets the users' expectations. Testing enables the developers to ensure that the software complies with the users' requirements and thus a user-friendly software is provided.

- **Ensuring Reliability and performance**

Ensuring Reliability and Performance is aimed to assure application reliability, stability, and performance across different environments. It evaluates the software in real life and in real loads to give the user a uniformly productive experience.

1.2 Software quality and project trade-offs

In software Testing, understanding the pivotal role of quality with respect to the time costs remains a challenge. Natural exploration of requirements focuses on functionality, however quality of software includes many aspects such as more reliable, more usable, more efficient, more maintainable, and more portable. But time, cost, and resource constraints mean that a compromise has to be made. It is also worth noting that these quality attributes tend to compete with one friend thus the need to re-evaluate prioritization. For example, a project may be implemented within stringent budgetary and time constraints, thus avoiding thorough testing in order to ensure a timely release of the product, therefore may compromise on the even containment of the product. On the other hand, an insistence on quality, might unnecessarily lengthen the time to complete the project and of course the costs involved will increase. Hence such conflicts and trade offs have to be considered based on project objectives, stakeholder requirements and market situations. The last aspect includes the trade offs themselves, which include the goals and process of communication for effective decision making such that the quality and constraints of the deliverable in terms of the product are both met [3].

- **Cost Time and Quality**

Improving the quality of the delivered software may often mean that more time and resource inputs are required, thus leading to additional costs and prolonged time frames for the project. On the other hand, the radius of development may be reduced in order to adhere to a strangling time frame. This may result in inadequacy of thorough testing and hence defects may be found in the End Product. Project management is the art of ensuring that all these elements are handled properly so as to satisfy all the stakeholders [4].

- **Testing Depth vs. Feedback Speed**

Extensive testing offers a high level of belief regarding the performance of the software however this is oftentimes a slow process. Shorter feedback loops, which can be implemented through continuous testing, make it possible to find problems more quickly but not necessarily all of them. This is what determines how deep testing will be done for a given project and the

scope of each project defines the requirement and the risk thresholds [5].

1.3 Defects: faults and failures

The creation and maintenance of software is a multi-faceted process with several analytically distinct phases which include analysis, testing, debugging, refinement and so forth. In reference to this process, several terms are often confusingly distinguished or used interchangeably that are: bug, defect, error, failure, fault and more. A defect is defined as failure to conform to the specification or requirements. A defect can thus be understood as a departure from the expected behavior of the software. Defects in a piece of software can be produced at any instance within the software's life cycle and more often than not are traceable to improper interpretation of the software's requirements or design. A Defect is the difference between what is anticipated and what is achieved. By definition a Defect is any Error that the tester is able to uncover. A defect in a software product indicates unskillfulness or incompetence in the meeting of requirements or standards laid down and as a result helps in inhibiting the software application from executing the needed and anticipated task. Also called a Fault [6].

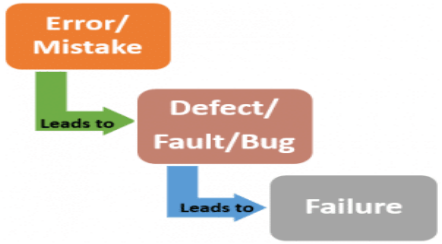


Figure 2: Defects: Faults and failure [6]

Categories of defects include the following:

- **Critical**

The term "critical" in software testing denotes a defect that needs a solution immediately. A critical defect involves the core capabilities that can otherwise affect the software product or its functionality at a broader perspective. For example, unavailability of any feature/functionality or the complete application crash

- **Major**

Defects that cause the majority of the functions of the software product to come to a standstill are known as Major Defects. Even though these defects do not leave the system unusable, they tend to suspend several core functions of the software.

- **Minor**

Minor defects have less severity and therefore do not have a great effect on the software product. The effects of these defects are apparent when the product is in use. Nevertheless, it does not stop the user from

performing the action. One may, however, perform the action in a different way.

- **Trivial**

Such defects have no bearing on product performance.

Failure is described as the incapacity of a particular software to execute its intended purpose. Failures take place when the software does not serve the desired needs or expectations of users. Failures can be caused by a plethora of issues; bugs, defects, errors and faults inclusive. A fault on the other hand is said to be a defect or error in software due to which malfunction appears on execution. A fault is a distinct problem in the code or system. It, when exercised, results in divergence from the expected outcome. This can occur due to programming mistakes, incorrect overlap of system parts, or even something related to hardware issues. Fault lies at the core of the malfunction, differentiating it from a defect, a different view altogether, which might be the result of many things [7].

The Definitions of Faults in Software Engineering and their Causes of Failures. A fault is more commonly called a bug, defect or even a gaffe and can be defined as an erroneous step, process of development or data definition in a software product. Such faults can be introduced at any point in the Software Development Life Cycle (SDLC), from the requirement stage all the way to the testing stage. If they are not exposed and rectified, faults may cause failures in which the software does not work as required [8].

1.4 The cost of fixing a defect

As the software development process evolves over time, the cost of detecting and fixing defects in software increases steeply. Finding and fixing flaws that have already been embedded in the software after it has been deployed is particularly dangerous and costly, in most cases, much more expensive than rectifying them during the early stages. There is also the risk of affecting the induced functionality of the application whenever the code is changed in an attempt to correct a bug, which implies that further changes may be required thus raising the cost, time, and labor involved.

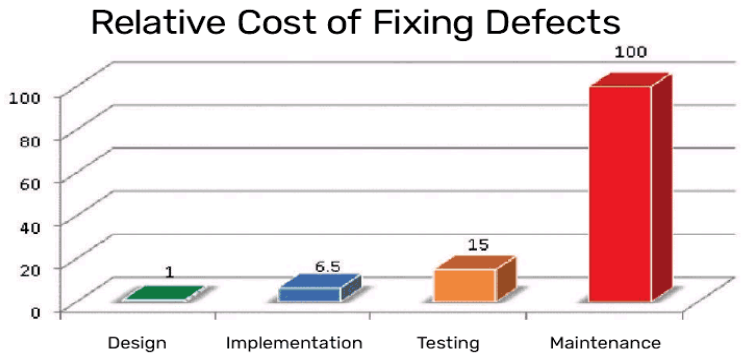


Figure 3: Relative Cost of Fixing Defects [9]

The potency of code problem detection is higher for a developer when they are working on the code rather than any other time because the code is still fresh in their mind and it is easier to resolve even intricate issues. After a while, since the code progressed to

different later stages of development, it becomes difficult for software developers to remember every little detail and how to fix issues without identifying them first. Encouraging corrections by providing an automated system of continuous quality integration is also an advantage for concurrent code writing in that corrections can be easily done as it is fresher in the developer's mind. In the final testing phase, reproducing defects in the developer's local environment can be a lengthy process. Out of certain, clear borderline cases or those which do not match the specifications, it is easy to find the problem. However, the identification of such problems as a memory leak or race condition is significantly difficult. Regrettably, this engineering defect usually doesn't manifest itself until the application goes live, assuming it wasn't found when it was still being coded [9].

PART 2: Testing Process

Software testing is an important process in the software development lifecycle. It involves verifying and validating that a software application is free of bugs, meets the technical requirements set by its design and development , and satisfies user requirements efficiently and effectively.

This process ensures that the application can handle all exceptional and boundary cases, providing a robust and reliable user experience. By systematically identifying and fixing issues, software testing helps deliver high-quality software that performs as expected in various scenarios. [10]

2.1 Software testing life cycle

Figure 4 shows the Software Testing Life Cycle (STLC) is a systematic procedure to make sure that the quality of the software is good by the definition of phases from planning to closure. During the first phase, the requirement analysis, the testers search for the parts that can be tested and clarify the objectives accordingly. The test planning is then the next stage and it is the step, where the testers describe the plans, the resources, the tools, and the timelines. The developers of the test cases are then following the needs and the data preparation is also included.

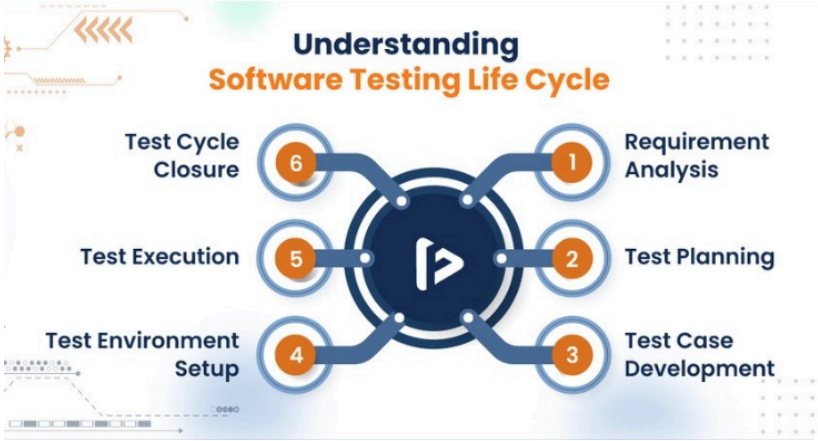


Figure 4: Understanding Software Testing Life Cycle. [11]

In the Phase 2: Environment Setup phase, an organization needs to make sure that all necessary hardware, software, and dependencies are available for executing the test. The process of testing is actually the

execution of test cases, logging the outcomes, and highlighting the errors for resolution. Testers then check the fixed problems to see if they are really corrected after that. In the last step of the process, the evaluation phase ends where the results are analyzed and the lessons learned are recorded as well as the readiness for deployment is checked [11]. On the one hand, a widely-used tool to perform unit testing in PHP, PHPUnit, typically covers minor units of code. The test cases are coded by means of PHPUnit style, and the programmers use the methods like setUp() and tearDown() to control the test environment. Assertions confirm the expected symptoms and the results are gotten to know the defects. Debugging, regression testing, and CI/CD integration are the activities that help in the continuous improvement of the quality of the software, therefore, PHP Unit is a useful tool for developers to be able to maintain the robustness of PHP applications [12].

To illustrate the software testing life cycle in PHPUnit, suppose we have a hypothetical user registration system, that is,

```
Unset
class UserRegistrationTest extends TestCase
{
  private $baseUrl =
'https://example-app.com/api';

  /**
   * Requirements Phase: User story #123
   * "As a new user, I want to register with
valid credentials"
   */
  protected function setUp(): void
  {
    // Design Phase: Setting up test environment
    $this->client = new \GuzzleHttp\Client();
  }

  /**
   * Planning/Analysis Phase:
   * - High-risk area: User authentication
   * - Early testing strategy: API contract
validation
   */
  public function testValidUserRegistration()
  {
    // Implementation Phase
    $response =
$this->client->post($this->baseUrl . '/register',
[
    'json' => [
   'email' => 'test@example.com',
   'password' => 'SecurePass123!',
   'name' => 'Test User'
   ]
]);
```

```
// Execution Phase: Assertions
    $this->assertEquals(201,
$response->getStatusCode());
    $this->assertArrayHasKey('user_id',
json_decode($response->getBody(), true));
  }
}
```

This test case example represents the software testing life cycle in PHPUnit.

2.2 Defect injections and defect causes

Figure 2: When mistakes or flaws are introduced into a software system during its development, it is termed as “defect injections”. These flaws are often the result of human errors, including but not limited to, incorrect requirement interpretation, wrong coding, or ineffective testing procedures. In real-life software development, defects can occur at any time during the Software Development Life Cycle (SDLC) [13].

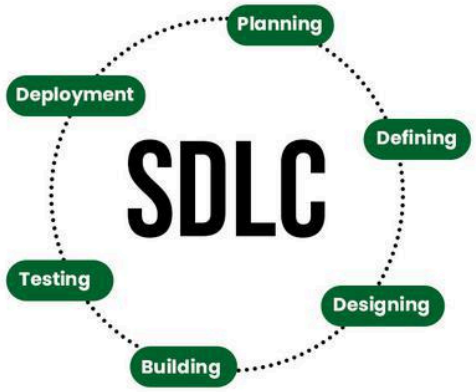


Figure 5: Software Development Life Cycle [13]

If the requirements are unclear or incomplete, the developers may get a false impression and hence, may implement the incorrect features. Likewise, coding faults are caused by logical errors, syntax issues, or deviance from best practices. The defects associated with testing come to light when the main problems are skipped in manual testing or when there are no test cases for the edge cases [14].

The dynamic nature of PHP can sometimes be the cause for fault injections in PHP-based applications through insecure server-side logic handling, insufficient user input validation, and flawed database queries that expose users to SQL injection vulnerabilities. These defects often come from the pressure of short deadlines and the lack of proper reviews, or improper coding standards. Automatization and repeatability of tests are the two main aspects that technologies such as PHPUnit offer to reduce the injection of errors. By this, developers

can confirm the code's functionality early on and regularly. Teams can minimize the number of faults as well as increase the quality of the software by finding the main causes of the faults and then taking preventive actions against them, e.g. doing a thorough requirement analysis, peer code reviews, and test-driven development [15].

We can use the hypothetical user registration system mentioned in the previous section and transform it by adding defect prevention mechanism of the application, that is,

```
Unset
/**
 * Defect Prevention: Input validation testing
 * Common injection point: Missing requirement
validation
 * @dataProvider validRegistrationDataProvider
 * @test
 */
public function
shouldSuccessfullyRegisterValidUser(array
$userData, array $expectedResponse): void
{
  // Arrange
  $this->mockHandler->append(
    new Response(201, [],
json_encode($expectedResponse))
  );

  // Act
  $response = $this->client->post('/register',
['json' => $userData]);
  $responseData =
json_decode($response->getBody(), true);

  // Assert
  $this->assertEquals(201,
$response->getStatusCode());
  $this->assertArrayHasKey('user_id',
$responseData);

$this->assertEquals($expectedResponse['user_id'],
$responseData['user_id']);
}

/**
 * @dataProvider invalidRegistrationDataProvider
 * @test
 */
public function
shouldRejectInvalidRegistrationData(array
$userData, array $expectedResponse): void
{
  // Arrange
  $this->mockHandler->append(
    new Response(400, [],
json_encode($expectedResponse))
  );

  // Act
```

```
  $response = $this->client->post('/register',
['json' => $userData]);
  $responseData =
json_decode($response->getBody(), true);

  // Assert
  $this->assertEquals(400,
$response->getStatusCode());
  $this->assertArrayHasKey('error',
$responseData);
  $this->assertEquals($expectedResponse['error'],
$responseData['error']);
}
```

2.3 Testing philosophies

The context-oriented approach to software quality assurance is stressed in testing philosophy, where control strategies are determined by the development phase, the application role and organizational needs. In contemporary development workspaces, and especially in corporate environments where dedicated QA teams are not provided, testing has become de facto a part of the development process, rather than an isolated one. For tests to be written and validated by developers, correspondingly it creates a feeling of responsibility and facilitates the registration to the highest quality standards. Testing strategies typically revolve around unit tests, which validate individual business logic, and integration tests, which ensure that components interact correctly. They are both of paramount importance, but their weight depends on the application. For example, prototypes may require fewer tests whereas production-trained systems need extensive coverage to minimize risk.

In the context of legacy codebases, integration testing is practical to begin with, focusing on critical functionalities and evolution hotspots. Then, unit tests can be added on an ad hoc basis as new or changed code is implemented. This focused strategy reduces wasted effort, but allows important parts to be properly tested. Direct testing of private functions is not recommended; instead, private functions should be implicitly validated through public interfaces. If it is not possible to test private functionality sufficiently at public interfaces, it may be a sign of design shortcomings that need to be addressed.

In some cases, organizations may implement testing in production, to gather real-world information and behavior data. Yet, that strategy is dependent on powerful observability tools, feature flags, and fast rollback features. It is most appropriate for non-critical uses, because it is a high trust area, e.g, banking, cannot justify the risks related to such tests.

Ultimately, testing is not a standalone guarantee of quality. High-quality software results from rich domain expertise, clear goals and a strong development environment with fast feedback mechanisms. Tests are

useful only when they verify the correct behavior, functioning as safety measures that prevent serious errors. Poorly designed tests or tests that confirm flawed reasoning can actually do more harm than good in the development process. Through a balanced and pragmatism outlook, companies are able to fully capitalize on testing frameworks such as PHPUnit to develop trustworthy, resilient applications without losing the flexibility to adapt the system to each project's particular requirements [16].

2.4 Testing pipeline

The testing pipeline entails the series of work, which is ordered, that aims to achieve software that complies with all quality preferences before the release. It combines different types of tests into the software development process, making it possible for developers to notice and hence to deal with defects step by step [17]. A usual testing pipeline is characterized by the transition of different methods such as unit testing, integration testing, system testing, and acceptance testing, which are in most cases automated and in alignment with CI/CD workflows hence being efficient [18].

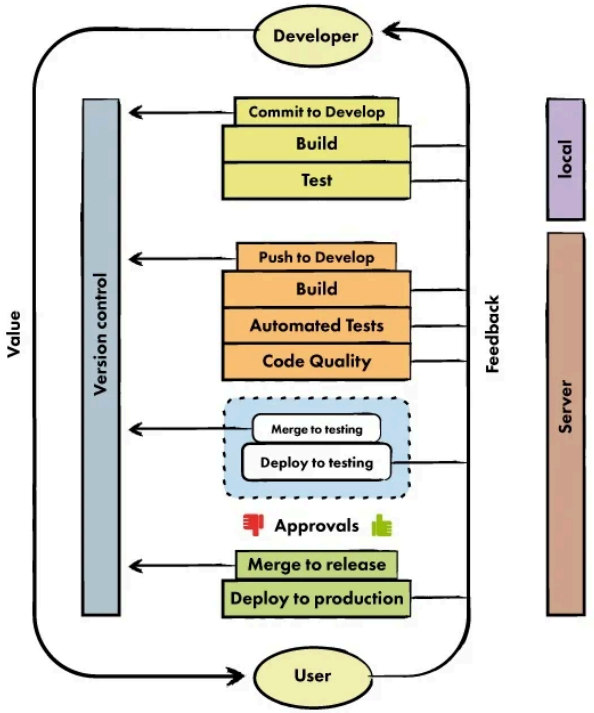


Figure 6: Simplified deployment pipeline can look like this [20].

In the case of PHP development, PHPUnit occupies the main place in the testing pipeline by carrying out unit and integration testing. Initially, the developers record test cases in PHPUnit along with their code, besides, checking the correctness of the individual components/modules. These tests are then integrated into CI/CD pipelines, where they are automatically run for each code commit or pull request, thus ensuring the non-formation of new defects. The process is escalated

in situations that are identified and hence, the pipeline suspends, giving the developers a chance to evaluate them and solve the issues. The next step is more extensive system and integration tests that authenticate the interoperability between the components.

A typical situation in a PHP-based web application, where the PHPUnit testing pipeline may start with testing diverse functions like user authentication or database queries. The pipeline is then preceded by the workflow wherein the validation of the interaction of these components within the application is done and hence smooth functionality of the whole remains undisturbed. This way of management puts every production process under very tight checks and it is a good way of ensuring that the probability of defects is lower and it also makes it possible for developers to work faster and to deliver more reliable software. Together with the CI/CD tools and following a straightforward testing pipeline, PHPUnit makes it possible for the quality assurance process to be continuous and automated [19].

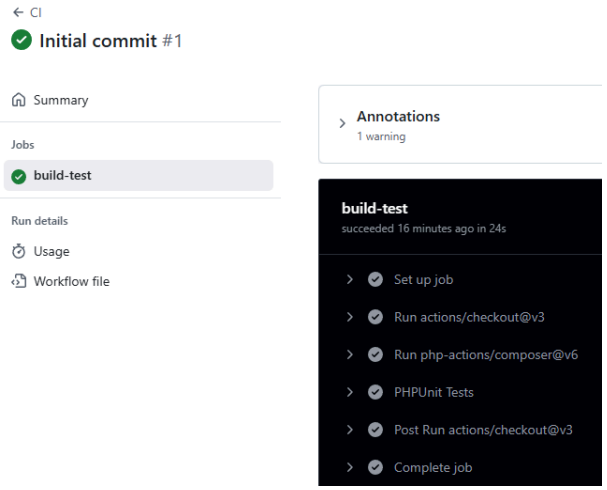
As demonstrated in php-actions/example-phpunit [21]. The GitHub Actions workflow for PHPUnit testing can be configured in .github/workflows/ci.yml. The project structure looks like this, that is,

```
Unset
- Top Directory
  - .github
    - workflows
      - ci.yml (contains configuration for
PHPUnit in GitHub Actions)
- etc..
```

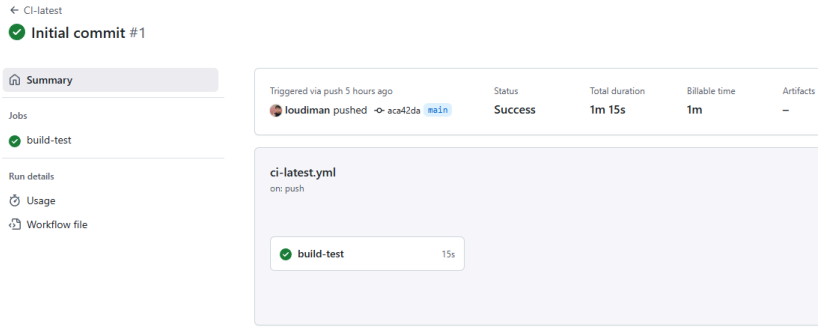
The code within ci.yml is as follows,

```
Unset
name: CI
on: [push]
jobs:
  build-test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - uses: php-actions/composer@v6

      - name: PHPUnit Tests
        uses: php-actions/phpunit@master
        env:
          TEST_NAME: Scarlett
        with:
          version: 9.6
          bootstrap: vendor/autoload.php
          configuration: test/phpunit.xml
          args: --coverage-text
```

Assuming that we pushed changes in the repository the ci.yml executes and runs automated tests, that is,



Result: Test result of the execution of ci.yml on GitHub Actions



Result: Result of PHPUnit as part of continuous integration and continuous deployment CI/CD pipeline.

This workflow automates the process of checking out the code, installing dependencies, and running tests with PHPUnit, ensuring that the codebase is continuously tested for correctness.

PART 3: Defect Detection Approaches

Defect detection is backward-facing and can be likened to steering a car by looking only in the rear-vision mirror [22]. Defect detection approaches are strategies and methodologies used to identify flaws or bugs within software applications. These approaches are critical in ensuring software functions as intended and meets quality standards before release. Effective defect detection helps minimize the occurrence of defects in production, improves user experience, and reduces long-term maintenance costs.

3.1 Test coverage and testing dimensions

Test coverage is the effectiveness of system tests in testing the code of an entire system [22]. Some companies have standards for test coverage (e.g., the system tests shall ensure that all program statements are executed at least once) [22]. There are 4 types of test coverage, that is,

- Line Coverage - Percentage of executable statements executed during testing [23].
- Branch Coverage - Percentage of decision outcomes executed [23].
- Function Coverage - Percentage of functions executed during tests [23].
- Class Coverage - Percentage of classes instantiated during testing [23].

In the context of PHPUnit, test coverage can be obtained via generating a HTML report showcasing the codes that are executed. Suppose we have a test class that measures the overall code coverage functionality, that is,

```
Unset
<?php
#[CoversClass(CodeCoverage::class)]
#[Small]
#[Group('metadata')]
final class CodeCoverageTest extends TestCase

# Data Provider Method
public static function
linesToBeCoveredProvider(): array
```

This test cases include the following:

- Empty Coverage (CoverageNoneTest)
- Class Coverage (CoverageClassTest) - covers specific lines in both a class and its parent.
- Method Coverage (CoverageMethodTest) - covers specific method lines.
- Trait Coverage (CoveredClassUsingCoveredTraitTest) - covers classes using traits.
- No Coverage (CoverageMethodNothingTest) - expects no coverage.
- Function Coverage (CoverageFunctionTest) - covers specific function lines.

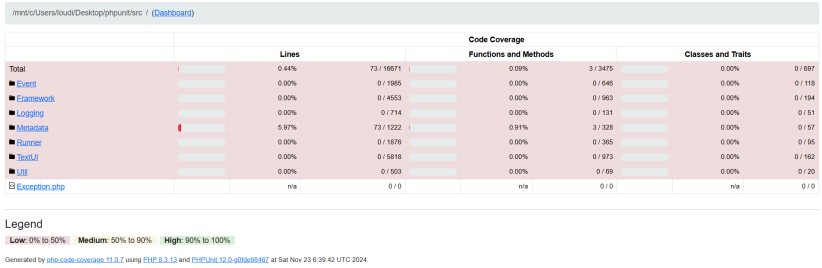
Running the CodeCoverageTest.php produces a HTML report measuring codes that are executed, that is,

```
loudayamond@LAPTOP-P2SUF1JJ:/mnt/c/Users/loudi/Desktop/phpunit$ ./phpunit --testsuite unit tests/unit/Metadata/Api/CodeCoverageTest.php --coverage-html coverage
Cannot load Xdebug - it was already loaded
PHPUnit 12.0-g0fde98467 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.3.13 with Xdebug 3.3.2
Configuration: /mnt/c/Users/loudi/Desktop/phpunit/phpunit.xml

.....                                                21 / 21 (100%)

Time: 00:09.323, Memory: 16.00 MB

OK (21 tests, 27 assertion)
```

Result: CodeCoverageTest.php when executed

The coverage report includes:

- The target test file CodeCoverageTest.php
- Any code that file tests/covers (as specified by @covers annotations)
- Any dependencies required by the tests
- Any code executed as part of the test infrastructure



Result: Coverage report

The most important test dimensions include accuracy, capability, communication, completeness, conformance, features, flexibility, serviceability, simplicity, stability, and structuredness [24]. To illustrate, let us use the following test case example made by the author of PHPUnit Sebastian Bergmann, that is,

```
Unset
#[CoversClass(DefaultPrinter::class)]
#[Medium]
final class DefaultPrinterTest extends TestCase
{
A data provider that creates different printer
instances to test
public static function providePrinter(): array


Tests the flush functionality of the printer
#[DataProvider('providePrinter')]
public function testFlush(DefaultPrinter
$printer): void

Tests error handling when an invalid socket URL
is provided
public function testInvalidSocket(): void
}
```

Running the DefaultPrinterTest.php outputs a passed unit test case, that is,

```
loudayamond@LAPTOP-P2SUF1JJ:/mnt/c/Users/loudi/Desktop/phpunit$ ./phpunit --testsuite unit tests/unit/TextUI/Output/Default/DefaultPrinterTest.php
Cannot load Xdebug - it was already loaded
PHPUnit 12.0-g0fde98467 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.3.13 with Xdebug 3.3.2
Configuration: /mnt/c/Users/loudi/Desktop/phpunit/phpunit.xml

....                                                                4 / 4 (100%)

Time: 00:13.911, Memory: 14.00 MB

OK (4 tests, 4 assertions)
```

Result: DefaultPrinterTest.php when executed

Test coverage and testing dimensions form the foundation for effective quality assurance strategies. Coverage metrics provide quantifiable data to assess testing thoroughness, helping teams identify gaps in their test suites. While 100% coverage should not be the sole goal [25], maintaining appropriate coverage across different dimensions - including statement, branch, and path coverage - enables organizations to make informed decisions about testing efforts and resource allocation. Moving forward, these concepts will guide the implementation of comprehensive testing approaches discussed in subsequent sections.

3.2 White-box versus Black-box testing versus Gray-box testing

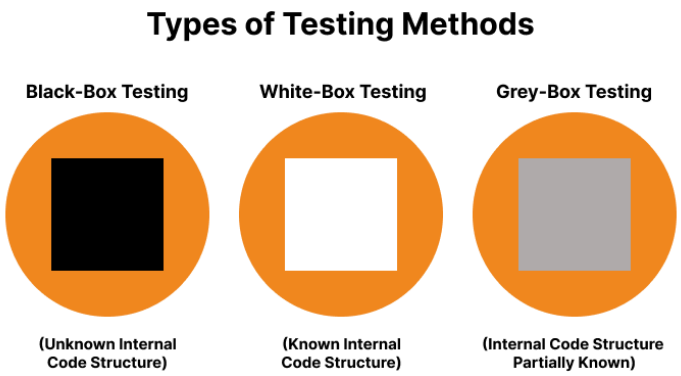


Figure 7: Types of Testing Methods

In figure 7, imagine a fintech company developing a mobile banking app that allows customers to manage their bank accounts, transfer money, and check balances. The app must be secure, functional, and user-friendly to ensure a positive customer experience. As part of the development process, the app will undergo

various testing to ensure it meets all necessary functional, security, and performance standards.

PHPUnit is used for unit testing throughout the development lifecycle to validate both internal code and external functionality.

The testing approaches used in the app development include:

- **Black-box testing**: Testing the app's functionality from an end user's perspective without knowing the internal code.
- **White-box testing**: Examining the app's internal code and logic to ensure it is secure and free of vulnerabilities.
- **Gray-box testing**: Combining the knowledge of internal structures with external functionality testing to focus on specific areas, such as authentication and permission handling.

Each testing approach is applied in PHPUnit to ensure the app is secure, user-friendly, and bug-free. Let's dive into each testing approach and see how it would be implemented in PHPUnit.

White-box Testing: Involves testing the internal structures or workings of an application. The tester has full access to the source code and examines the internal logic to identify issues such as inefficiencies, security flaws, or incorrect function implementations [26].

In the context of PHPUnit, this could involve directly testing private methods or internal behaviors.

Example: Testing the Authentication Process

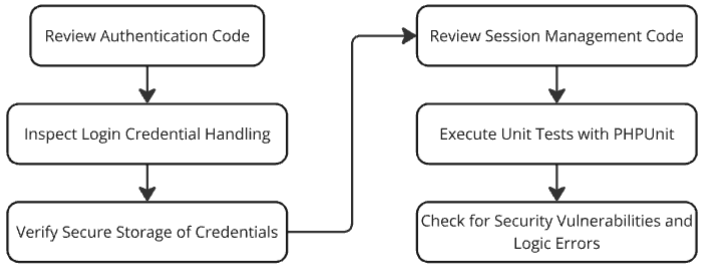


Figure 8: Testing the Authentication Process

In the mobile banking app, white-box testing could involve reviewing the code for handling user authentication to ensure it securely processes login credentials and handles session management correctly.

For instance, PHPUnit allows access to private methods through reflection, enabling detailed testing of internal logic.

```
Unset
<?php
use PHPUnit\Framework\TestCase;
```

```php
class Test2 extends TestCase
{
  private function privateMethod()
  {
    return 'expected result';
  }

  public function testPrivateMethod()
  {
    $object = new Test2();
    $method = new \ReflectionMethod(Test2::class,
'privateMethod');
    $method->setAccessible(true);
    $result = $method->invoke($object);
    $this->assertEquals('expected result',
$result);
  }
}
?>
```

```
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$ phpunit Test2.php
PHPUnit 9.6.17 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 00:00.009, Memory: 4.00 MB

OK (1 test, 1 assertion)
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$
```

Result: privateMethod when executed

- In this PHPUnit example, the private method **privateMethod** is accessed using reflection, a technique commonly employed in white-box testing to directly test internal methods and logic that are not normally accessible due to visibility restrictions.

Black-box Testing: Focuses on testing the external functionality of the application, with no knowledge of the internal code. Testers interact with the app as end users would, focusing on features such as UI responsiveness, transaction functionality, and error handling [26].

In PHPUnit, while traditionally used for unit testing, you can simulate black-box testing by checking the behavior of functions and APIs without delving into the internal workings.

Example: Testing a Login Functionality

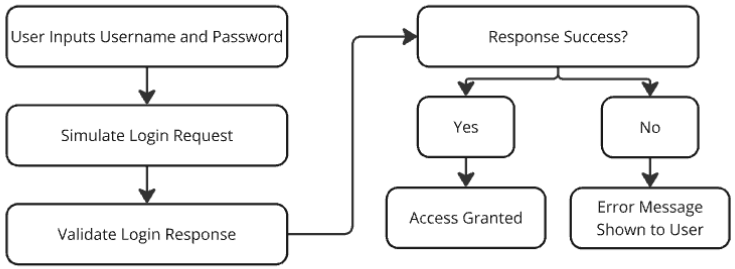


Figure 9: Testing a Login Functionality

In this scenario, black-box testing would be used

to test the login functionality of the mobile banking app. The tester interacts with the login page as an end user would, providing a username and password to verify that the system behaves correctly without knowing the internal details such as how the credentials are stored or how the authentication process works internally.

```
Unset
<?php declare(strict_types=1);

namespace PHPUnit\TestFixture;

use PHPUnit\Framework\TestCase;

final class Test3 extends TestCase
{
  public static function loginDataProvider():
array
  {
    return [
      ['validuser', 'validpassword', true],
      ['invaliduser', 'validpassword', false],
      ['validuser', 'wrongpassword', false],
      ['emptyuser', 'empty', false],
    ];
  }

  /**
   * @dataProvider loginDataProvider
   */
  public function testLogin(string $username,
string $password, bool $expectedResult): void
  {
    $response =
$this->simulateLoginRequest($username,
$password);
    $this->assertEquals($expectedResult,
$response['success']);
  }

  private function simulateLoginRequest(string
$username, string $password): array
  {
    if ($username === 'validuser' &&
$password === 'validpassword') {
      return ['success' => true];
    }
    return ['success' => false];
  }
}
```

```
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$ phpunit Test3.php
PHPUnit 9.6.17 by Sebastian Bergmann and contributors.

....                                                                4 / 4 (100%)

Time: 00:00.005, Memory: 4.00 MB

OK (4 tests, 4 assertions)
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$
```

Result: simulateLoginRequest when executed

In this PHPUnit example, the **simulateLoginRequest** method mocks the login

process, testing different username and password combinations to check for successful or failed logins. Black-box testing focuses on verifying the system's external behavior—whether it returns success or failure—without knowledge of its internal workings, such as database queries or session management. The tester validates that the system responds correctly to user input.

Gray-box Testing: Combines aspects of both white-box and black-box testing. The tester has partial knowledge of the internal workings of the application but primarily tests from the user's perspective [26].

In PHPUnit, this means that while the tester may not have full access to the code, they can still test specific components with some understanding of the underlying implementation.

Example: Testing for Invalid Account ID Handling

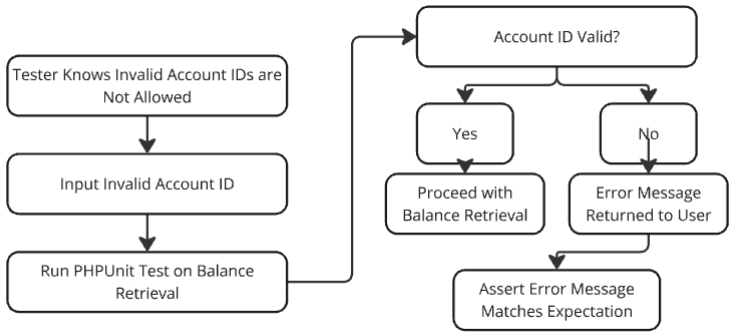


Figure 10: Testing for Invalid Account ID Handling

In this scenario, the mobile banking app needs to handle invalid account IDs gracefully during the balance retrieval process. The tester knows the app should return an error message when the account ID is invalid, but they don't know the exact implementation of the validation logic.

```
Unset
<?php
use PHPUnit\Framework\TestCase;

class Test4 extends TestCase
{
  public function
testBalanceRetrievalWithInvalidAccountId()
  {
    $account = new Account();
    $result = $account->getBalance(-1);

    $this->assertFalse($result['success']);
    $this->assertEquals('Invalid account ID',
$result['message']);
  }
}

class Account
{
  public function getBalance(int $accountId):
array
  {
    if ($accountId <= 0) {
```

```
        return ['success' => false, 'message' =>
'Invalid account ID'];
    }

    return ['success' => true, 'balance' =>
100.0];
  }
}
?>
```

```
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$ phpunit Test4.php
PHPUnit 9.6.17 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 00:00.005, Memory: 4.00 MB

OK (1 test, 2 assertions)
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$
```

Result: Account ID handler

- In this PHPUnit example, the tester focuses on the external behavior of the app—precisely how it handles invalid account IDs. While the tester has partial knowledge that invalid inputs should trigger an error message, they don't know the exact implementation details of the validation process.

3.3 Scripted versus non-scripted tests

In PHPUnit, tests are inherently **scripted**: predefined test cases are created based on expected outcomes, making PHPUnit ideal for verifying that code behaves predictably in known scenarios. Scripted tests are particularly valuable for **regression testing** (ensuring no new defects are introduced in updated code).

Example of Scripted Testing:

Unset

```
<?php
use PHPUnit\Framework\TestCase;

class Test5 extends TestCase {
    public function testValidLogin() {
        $loginSystem = new LoginSystem();
        $response =
$loginSystem->login('validuser',
'validpassword');
        $this->assertEquals('Login successful',
$response['message']);
    }
}

class LoginSystem {
    public function login(string $username,
string $password): array {
        // Simulated login logic
        if ($username === 'validuser' &&
$password === 'validpassword') {
            return ['message' => 'Login
successful'];
        }
        return ['message' => 'Invalid
credentials'];
```

```
    }
}
?>
```

```
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$ phpunit Test5.php
PHPUnit 9.6.17 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 00:00.005, Memory: 4.00 MB

OK (1 test, 1 assertion)
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$
```

Result: LoginSystem when executed

In this example, the **LoginSystem** class's **login()** method is tested with predefined inputs ('**validuser**' and '**validpassword**') and an expected output ('**Login successful**'). This type of testing ensures that the system behaves as expected under specific conditions, providing consistent, repeatable results for known scenarios.

Non-scripted Testing: While PHPUnit focuses on scripted tests, non-scripted or exploratory testing is typically done manually or through tools that allow flexible interaction with the software. However, non-scripted tests can be supplemented by PHPUnit in some cases.

For instance, you could run PHPUnit tests with different, unexpected inputs to simulate a form of non-scripted testing where you're exploring unforeseen interactions, though these wouldn't be true non-scripted tests.

Example of Non-scripted Testing in Practice:

Unset

```
<?php
use PHPUnit\Framework\TestCase;

class Test6 extends TestCase
{
  public function testUnexpectedInputHandling()
  {
    $account = new Account();

    // Exploring different unexpected inputs

$this->assertNull($account->getBalance(null)); //
Null input

$this->assertNull($account->getBalance('text'));
// Invalid data type
    $this->assertEquals(0,
$account->getBalance(-1000)); // Boundary
condition
  }
}

class Account
{
  public function getBalance($accountId)
  {
    if (!is_int($accountId) || $accountId <= 0) {
```

```
        return null; // Invalid input handling
    }
    // Simulate fetching account balance for
valid IDs
    return 1000; // Example valid balance
  }
}
?>
```

```
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$ phpunit Test6.php
PHPUnit 9.6.17 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 00:00.005, Memory: 4.00 MB

OK (1 test, 3 assertions)
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$
```

Result: getBalance method when executed

In this example, we're testing various unusual inputs to see how the **getBalance()** method handles them. While this isn't pure non-scripted testing, it incorporates some exploratory elements that could help catch unexpected behavior, such as null or invalid inputs.

3.4 Static versus dynamic tests

Static Testing: Static testing involves examining code without executing it [27], which is usually achieved through code reviews or static analysis tools rather than PHPUnit. However, a static analysis tool can be used alongside PHPUnit to identify potential issues before running tests. Static analysis checks for syntax errors, security vulnerabilities, or code standards compliance without executing the code [27].

Example: Running PHP CodeSniffer in your command line for static analysis:

```
Unset
vendor/bin/phpcbf --standard=PSR12 src/
```

```
FILE: /mnt/c/Users/Xaise/phpunit/src/Util/Xml/Xml.php
----------------------------------------------------------------------------------------------------------------------
FOUND 6 ERRORS AFFECTING 3 LINES
----------------------------------------------------------------------------------------------------------------------
  1 | ERROR | [x] Header blocks must be separated by a single blank line
  1 | ERROR | [x] Opening PHP tag must be on a line by itself
  1 | ERROR | [x] End of line character is invalid; expected "\n" but found "\r\n"
  1 | ERROR | [x] Header blocks must be separated by a single blank line
 12 | ERROR | [x] Header blocks must be separated by a single blank line
 13 | ERROR | [ ] The function-based use imports must follow the namespace declaration in the file header
----------------------------------------------------------------------------------------------------------------------
PHPCBF CAN FIX THE 5 MARKED SNIFF VIOLATIONS AUTOMATICALLY
----------------------------------------------------------------------------------------------------------------------

Time: 25.9 secs; Memory: 38MB

lenar@Xaise:/mnt/c/Users/Xaise/phpunit$
```

Result: PHP CodeSniffer when executed

- This command analyzes the code in the **src/** folder based on the PSR-12 coding standard, identifying issues such as incorrect indentation, missing docblocks, or violations of best practices without executing the code. By incorporating static analysis tools like PHP CodeSniffer before running PHPUnit tests, you can detect potential coding issues early, improve code consistency, and reduce the likelihood of runtime errors.

Dynamic Testing: Dynamic testing involves examining codes first and checking for expected outcomes [27]. Dynamic tests with PHPUnit are valuable in continuous integration environments to catch runtime errors and validate code changes during development.

Example Code for Dynamic Testing:

```
Unset
<?php
use PHPUnit\Framework\TestCase;

class ExampleClass
{
  public function
methodThatThrowsException(string $input)
  {
    if ($input === 'invalid argument') {
      throw new InvalidArgumentException('Invalid
argument provided.');
    }
  }
}

class Test7 extends TestCase
{
  public function testRuntimeError()
  {

$this->expectException(InvalidArgumentException::
class);
    (new
ExampleClass())->methodThatThrowsException('inval
id argument');
  }
}
?>
```

```
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$ phpunit Test7.php
PHPUnit 9.6.17 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 00:00.005, Memory: 4.00 MB

OK (1 test, 1 assertion)
lenar@Xaise:/mnt/c/Users/Xaise/composer-phpunit-boilertemplate/tests$
```

Result: methodThatThrowsException when executed

In this test, the **methodThatThrowsException** method is expected to throw an **InvalidArgumentException** when it is passed the string '**invalid argument**'. Dynamic testing with PHPUnit allows for the detection of runtime errors, such as this exception being thrown, ensuring that the application's error handling functions as expected under specific conditions.

PART 4: Unit & Integration Testing

Unit and integration testing are essential practices in software development that ensure individual components of an application function correctly and work together as intended.

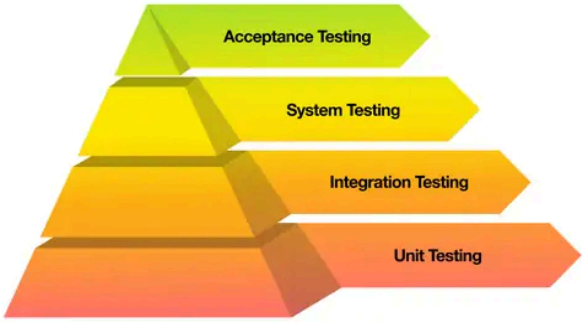


Figure 11: Levels of testing [28]

This diagram illustrates the relationship between different levels of software testing, starting from unit testing (focused on individual components) and progressing to integration, system, and acceptance testing. It emphasizes the foundational role of unit and integration testing in building reliable software [28].

Below, we explore key concepts such as functional analysis, exception testing, equivalence partitioning, boundary value analysis, and decision tables, with examples from Sebastian Bergmann's Raytracer project. Testing in this project is essential to ensure the accuracy of core algorithms, including rendering and transformations, using these techniques [29].

4.1 Functional analysis and use cases

Functional analysis, in the context of PHPUnit, involves examining the functions or methods within a PHP application to verify that they operate as expected under various conditions [30].

Use cases describe specific scenarios where the system interacts with users or other systems to achieve a goal. These help in identifying test cases that ensure all expected user behaviors are covered [31].

The following test ensures the correctness of rotating a point using transformation matrices:

```
Unset
public function
test_a_point_can_be_rotated_around_the_Z_axis():
void
{
  $p = Tuple::point(0, 1, 0);

  $halfQuarter =
Transformations::rotationAroundZ(M_PI_4);
  $fullQuarter =
Transformations::rotationAroundZ(M_PI_2);

$this->assertTrue($halfQuarter->multiplyBy($p)->e
qualTo(Tuple::point(-sqrt(2) / 2, sqrt(2) / 2,
0)));

$this->assertTrue($fullQuarter->multiplyBy($p)->e
qualTo(Tuple::point(-1, 0, 0)));
}

public function
test_a_point_can_be_rotated_around_the_Z_axis():
void
{
```

```
  $p = Tuple::point(0, 1, 0);
  $halfQuarter =
Transformations::rotationAroundZ(M_PI_4);
  $fullQuarter =
Transformations::rotationAroundZ(M_PI_2);


$this->assertTrue($halfQuarter->multiplyBy($p)->e
qualTo(Tuple::point(-sqrt(2) / 2, sqrt(2) / 2,
0)));

$this->assertTrue($fullQuarter->multiplyBy($p)->e
qualTo(Tuple::point(-1, 0, 0)));
}
```

```
> phpunit --filter test_a_point_can_be_rotated_around_the_Z_axis tests/unit/math/TransformationsTest.php
PHPUnit 11.4.3 by Sebastian Bergmann and contributors.the_Z_axis tests/unit/math/TransformationsTest.php

Runtime:       PHP 8.3.13
Configuration: /Users/benny/raytracer/phpunit.xml

.                                                                   1 / 1 (100%)

Time: 00:00.002, Memory: 23.14 MB

OK (1 test, 2 assertions)
```

Result: TransformationsTest.php when executed

This test begins by initializing a point *p* at coordinates (0, 1, 0). A transformation matrix is then created for a 45-degree (π/4 radians) rotation around the Z-axis, referred to as a half-quarter rotation, and applied to the point. The test asserts that the result of this rotation is approximately (-√2/2, √2/2, 0). Next, a transformation matrix for a 90-degree (π/2 radians) rotation, called a full-quarter rotation, is applied to the same point. The test verifies that the resulting point is (-1, 0, 0).

By confirming the expected outcomes of these transformations, this test demonstrates the accurate implementation of rotation operations around the Z-axis.

4.2 Exception testing

Exception testing ensures that the application behaves appropriately under unexpected or erroneous conditions. This focuses on how well the system handles anomalies, such as invalid inputs or system failures, without crashing. It validates that the system gracefully manages edge cases and provides meaningful feedback for errors [32].

The following test ensures that the program handles unexpected errors at runtime:

```
Unset
public function
test_a_point_cannot_be_added_to_a_point(): void
{
  $p = Tuple::point(3.0, -2.0, 5.0);

$this->expectException(RuntimeException::class);
```

```
        /* @noinspection
    UnusedFunctionResultInspection */
        $p->plus($p);
    }
```

```
> phpunit --filter test_a_point_cannot_be_added_to_a_point tests/unit/math/TupleTest.php
PHPUnit 11.4.3 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.3.13
Configuration: /Users/benny/raytracer/phpunit.xml

.                                                                   1 / 1 (100%)

Time: 00:00.002, Memory: 23.14 MB

OK (1 test, 1 assertion)
```

Result: TupleTest.php when executed

This test validates how the system handles exceptional situations by ensuring that invalid operations are correctly identified and handled. Specifically, it verifies that attempting to add a point to another point throws a *RuntimeException*. Proper exception handling like this is essential for maintaining system robustness and preventing invalid actions.

4.3 Equivalence partitioning

Equivalence partitioning is a black-box testing technique used to divide input data into valid and invalid partitions (or groups). This approach reduces the number of test cases while ensuring comprehensive coverage by focusing on representative values from each partition [33].

This test verifies that matrix multiplication handles valid and invalid input partitions by checking valid square matrices and invalid non-square matrices:

```
Unset
<?php declare(strict_types=1);

namespace SebastianBergmann\Raytracer\Tests;

use InvalidArgumentException;
use PHPUnit\Framework\TestCase;
use SebastianBergmann\Raytracer\Matrix;

final class MatrixEquivalenceTest extends
TestCase
{
  /**
   * Valid Partition:
   * - Both matrices are square matrices of the
same size
   * - Contains valid numeric elements
   */
  public function
testValidMatrixMultiplication(): void
  {
    // Arrange
    $matrixA = Matrix::fromArray([
      [1.0, 2.0],
      [3.0, 4.0]
    ]);

    $matrixB = Matrix::fromArray([
      [2.0, 1.0],
```

```
      [1.0, 2.0]
    ]);

    $expected = Matrix::fromArray([
      [4.0, 5.0],
      [10.0, 11.0]
    ]);

    // Act
    $result = $matrixA->multiply($matrixB);

    // Assert

$this->assertTrue($result->equalTo($expected));
  }

  /**
   * Invalid Partition:
   * - Non-square matrix input
   */
  public function
testInvalidMatrixMultiplication(): void
  {
    // Assert

$this->expectException(InvalidArgumentException::
class);
    Matrix::fromArray([
      [1.0, 2.0],
      [3.0, 4.0]
    ])->multiply(
```

```
      Matrix::fromArray([
        [2.0, 1.0],
        [1.0, 2.0],
        [3.0, 4.0] // This makes it non-square
(2x3)
      ])
    );
  }
}
```

```
PS C:\Users\RandomName\raytracer> vendor\bin\phpunit --no-coverage .
\tests\unit\math\MatrixEquivalenceTest.php
PHPUnit 11.4.3 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.4.1
Configuration: C:\Users\RandomName\raytracer\phpunit.xml

..
2 / 2 (100%)

Time: 00:00.013, Memory: 8.00 MB

OK (2 tests, 2 assertions)
```

Result: MatrixEquivalenceTest when executed

Input Range	**Representative Value**	**Expected Output**
Valid (Square	Matrix A = [[1.0, 2.0],	Valid (Returns

matrices of same dimensions)	[3.0, 4.0]] Matrix B = [[2.0, 1.0], [1.0, 2.0]]	[[4.0, 5.0], [10.0, 11.0]])
Invalid (Non-square matrices)	Matrix A = [[1.0, 2.0], [3.0, 4.0]] Matrix B = [[2.0, 1.0], [1.0, 2.0], [3.0, 4.0]]	Invalid (Throws InvalidArgumentException)
Invalid (Different dimensions)	Matrix A = [[1.0, 2.0]] Matrix B = [[1.0], [2.0]]	Invalid (Throws InvalidArgumentException)

Table 1: Test Case Partitions Table

The example in aboove uses equivalence partitioning to test representative values from valid and invalid input ranges. It ensures that the multiply function correctly processes valid inputs (matrices of matching square dimensions) and rejects invalid inputs (non-square or mismatched dimensions). By focusing on representative values, this approach minimizes redundant tests while maximizing coverage of the matrix multiplication operation.

4.4 Boundary value analysis

Boundary Value Analysis (BVA) focuses on testing values at the edges of input ranges. The rationale is that defects are more likely to occur at boundary points. This technique complements equivalence partitioning by specifically targeting these critical values [30].

This test demonstrates boundary value analysis for rotating matrices by checking key rotational angles (zero, quarter, full, and negative rotations):

```
Unset
<?php

use PHPUnit\Framework\TestCase;
use PHPUnit\Framework\Attributes\DataProvider;
use SebastianBergmann\Raytracer\Transformations;
use SebastianBergmann\Raytracer\Matrix;

class TransformationsBoundaryTest extends
TestCase
{

#[DataProvider('rotationBoundaryValuesProvider')]
    public function
testRotationAroundXBoundaryValues(float $angle,
Matrix $expected): void
    {
        $matrix =
Transformations::rotationAroundX($angle);

$this->assertTrue($matrix->equalTo($expected));
```

```
    }

    public static function
rotationBoundaryValuesProvider(): array
    {
        return [
            'zero_rotation' => [
                0.0,
                Matrix::fromArray([
                    [1.0, 0.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0, 0.0],
                    [0.0, 0.0, 1.0, 0.0],
                    [0.0, 0.0, 0.0, 1.0],
                ])
            ],
            'quarter_rotation' => [
                pi() / 2,
                Matrix::fromArray([
                    [1.0, 0.0, 0.0, 0.0],
                    [0.0, cos(pi() / 2),
-sin(pi() / 2), 0.0],
                    [0.0, sin(pi() / 2), cos(pi()
/ 2), 0.0],
                    [0.0, 0.0, 0.0, 1.0],
                ])
            ],
            'full_rotation' => [
                2 * pi(),
                Matrix::fromArray([
                    [1.0, 0.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0, 0.0],
                    [0.0, 0.0, 1.0, 0.0],
                    [0.0, 0.0, 0.0, 1.0],
                ])
            ],
            'negative_angle' => [
                -pi() / 4,
                Matrix::fromArray([
                    [1.0, 0.0, 0.0, 0.0],
                    [0.0, cos(-pi() / 4),
-sin(-pi() / 4), 0.0],
                    [0.0, sin(-pi() / 4),
cos(-pi() / 4), 0.0],
                    [0.0, 0.0, 0.0, 1.0],
                ])
            ]
        ];
    }
}
```

```
PS C:\Users\RandomName\raytracer> vendor\bin\phpunit --no-cov
erage .\tests\unit\math\TransformationsBoundaryTest.php
PHPUnit 11.4.3 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.4.1
Configuration: C:\Users\RandomName\raytracer\phpunit.xml

....
       4 / 4 (100%)

Time: 00:00.013, Memory: 8.00 MB

OK (4 tests, 4 assertions)
```

Result: TransformationsBoundaryTest when executed

This test demonstrates boundary value analysis for the *rotationAroundX* function by testing critical angles: zero rotation (0), quarter rotation (π/2), full rotation (2π), and negative rotation (-π/4). These values verify both the mathematical correctness of the transformation matrix and proper handling of boundary conditions in rotation calculations. The test ensures the function maintains accuracy at these key rotational points where trigonometric calculations are most prone to errors.

4.5 Decision tables

Decision tables are used to model complex decision-making processes where multiple conditions determine different outcomes. They help in identifying test scenarios involving combinations of inputs [32].

This test uses a decision table to validate the behavior of ray-sphere intersection computation under different conditions, such as ray origin, direction, and intersection time:

```
Unset
public function
test_intersection_using_decision_table(): void
{
$testCases = [
// t, ray origin, ray direction, expected point,
expected eye, expected normal, expected inside
[4.0, Tuple::point(0, 0, -5), Tuple::vector(0, 0,
1), Tuple::point(0, 0, -1), Tuple::vector(0, 0,
-1), Tuple::vector(0, 0, -1), false],
[1.0, Tuple::point(0, 0, 0), Tuple::vector(0, 0,
1), Tuple::point(0, 0, 1), Tuple::vector(0, 0,
-1), Tuple::vector(0, 0, -1), true],
];

foreach ($testCases as $testCase) {
[$t, $rayOrigin, $rayDirection, $expectedPoint,
$expectedEye, $expectedNormal, $expectedInside] =
$testCase;

$r = Ray::from($rayOrigin, $rayDirection);
$s = Sphere::default();
$i = Intersection::from($t, $s);

$comps = $i->prepare($r);

$this->assertSame($comps->t(), $i->t());
$this->assertSame($comps->shape(), $i->shape());
$this->assertTrue($comps->point()->equalTo($expec
tedPoint));
$this->assertTrue($comps->eye()->equalTo($expecte
dEye));
$this->assertTrue($comps->normal()->equalTo($expe
ctedNormal));
$this->assertSame($comps->inside(),
$expectedInside);
```

```
}
}
```

```
PS C:\Users\RandomName\raytracer> vendor\bin\phpunit --no-coverage -
-filter test_intersection_using_decision_table .\tests\unit\intersec
tion\IntersectionTest.php
PHPUnit 11.4.3 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.4.1
Configuration: C:\Users\RandomName\raytracer\phpunit.xml

.
1 / 1 (100%)

Time: 00:00.017, Memory: 8.00 MB

OK (1 test, 12 assertions)
```

Result: Test case result that uses a decision table

Parameter	Value
t	4.0
Ray Origin	Tuple::point(0, 0, -5)
Ray Direction	Tuple::vector(0, 0, 1)
Expected Point	Tuple::point(0, 0, -1)
Expected Eye	Tuple::vector(0, 0, -1)
Expected Normal	Tuple::vector(0, 0, -1)
Expected Inside	false
Condition	Ray starts outside the sphere, pointing inward

Table 2: Test Case 1

Parameter	Value
t	1.0
Ray Origin	Tuple::point(0, 0, 0)
Ray Direction	Tuple::vector(0, 0, 1)
Expected Point	Tuple::point(0, 0, 1)
Expected Eye	Tuple::vector(0, 0, -1)
Expected Normal	Tuple::vector(0, 0, -1)
Expected Inside	true
Condition	Ray starts inside the sphere, pointing outward.

Table 3: Test Case 2

This test uses a decision table approach to validate the behavior of the ray-sphere intersection computation. It evaluates different combinations of

conditions, such as the ray's origin, direction, and intersection time (t), and verifies that the prepared computations (prepare) return expected outcomes. Key parameters include:

1. **t**: Intersection time along the ray.
2. **Ray Origin**: Starting point of the ray.
3. **Ray Direction**: Direction in which the ray is cast.
4. **Expected Point**: The point of intersection on the sphere.
5. **Expected Eye**: The vector pointing from the intersection point to the ray's origin.
6. **Expected Normal**: The normal vector at the intersection point.

Expected Inside: Boolean indicating whether the intersection occurs inside the sphere.

This structured method ensures that all combinations of conditions are tested, making it easier to handle scenarios like intersections inside or outside the sphere. Each test case validates that the computations align with the expected physics-based behavior of ray-sphere intersections.

PART 5: System & User Acceptance Testing

System testing is a set of tests that compares the demands and requirements of the end user with the behavior and overall function of the system [34]. It checks whether components are working with each other correctly and handles the correct data upon interaction. This form of testing, which comes after integration testing, focuses on confirming that the system satisfies both functional and non-functional requirements, including security, usability, and performance [35].

On the other hand, User acceptance testing is often the final stage in software development, where the intended or target audience and business representatives test the software itself [36], [37]. The intended users test the software to see if it performs as intended in practical settings. This stage is also known as beta testing [36], in which end users can use the program before its official release to determine whether any features have been missed or are flawed.

With the team’s tool, PHPUnit, system testing can be observed through integration tests, which can check interactions between different modules or services in the system. Regarding user acceptance testing, one of PHPUnit's features can simulate user scenarios by executing tests replicating end-user interactions.

5.1 Cross feature testing

Under system testing, cross feature testing ensures that software works as an integrated whole. This activity tests different features of a system with each other to make sure that there are no conflicts or unexpected behavior between them. Since PHPUnit facilitates integration testing, developers can test multiple modules or features on how they interact with each other.

To showcase cross feature testing, the following test case demonstrates how PHPUnit and WebDriver can

validate multiple functionalities on an already deployed website like Wikipedia. The test verifies:

1. Search Functionality: Ensures that entering a query in the search bar and submitting it leads to the correct results page.
2. Language Switching: Tests whether switching to another language (e.g., Español) works correctly.
3. Navigation to "Contents": Validates navigation to the "Contents" page through the dropdown menu.

Below is the testCrossFeatureIntegration function, which highlights how these interactions are tested:

```
Unset
<?php
private function navigateToHomePage()
{
  $startTime = microtime(true);
  $this->driver->get(self::BASE_URL);
  $endTime = microtime(true);
  $this->logger->info('Navigated to Wikipedia
homepage', [
    'url' => self::BASE_URL,
    'response_time' => $endTime - $startTime
  ]);
}

private function searchForTerm($term)
{

$this->waitForElement(self::SEARCH_BOX_SELECTOR);
  $searchBox =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BOX_SELECTOR));
  $this->assertTrue($searchBox->isDisplayed(),
"Search bar is not visible.");
  $searchBox->sendKeys($term);
  $this->logger->info('Entered search term',
['term' => $term]);

  $startTime = microtime(true);
  $searchButton =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BUTTON_SELECTOR));
  $searchButton->click();
  $endTime = microtime(true);
  $this->logger->info('Clicked search button', [
    'response_time' => $endTime - $startTime
  ]);
}

private function
verifySearchResults($expectedUrlContains)
{
  $startTime = microtime(true);
  try {

$this->waitForUrlContains($expectedUrlContains);
  } catch
(\Facebook\WebDriver\Exception\TimeoutException
$e) {
    $currentURL = $this->driver->getCurrentURL();
```

```
    $this->logger->error('Timeout waiting for URL
to contain expected string', [
      'current_url' => $currentURL,
      'expected_url_contains' =>
$expectedUrlContains
    ]);
    throw $e;
  }
  $endTime = microtime(true);
  $currentURL = $this->driver->getCurrentURL();
  $this->logger->info('Search results loaded', [
    'url' => $currentURL,
    'response_time' => $endTime - $startTime
  ]);

$this->assertStringContainsString($expectedUrlCon
tains, $currentURL, "Search result page URL is
not as expected.");
}
```

Note: We used ChromeDriver server to act as a bridge between selenium and chrome.

The test case runs successfully on the command line interface, showing the memory usage and completion time, which is:

```
C:\Users\sdaoc\composer-phpunit-boilertemplate>vendor\bin\phpunit tests/WikipediaCrossFeatureTest.php
PHPUnit 11.4.0 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.4.1

.                                                                   1 / 1 (100%)

Time: 00:07.674, Memory: 8.00 MB

OK (1 test, 2 assertions)
```

Result: Test case result for Cross feature testing

This test showcases cross feature testing by evaluating independent functionalities and their integration within the system. Executing this test ensures that Wikipedia's features, such as search, language switching, and navigation, work seamlessly together, reflecting how users interact with the site.

5.2 Cause-effect graphing

Cause-effect graphing is a technique used in black box testing, in which logical operators (e.g., AND, OR) ensure that the system behaves as expected under all combinations of input conditions. This method lowers the cost and execution time, leading to fewer test cases while covering the test areas required to meet software quality standards [38]. This technique maps a set of causes to a set of effects, where the causes are the system's inputs, and the effects are the outputs. These causes and effects are now graphed to visually depict their relationships, allowing the testers to create optimized tests covering all critical scenarios.

With PHPUnit, cause-effect graphing can be addressed by using data providers to generate test cases dynamically based on input conditions, which, in this case, are the causes and their expected outputs, which are the effects. Each input and output combination is

derived from the graph and can be represented as a data set or a decision table to display the coverage of all logical paths.

To showcase cause-effect graphing, the test case below verifies the search functionality on Wikipedia, in which we test two scenarios. The first scenario is where search results are found, and the second is an unsuccessful search where no results are returned.

```
Unset
/**
 * @dataProvider causeEffectDataProvider
 */
public function testCauseEffectSearch($input,
$expectedOutcome)
{
  $this->driver->get(self::BASE_URL);

$this->waitForElement(self::SEARCH_BOX_SELECTOR);
  $searchBox =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BOX_SELECTOR));
  $searchBox->sendKeys($input);
  $searchButton =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BUTTON_SELECTOR));
  $searchButton->click();

  if ($expectedOutcome === 'success') {
    try {

$this->waitForElement(self::PAGE_TITLE_SELECTOR);
      $pageTitle =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::PAGE_TITLE_SELECTOR));
      $this->assertStringContainsString($input,
$pageTitle->getText(), "Page title does not
contain the search term.");
    } catch
(\Facebook\WebDriver\Exception\TimeoutException
$e) {
      $currentURL =
$this->driver->getCurrentURL();
      echo "Current URL: " . $currentURL . "\n";
      $pageSource =
$this->driver->getPageSource();
      echo "Page Source: " . $pageSource . "\n";
      throw $e;
    }
  } else {
    try {

$this->waitForElement(self::NO_RESULTS_SELECTOR);
      $noResultsMessage =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::NO_RESULTS_SELECTOR));

$this->assertTrue($noResultsMessage->isDisplayed(
), "No results message is not displayed.");
    } catch
(\Facebook\WebDriver\Exception\TimeoutException
$e) {
```

```
        $currentURL =
$this->driver->getCurrentURL();
        echo "Current URL: " . $currentURL . "\n";
        $pageSource =
$this->driver->getPageSource();
        echo "Page Source: " . $pageSource . "\n";
        throw $e;
      }
    }
}
```

Test Case ID	**Input Search Term (Cause)**	**Expected Outcome (Effect)**	**Condition to Check for Success**	**Condition to Check for Failure**
TC1	Software engineering	success	Page title contains search term	N/A
TC2	Nonexistentsearch	error	N/A	Presence of “no results” message

Table 4: Decision table of WikipediaCauseEffectTest.php

Table 4, the series of outcomes of the test cases are shown. In the first case, where the search term “Software engineering” is provided, the test case expects the search results about the search term, with the page title containing the search term. This first case is the successful scenario, and the test passes when the expected outcome occurs, that is, when the system behaves as it is expected.

The second scenario is where the search term “Nonexistentsearch” is given. Since there is no information about this search term on Wikipedia, the outcome of the system is that it will display a “no results” message. This now verifies the failure condition, which the system responds appropriately when no matching results are found.

Overall, by thoroughly evaluating both predicted behaviors and edge cases and reducing needless repetition in test case execution, this technique aids in maintaining high software quality.

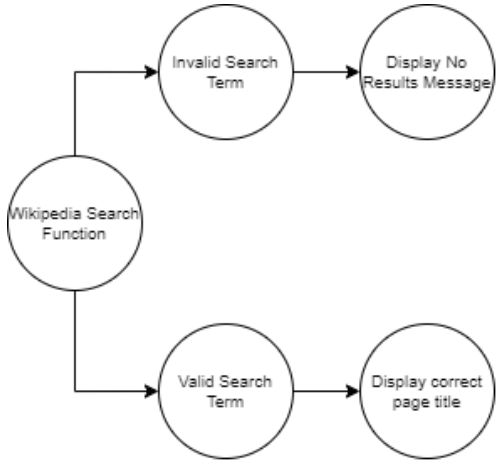


Figure 12: Cause-effect graph of the test case

In figure 12, it represents the relationships of the input conditions (causes), and the expected system behaviors (effects). If we go further into detail, the first scenario, in which the user provides a valid search term, as indicated on the graph, proceeds to the displaying of the correct page title. And for the second scenario, when there are no matching search terms, the system should display a no “no results” message.

5.3 Scenario testing

Scenario testing, primarily an activity under User Acceptance Testing, involves creating and executing tests based on real-world use cases to ensure that the system meets user expectations. The process of developing a scenario test is as follows [39]:

1. A scenario is designed based on the use cases related to the end users.
2. The test cases are developed from those scenarios.
3. The test cases are evaluated to check whether it covers all the use cases.

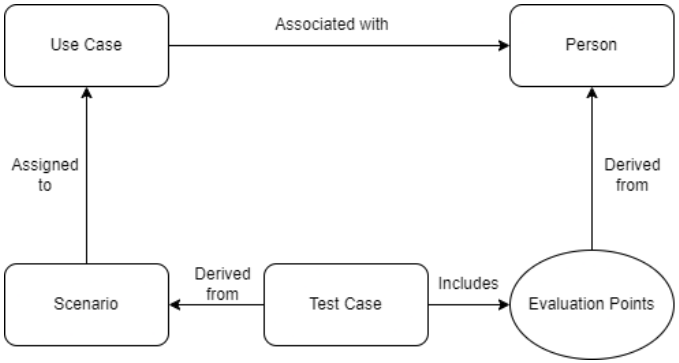


Figure 13: Scenario Testing Process

In figure 13, the process identifies use cases linked to particular users or personas (shown as "Person"). These use cases are then transformed into comprehensive scenarios that specify how users engage with the system. Test cases are created from these scenarios to ensure the system operates as intended in a range of scenarios. Specific evaluation points that specify the requirements for validation are included in every test case. The iterative process ensures comprehensive testing by assigning scenarios to relevant stakeholders or testers.

While being primarily a unit testing tool, PHPUnit can handle this task by programmatically emulating workflows. Despite its emphasis on testing individual functions or classes, PHPUnit can be modified for scenario testing in the methods listed below:

1. Simulating Real-World Workflows

   By executing a series of methods and verifying what was expected at each stage,

PHPUnit allows developers to create test cases that replicate real-world situations. In the case of e-commerce, there are scenarios where:

- A user searches for a product.
- Adds the product to the cart.
- Proceeds to checkout and place an order.

2. Testing Interdependencies

   Testing several interconnected components is a common task for scenarios. Mock objects can be used with PHPUnit to simulate system actions or API responses. By doing this, the entire scenario is guaranteed to function flawlessly without requiring real service dependencies for testing.

3. Validating Edge Cases and Alternate Flows

   PHPUnit is capable of validating alternative or edge-case workflows and evaluating the main success scenario. As an example, testing what occurs when:

   - A user enters invalid credentials during login.
   - A product in the cart goes out of stock before checkout.

4. Data-Driven Scenario Testing

   PHPUnit supports data providers, allowing the execution of several test scenarios with a variety of input datasets. Because it guarantees coverage across a wide range of real-world conditions, this feature is beneficial for scenario testing.

In this example below, the AmazonScenarioTest test case simulates a real-world user journey. This scenario includes the following steps:

1. Searching for a product using the search bar.
2. Selecting the first product from the results.
3. Adding the selected product to the shopping cart.
4. Navigating to the cart and initiating the checkout process.
5. Handling the sign-in process required for checkout.
6. Verifying redirection to the checkout address selection page.

```
Unset
public function testScenarioWorkflowWithSignIn()
{
  $this->navigateToHomePage();
  $this->performSearch();
  $this->selectProduct();
  $this->addToCart();
  $this->navigateToCart();
  $this->proceedToCheckout();
  $this->handleSignIn();
  $this->verifyCheckoutRedirection();
}

private function navigateToHomePage()
{
  $this->driver->get(self::BASE_URL);
  $this->logger->info('Navigated to Amazon
homepage', ['url' => self::BASE_URL]);
```

```
}

private function performSearch()
{

$this->waitForElement(self::SEARCH_BOX_SELECTOR);
  $searchBox =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BOX_SELECTOR));
  $searchBox->sendKeys('Fan');
  $searchButton =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BUTTON_SELECTOR));
  $searchButton->click();
  $this->logger->info('Performed search for
Fan');
}

private function selectProduct()
{
  $this->waitForElement(self::PRODUCT_SELECTOR);
  $product =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::PRODUCT_SELECTOR));
  $product->click();
  $this->logger->info('Selected product');
}

private function addToCart()
{
```

```
$this->waitForElement(self::ADD_TO_CART_SELECTOR)
;
  $addToCartButton =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::ADD_TO_CART_SELECTOR));
  $addToCartButton->click();
  $this->logger->info('Added product to cart');
}

private function navigateToCart()
{

$this->waitForElement(self::CART_BUTTON_SELECTOR)
;
  $cartButton =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::CART_BUTTON_SELECTOR));
  $cartButton->click();
  $this->logger->info('Navigated to cart');
}

private function proceedToCheckout()
{

$this->waitForElement(self::CHECKOUT_BUTTON_SELEC
TOR);
  $checkoutButton =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::CHECKOUT_BUTTON_SELECTOR));
  $checkoutButton->click();
```

```
  $this->logger->info('Proceeded to checkout');
}

private function handleSignIn()
{

$this->waitForElement(self::SIGN_IN_EMAIL_SELECTO
R);
  $emailField =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SIGN_IN_EMAIL_SELECTOR));
  $emailField->sendKeys(self::TEST_EMAIL);
  $this->logger->info('Entered email');
  $continueButton =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SIGN_IN_CONTINUE_SELECTOR));
  $continueButton->click();


$this->waitForElement(self::SIGN_IN_PASSWORD_SELE
CTOR);
  $passwordField =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SIGN_IN_PASSWORD_SELECTOR));
  $passwordField->sendKeys(self::TEST_PASSWORD);
  $this->logger->info('Entered password');

  $signInButton =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SIGN_IN_SUBMIT_SELECTOR));
  $signInButton->click();
```

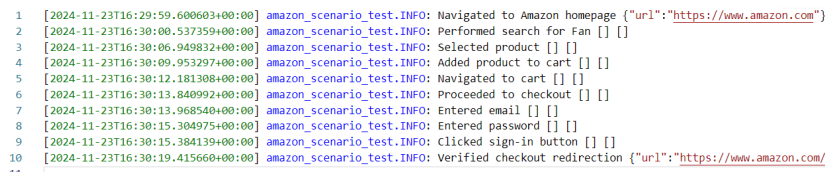
```
  $this->logger->info('Clicked sign-in button');
}

private function verifyCheckoutRedirection()
{
  $this->waitForUrlContains('addressselect');
  $currentURL = $this->driver->getCurrentURL();
  $this->logger->info('Verified checkout
redirection', ['url' => $currentURL]);

$this->assertStringContainsString('addressselect'
, $currentURL, "Failed to reach the checkout
page.");
}
```

Note: We used ChromeDriver server to act as a bridge between selenium and chrome.

```
C:\Users\sdaoc\composer-phpunit-boilertemplate>vendor\bin\phpunit tests/AmazonScenarioTest.php
PHPUnit 11.4.0 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.4.1

.                                                                   1 / 1 (100%)

Time: 00:21.576, Memory: 8.00 MB

OK (1 test, 1 assertion)
```

Result: Test case result for Scenario Testing

The result showcases the results of the test case, it displays the memory usage along with the completion time.



Result: Logs of the Scenario Testing test case

As for the logs, it displays the logs of the test case itself.

5.4 Usability testing

Usability testing is testing a design's usability using a representative sample of users [40]. Testers can apply it to several kinds of design, and it typically entails watching people as they try to finish tasks. From the beginning of a product's development to its release, it is frequently carried out repeatedly. Due to its focus on user satisfaction and user experience, Usability testing primarily falls under User Acceptance Testing. However, it indirectly supports System Testing when ensuring back-end functionality.

Below is the testUsabilityFeatures function, which emulates the actions of a user in the e-commerce platform, Amazon, which are:

1. Searching for a product.
2. Returning to the homepage.
3. Attempting to log in.

```
Unset
public function testUsabilityFeatures()
{
  $startTime = microtime(true);
  $this->driver->get(self::BASE_URL);
  $endTime = microtime(true);
  $this->logger->info('Navigated to Amazon
homepage', [
    'url' => self::BASE_URL,
    'response_time' => $endTime - $startTime
  ]);

  $startTime = microtime(true);

$this->waitForElement(self::SEARCH_BOX_SELECTOR);
  $endTime = microtime(true);
  $this->logger->info('Search box loaded', [
    'response_time' => $endTime - $startTime
  ]);

  $searchBox =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BOX_SELECTOR));
  $this->assertTrue($searchBox->isDisplayed(),
"Search bar is not visible.");
  $searchBox->sendKeys('Laptop');
  $this->logger->info('Entered search term',
['term' => 'Laptop']);

  $startTime = microtime(true);
```

```
  $searchButton =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BUTTON_SELECTOR));
  $searchButton->click();
  $endTime = microtime(true);
  $this->logger->info('Clicked search button', [
    'response_time' => $endTime - $startTime
  ]);

  $startTime = microtime(true);
  $this->waitForUrlContains('Laptop');
  $endTime = microtime(true);
  $currentURL = $this->driver->getCurrentURL();
  $this->logger->info('Search results loaded', [
    'url' => $currentURL,
    'response_time' => $endTime - $startTime
  ]);
  $this->assertStringContainsString('Laptop',
$currentURL, "Search functionality is not working
as expected.");

  $startTime = microtime(true);
  $this->driver->navigate()->back();
  $endTime = microtime(true);
  $this->logger->info('Navigated back to
homepage', [
    'response_time' => $endTime - $startTime
  ]);

  $startTime = microtime(true);
```

```
  $this->waitForElement(self::SIGN_IN_LINK_ID,
'id');
  $endTime = microtime(true);
  $this->logger->info('Sign-in link loaded', [
    'response_time' => $endTime - $startTime
  ]);

  $signInLink =
$this->driver->findElement(WebDriverBy::id(self::
SIGN_IN_LINK_ID));
  $this->assertTrue($signInLink->isDisplayed(),
"Sign-in link is not visible.");
  $signInLink->click();
  $this->logger->info('Clicked sign-in link');

  $startTime = microtime(true);
  $this->waitForUrlContains('signin');
  $endTime = microtime(true);
  $currentURL = $this->driver->getCurrentURL();
  $this->logger->info('Navigated to sign-in
page', [
    'url' => $currentURL,
    'response_time' => $endTime - $startTime
  ]);
  $this->assertStringContainsString('signin',
$currentURL, "Failed to navigate to the login
page.");

  $startTime = microtime(true);
  $this->waitForElement(self::EMAIL_FIELD_ID,
'id');
```

```
  $endTime = microtime(true);
  $emailField =
$this->driver->findElement(WebDriverBy::id(self::
EMAIL_FIELD_ID));
  $this->assertTrue($emailField->isDisplayed(),
"Email input field is not visible.");
  $this->logger->info('Email input field is
visible', [
    'response_time' => $endTime - $startTime
  ]);
}
```

Note: We used ChromeDriver server to act as a bridge between selenium and chrome.

In the test case context covering e-commerce, the emphasis on usability testing allows testers to evaluate critical workflows and user interactions. For the website that was used for the test case, which is Amazon's website, this might involve validating features such as:

- The search bar ensures users can find desired products effortlessly.
- The login workflow to confirm users can access their accounts without issues.
- The ability to navigate back to the homepage smoothly after completing tasks.

The test case for Amazon usability testing captures these critical aspects, ensuring a consistent and user-friendly experience for customers. Since PHPUnit is not designed to test the direct usability of UIs, it instead plays a supportive role by validating the back-end processes (e.g., API validation tests).

The command line interface successfully executes the test case, displaying the memory use and completion time, which are:

```
C:\Users\sdaoc\composer-phpunit-boilertemplate>vendor\bin\phpunit tests/AmazonUsabilityTest.php
PHPUnit 11.4.0 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.4.1

.                                                                   1 / 1 (100%)

Time: 00:09.109, Memory: 8.00 MB

OK (1 test, 5 assertions)
```

Result: Test case result for Usability Test

To record important user interactions and backend answers, the test case creates logs. An illustration of the logging structure that shows how events are noted and examined during usability testing may be found in the the result below:

```
[2024-11-23T08:44:49.751210+00:00] amazon_usability_test.INFO: Navigated to Amazon homepage {"url":"https://www.ama
[2024-11-23T08:44:49.798615+00:00] amazon_usability_test.INFO: Search box loaded {"response_time":0.044425010681152
[2024-11-23T08:44:49.983967+00:00] amazon_usability_test.INFO: Entered search term {"term":"Laptop"} []
[2024-11-23T08:44:50.524268+00:00] amazon_usability_test.INFO: Clicked search button {"response_time":0.54015517234
[2024-11-23T08:44:53.330371+00:00] amazon_usability_test.INFO: Search results loaded {"url":"https://www.amazon.com
[2024-11-23T08:44:53.370270+00:00] amazon_usability_test.INFO: Navigated back to homepage {"response_time":0.038954
[2024-11-23T08:44:53.908141+00:00] amazon_usability_test.INFO: Sign-in link loaded {"response_time":0.5373909473419
[2024-11-23T08:44:55.483233+00:00] amazon_usability_test.INFO: Clicked sign-in link [] []
[2024-11-23T08:44:55.509451+00:00] amazon_usability_test.INFO: Navigated to sign-in page {"url":"https://www.amazon
[2024-11-23T08:44:55.573647+00:00] amazon_usability_test.INFO: Email input field is visible {"response_time":0.0271
```

Result: Logs of the Usability Test test case

It displays the inclusion of logging in the test case in the following ways:

1. Action Monitoring:

- Necessary user actions (e.g., going to the sign-in page) are recorded in logs. This makes it easier to confirm that the anticipated actions are taking place as intended.

2. Debugging and Troubleshooting:
   - Logs offer comprehensive information about what occurred behind the scenes in the event of a usability issue (such as a page failing to load), which aids in identifying and resolving problems.

PART 6: Special Tests

Special Testing in PHPUnit are specialized testing techniques designed to verify specific aspects of software quality beyond basic functionality. These tests focus on validity, performance, reliability, and system stability under various conditions.

6.1 Smoke Testing

Smoke testing is an integration testing approach that is commonly used when “shrink wrapped” software products are being developed [41]. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis [41].

The smoke testing is simply a shallow testing of critical paths. It validates that the core functionality of a system works as intended. Smoke testing is set in motion after the initial deployment of the application on the internet. To illustrate, suppose we want to verify that the endpoint for logging in to the GitHub website is working, that is,

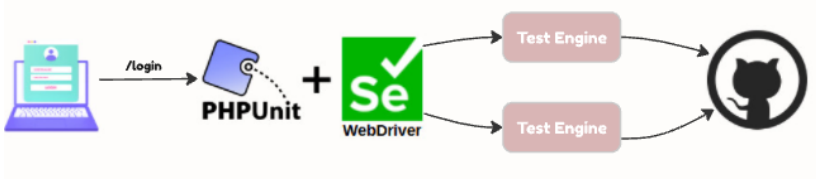


Figure 14: Endpoint verification

Note: We used the PHP-WebDriver binding the Selenium WebDriver.

Writing this test case in PHP programming language using PHPUnit as a software testing tool, that is,

```
Unset
public function testGitHubLoginForm()
{
  // Navigate to GitHub login page
  $this->driver->get('https://github.com/login');

  // Test username field
  $loginField =
$this->driver->findElement(WebDriverBy::id('login
_field'));
  $this->assertTrue($loginField->isDisplayed());
  $this->assertEquals('login_field',
$loginField->getAttribute('id'));

  // Test password field
```

```
  $passwordField =
$this->driver->findElement(WebDriverBy::id('passw
ord'));

$this->assertTrue($passwordField->isDisplayed());
  $this->assertEquals('password',
$passwordField->getAttribute('type'));

  // Test sign-in button
  $signInButton =
$this->driver->findElement(WebDriverBy::cssSelect
or('input[type="submit"]'));

$this->assertTrue($signInButton->isDisplayed());
  $this->assertEquals('Sign in',
$signInButton->getAttribute('value'));

  // Test forgot password link
  $forgotPasswordLink =
$this->driver->findElement(WebDriverBy::linkText(
'Forgot password?'));

$this->assertTrue($forgotPasswordLink->isDisplaye
d());

  // Test page title
  $this->assertEquals('Sign in to GitHub ·
GitHub', $this->driver->getTitle());
}
```

Note: We used ChromeDriver server to act as a bridge between selenium and chrome.

Running the test case results in success because of the fact that the login of GitHub is properly configured.

```
PS C:\Users\loudi\Desktop\softwaretest> vendor\bin\phpunit tests/GitHubLoginSmokeTest.php
PHPUnit 11.4.3 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12

D                                                                   1 / 1 (100%)

Time: 00:04.769, Memory: 8.00 MB

OK, but there were issues!
Tests: 1, Assertions: 8, Deprecations: 1.
```

Result: GitHubLoginSmokeTest when executed

Note: We used composer as it is a dependency manager for PHP programming language to install the PHPUnit.

6.2 Performance Testing

Performance testing is designed to test the run-time performance of software within the context of an integrated system [41]. It tests for conforming to basic performance requirements [42].

Performance testing measures the application responsiveness and stability under normal conditions. It centers around speed, scalability, and resource utilization. Typically, its metrics are response time, throughput, CPU/memory usage. Suppose we want to measure the performance of loading the homepage of GitHub website and record the result, that is,

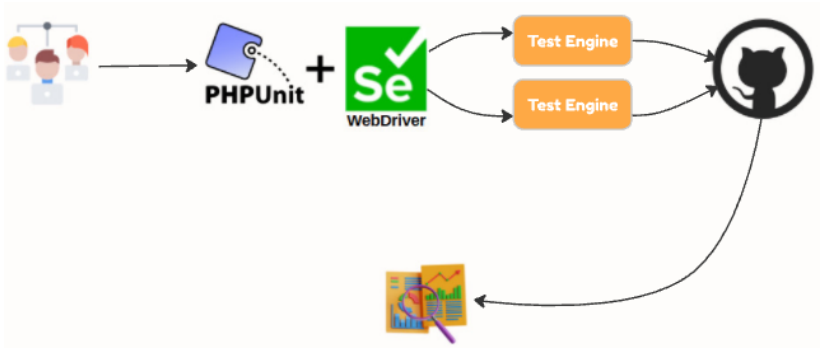


Figure 15: Performance metrics process

Note: We used the PHP-WebDriver binding the Selenium WebDriver.

Translating this in PHP programming language, that is,

```
Unset
public function testGitHubLoginForm()
{
  // Navigate to GitHub login page
  $this->driver->get('https://github.com/login');

  // Test username field
  $loginField =
$this->driver->findElement(WebDriverBy::id('login
_field'));
  $this->assertTrue($loginField->isDisplayed());
  $this->assertEquals('login_field',
$loginField->getAttribute('id'));

  // Test password field
  $passwordField =
$this->driver->findElement(WebDriverBy::id('passw
ord'));

$this->assertTrue($passwordField->isDisplayed());
  $this->assertEquals('password',
$passwordField->getAttribute('type'));

  // Test sign-in button
  $signInButton =
$this->driver->findElement(WebDriverBy::cssSelect
or('input[type="submit"]'));

$this->assertTrue($signInButton->isDisplayed());
  $this->assertEquals('Sign in',
$signInButton->getAttribute('value'));

  // Test forgot password link
  $forgotPasswordLink =
$this->driver->findElement(WebDriverBy::linkText(
'Forgot password?'));

$this->assertTrue($forgotPasswordLink->isDisplaye
d());

  // Test page title
  $this->assertEquals('Sign in to GitHub ·
GitHub', $this->driver->getTitle());
}
```

Note: We used ChromeDriver server to act as a bridge between selenium and chrome.

Running the test case in command line interface results a success displaying the time it takes to finish and memory usage, that is,

```
PS C:\Users\loudi\Desktop\softwaretest> vendor\bin\phpunit tests/GitHubHomePagePerformanceTest.php
PHPUnit 11.4.3 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12

W                                                                   1 / 1 (100%)

Time: 00:04.131, Memory: 8.00 MB

OK, but there were issues!
Tests: 1, Assertions: 3, Warnings: 1, Deprecations: 1.
```

Result: Performance testing test case execution

Note: We used composer as it is a dependency manager for PHP programming language to install the PHPUnit. Additionally, it saves the performance metric in the .json file. The metrics are the initial and DOM (Document Object Model) load time, memory usage, and the overall time.

output > {} homepage_performance_report.json >

```json
{
    "initial_load": 2.75,
    "dom_ready": 0.08,
    "memory_usage": 5.72,
    "total_time": 3.19
}
```

Result: Saved performance metrics in JSON

6.3 Stress Testing

Stress tests are designed to confront programs with abnormal situations [41]. Stress testing intentionally creates high loads of requests to validate whether an APIs function as expected [43]. It assesses the behavior of a system under extreme conditions. It challenges the breaking points and recovery of an application. This pushes the application/system normal operational capabilities. To give an example, suppose we want to evaluate the load time of the login page of GitHub website and log the result, that is,

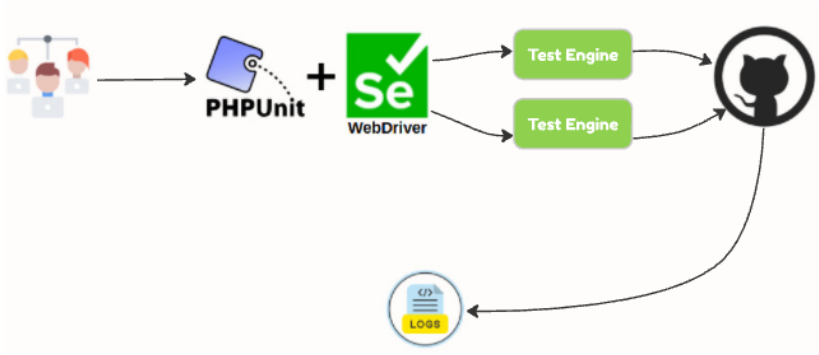


Figure 16: Evaluating log in load time process

Note: We used the PHP-WebDriver binding the Selenium WebDriver.

Transforming this in PHP programming language, that is,

Unset

```php
public function
testLoginPageLoadTime($iterations)
{
  $loadTimes = [];
```

```
  for ($i = 0; $i < $iterations; $i++) {
    $start = microtime(true);

    try {
      $this->driver->get(self::LOGIN_URL);
      $loadTime = microtime(true) - $start;
      $loadTimes[] = $loadTime;

      $this->assertLessThan(
        self::MAX_RESPONSE_TIME,
        $loadTime,
        "Response time exceeded threshold on
iteration {$i}"
      );

      $this->logMetrics($i, $loadTime);

$this->driver->manage()->deleteAllCookies();
      sleep(1); // Prevent rate limiting

    } catch (\Exception $e) {
      $this->fail("Iteration {$i} failed: " .
$e->getMessage());
    }
  }

  $this->outputStats($loadTimes);
}
```

```
public static function loadTestDataProvider():
array
{
  return [
    'small load test' => [10],
    'medium load test' => [50],
    'large load test' => [100]
  ];
}
```

Running the file showcases statistics of the .log file.

```
PS C:\Users\loudi\Desktop\softwaretest> vendor\bin\phpunit tests/GitHubStressTest.php
PHPUnit 11.4.3 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12


Test Statistics:
Avg Response: 0.32s
Max Response: 1.93s
D
Test Statistics:
Avg Response: 0.17s
Max Response: 1.94s
D
Test Statistics:
Avg Response: 0.15s
Max Response: 1.92s
D                                                                   3 / 3 (100%)

Time: 03:15.420, Memory: 8.00 MB

OK, but there were issues!
Tests: 3, Assertions: 160, Deprecations: 1, PHPUnit Deprecations: 1.
```

Result: Stress testing test case execution

Note: We used composer as it is a dependency manager for PHP programming language to install the PHPUnit.

output > stress_test_metrics.log

```
{"iteration":0,
"timestamp":
"2024-11-21 16:01:13",
"response_time":2.618,
"memory_usage":8388608}
{"iteration":1,
"timestamp":
"2024-11-21 16:01:14",
"response_time":0.171,
"memory_usage
":8388608}
```

Result: Log file of the stress test

6.4 Reliability Testing

Reliability testing considers the system stability over extended periods of time [44]. It focuses on consistency and non-error operation in the production environment. This type of testing usually runs for hours, days, or even weeks in order to verify that there is no such memory leakage. Let's assume that we want to check the reliability of the login page of GitHub website for 1 hour, 4 hours, and 8 hours and save the result as .json, that is,

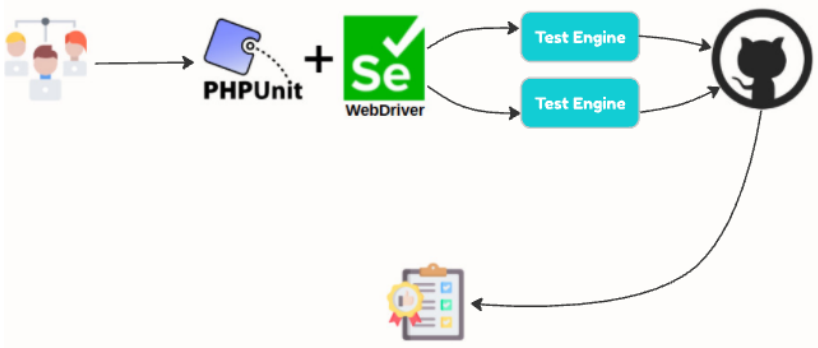


Figure 17: Reliability testing process

Note: We used the PHP-WebDriver binding the Selenium WebDriver.

Converting this in PHP programming language, that is,

Unset

```
/**
 * Performs reliability testing of GitHub login
page for a specified duration
 **/
public function
testLoginPageReliability($duration)
{
  $startTime = microtime(true);
  $errors = [];
  $checkResults = [];

  echo "\n[" . date('Y-m-d H:i:s') . "] Starting
reliability test for {$duration} seconds of
GitHub login page\n";
```

```
  while (time() - $startTime < $duration) {
    try {
      echo ".";
      $iterationStart = microtime(true);

      $this->driver->get(self::LOGIN_URL);
      $this->assertElementsPresent();
      $this->testFormValidation();

      $responseTime = microtime(true) -
$iterationStart;
      echo sprintf("\nResponse time: %.2fs",
$responseTime);

      $checkResults[] = [
        'timestamp' => date('Y-m-d H:i:s'),
        'status' => 'success',
        'response_time' => $responseTime
      ];

    } catch (\Exception $e) {
      echo "\n[ERROR] " . $e->getMessage() .
"\n";

      $errors[] = [
        'timestamp' => date('Y-m-d H:i:s'),
        'error' => $e->getMessage()
      ];
    }
```

```
    $this->driver->manage()->deleteAllCookies();

    // Show progress
    echo "\nCompleted: " . round(((time() -
$startTime) / $duration) * 100) . "%";
    sleep(self::CHECK_INTERVAL); // One test per
minute
  }

  echo "\nTest completed. Duration: {$duration}
seconds\n";

  // Generate reliability report
  $this->generateReport($checkResults, $errors,
$duration);

  // Assert reliability threshold
  $reliability = (count($checkResults) -
count($errors)) / count($checkResults) * 100;
  $this->assertGreaterThan(99, $reliability,
"Reliability below 99%");
}
```

Note: We used ChromeDriver server to act as a bridge between selenium and chrome.

Note: The standard formula for operation based reliability used in the snippet code is,

```
Unset
The operational reliability (R) is calculated as:
R = (S/N) × 100%
where S is the number of successful operations
and N is the total number of operations.
```

Operation-based measures success rate of repeated operations. This is appropriate because we are attempting to evaluate the success rate of repeated login on the GitHub website.

Running the file outputs the response time of the GitHub login page and generates a reliability report in .json file, that is,

```
PS C:\Users\loudi\Desktop\softwaretest> vendor\bin\phpunit tests/GitHubReliabilityTest.php
PHPUnit 11.4.3 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12


[2024-11-22 22:35:19] Starting reliability test for 3600 seconds of GitHub login page
.
Response time: 2.96s
Completed: 0%.
Response time: 1.10s
Completed: 2%.
Response time: 2.41s
Completed: 4%.
Response time: 1.51s
Completed: 5%.
Response time: 1.28s
Completed: 7%.
Response time: 1.47s
Completed: 9%.
Response time: 1.27s
Completed: 10%.
Response time: 1.18s
```

Result: GitHub login page 1 hour of reliability testing.

output > {} reliability_test_report.json > ...


```
{
    "test_duration": {
        "seconds": 3600,
        "hours": 1,
        "formatted": "1 hour(s)"
    },
    "total_checks": 59,
    "successful_checks": 59,
    "errors": 0,
    "reliability_rate": 100,
    "average_response_time": 1.363893169467732,
    "error_details": []
}
```


Result: Test result of the GitHub login page reliability testing for 1 hour.

```
[2024-11-22 23:35:43] Starting reliability test for 14400 seconds of GitHub login page
.
Response time: 2.85s
Completed: 0%.
Response time: 1.29s
Completed: 0%.
Response time: 1.48s
Completed: 1%.
Response time: 1.33s
Completed: 1%.
Response time: 1.46s
Completed: 2%.
Response time: 1.49s
Completed: 2%.
Response time: 1.29s
Completed: 3%.
Response time: 1.34s
Completed: 3%.
Response time: 1.18s
Completed: 3%.
Response time: 1.34s
Completed: 4%.
Response time: 1.23s
Completed: 4%.
Response time: 1.44s
Completed: 5%.
```

Result: GitHub login page 4 hours of reliability testing.

```
{
    "test_duration": {
        "seconds": 14400,
        "hours": 4,
        "formatted": "4 hour(s)"
    },
    "total_checks": 236,
    "successful_checks": 236,
    "errors": 0,
    "reliability_rate": 100,
    "average_response_time": 1.2305997054455644,
    "error_details": []
}
```


Result: Test result of the GitHub login page reliability testing for 4 hours.

```
[2024-11-23 03:36:42] Starting reliability test for 28800 seconds of GitHub login page
.
Response time: 2.62s
Completed: 0%.
Response time: 1.29s
Completed: 0%.
Response time: 1.16s
Completed: 0%.
Response time: 1.00s
Completed: 1%.
Response time: 1.23s
Completed: 1%.
Response time: 1.02s
Completed: 1%.
Response time: 1.01s
Completed: 1%.
Response time: 1.23s
Completed: 1%.
Response time: 0.98s
Completed: 2%.
```

Result: GitHub login page 8 hours of reliability testing.


```
    "test_duration": {
        "seconds": 28800,
        "hours": 8,
        "formatted": "8 hour(s)"
    },
    "total_checks": 470,
    "successful_checks": 470,
    "errors": 0,
    "reliability_rate": 100,
    "average_response_time": 1.317083111215145,
    "error_details": []
}
```


Result: Test result of the GitHub login page reliability testing for 8 hours.

6.5 Acceptance Testing

Acceptance testing also known as 'alpha testing' is where developers of a system and the client have come in terms of stipulated business requirements. This is the final stage in the testing process before the system is accepted for operational use [23]. The system is tested with data supplied by the system customer rather than with simulated test data [23].

Acceptance testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data [23]. Acceptance testing may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is unacceptable [23]. To illustrate, suppose we have an online shopping application. As a

customer, I want to purchase an item and then receive an order notification, that is,

Figure 18: Online shopping purchasing process

This feature of an online shopping application should work as expected relative to the stipulated business requirements.

PART 7: Test Execution

Test execution is simply the process of running tests to ensure a specific part of the code, a function, or even the whole software works as it ought to. The test planners plan and run various tests to see how the system would work in different situations. When they discover problems, they always notify the developers to correct them. If all tests are successful and the software works properly in every situation, it's ready to be released to users [45] [46].

Effective testing is a must because not only does it verify functionality, but it also captures defects or failures that may have been missed or seen during the development process. By testing each component of the software, testers ensure that defects are detected early so the chances of defects arriving in front of the end-users are minimal [47].

We have installed PHPUnit using **Composer**, **WSL** (Windows Subsystem for Linux), and downloaded it **manually**. We managed PHP dependencies using Composer, a tool that centrally deals with the installation of required libraries such as PHPUnit. As we were working a lot with Linux-based tools, we have also used WSL on Windows to make sure the development environment is more uniform. The world of WSL provides a Linux environment directly on Windows without requiring a virtual machine, making it an excellent choice for a scenario in which development platforms need to be bridged. However, while Composer is our preferred method for installing PHPUnit, there are many cases where manual download might be required.

Installation of the PHPUnit using Composer:

1. Download and install Composer from the official website.
2. Once we set up Composer, we can install PHPUnit either globally in your system or locally in your project. To install it locally, run this command:

```
Unset
composer require --dev phpunit/phpunit ^11
```

3. Once PHPUnit has been installed, you can run your tests using the command:

```
Unset
./vendor/bin/phpunit
```

7.1 Test cycles

A test cycle is a set of test cases planned to achieve particular testing goals.

It covers a wider range than individual test cases and can be allocated to specific testers and testing environments. Test cycles can be organized by different criteria, such as functionality, feature, or type of testing [48].

Common examples of test cycles include:

- **Regression Testing**: Ensures that existing functionality is not broken by new code changes.
- **Build Verification Testing (Smoke Testing)**: Checks the stability of a new build by running basic tests to ensure that the most crucial parts of the software are working.
- **End-to-End Testing**: Verifies the complete functionality of the system from the perspective of an end user.

In PHPUnit, test cycles are facilitated through **test suites**, which group related tests into logical collections. This organization helps ensure that specific sets of tests are executed together to meet particular objectives. [49]

Setting up a regression test suite in PHPUnit: Example Configuration (**phpunit.xml**)

```
Unset
<phpunit>
  <testsuites>
    <testsuite name="regression">
      <directory>tests/Regression</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

Note: We made use of WSL(Windows Subsystem for Linux) here.

To run all tests from the test suites, use the following command.

```
Unset
./phpunit
```

To run a specific test suite, use the following command:

```
Unset
./phpunit --testsuite regression
```

7.2 Approaching a new feature

Testing a new feature would involve the verification of functionality, integration with existing components, and robustness under varied conditions. It is done with the intention of ensuring that new features meet requirements without causing problems elsewhere.

In PHPUnit, there are several techniques and testing types that allow for proper testings of new features:

- **Unit Tests**: Testing individual functions or methods in isolation is the focus. It is important to test whether new features would work as expected at the most granular levels.
- **Integration Tests**: Whether the new feature plays well with other parts or modules of the system is a check of an integration test. This will ensure that it fits properly within the larger application.
- **Mocking and Stubbing**: These are techniques to mock external dependencies or components; this way, you isolate the new feature from other system parts when testing. For example, if the new feature relies on a third-party API, then you can mock that API and be able to test the feature without requiring the actual API.

PHPUnit's flexibility allows you to write **unit tests** and **integration tests** efficiently. It also provides you with mocking tools called **MockObject** so that you can simulate dependencies in order to test the feature in isolation.

7.3 Test data

In software testing, test data is the set of input values used to test a given software program so that its functionality, dependability, and performance can be verified. Test data offers various kinds of input types, such as normal, boundary, invalid, error-prone, stressed, and corner case data. While boundary data explores numbers on the peripherals of acceptable ranges in order to identify issues related to boundaries, normal input data represents average inputs by users. [50]

In PHPUnit, **data providers** are used to provide test methods with certain input data sets. It allows you to run the same test with different inputs and expected results without duplicating the test code. This is a very useful feature in methodically testing a wide range of scenarios. [49]

A data provider in PHPUnit is a method that returns an array of data sets, where each data set is used as a set of arguments for running a test method multiple times. This feature is really useful when you want to run the same logic with different inputs, securing the robustness of your code without having to duplicate test methods.

Example: Using a data provider that returns an array of arrays

```
Unset
<?php declare(strict_types=1);
use PHPUnit\Framework\Attributes\DataProvider;
use PHPUnit\Framework\TestCase;

final class NumericDataSetsTest extends TestCase
```

```
{
  public static function additionProvider():
array
  {
    return [
      [0, 0, 0],
      [0, 1, 1],
      [1, 0, 1],
      [1, 1, 3],
    ];
  }

  #[DataProvider('additionProvider')]
  public function testAdd(int $a, int $b, int
$expected): void
  {
    $this->assertSame($expected, $a + $b);
  }
}
```

Running the Command: The command tells PHPUnit to execute the test file *tests/NumericDataSetsTest.php* using the PHPUnit executable located at *./tools/phpunit*

```
Unset
./tools/phpunit tests/NumericDataSetsTest.php
PHPUnit 11.4.0 by Sebastian Bergmann and
contributors.

Runtime:       PHP 8.3.12

...F
4 / 4 (100%)

Time: 00:00.001, Memory: 23.08 MB

There was 1 failure:

1) NumericDataSetsTest::testAdd with data set #3
(1, 1, 3)
Failed asserting that 2 is identical to 3.

/path/to/tests/NumericDataSetsTest.php:20

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
```

7.4 Tracking and reporting test results

Tracking and reporting of test results is very important to help developers and testers identify defects and measure test coverage as well as monitor the quality of the software. [51] PHPUnit provides several ways of tracking and reporting on the outcome of test execution:

- **Console Output:** PHPUnit gives responses directly to the command line. It shows whether a

test is passed or not. If it is not passed, then it shows specific error messages and stack traces. All these enable developers to trace problem spots right away.

- **XML Reports:** PHPUnit can also generate XML reports, which are very useful in particular when integrated with the CI systems - Jenkins. The automated build systems can track the test results and also send out alerts in case tests fail. [52]

- **Code Coverage Reports:** The code coverage shows which parts of the codebase are tested and which are not. PHPUnit can generate code coverage reports in different formats, including HTML and Clover XML, which helps the developer ensure that most of his tests cover critical parts of the application and identify which areas remain untouchable. [53]

These tools allow testers and developers to prioritize issues, measure testing effectiveness, and improve overall software quality over time.

Here is a sample of the console output since it is the one used without additional setups:

```
$ ./phpunit --testsuite unit
PHPUnit 11.4.0 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.2
Configuration: /mnt/c/Users/keanu/Desktop/phpunit/phpunit.xml

.................F.............................................  63 / 172 ( 36%)
.............................FFF.............................. 126 / 172 ( 73%)
.......S.............S.....................                    172 / 172 (100%)

Time: 00:00.213, Memory: 24.27 MB

OK (172 tests, 637 assertions)
```

Result: Execution showing the report

Here are brief definitions for each of the PHPUnit command-line progress indicators:

- . (dot): Printed when a test is completed without any issues or failures-it indicates that the test passed.
- F (Failure): Printed when the assertion fails-there was no matching expected result from the actual output.
- E (Error): Printed when an error occurred during test running. A sort of exception was thrown and not caught within the test method.
- W (Warning): Printed when the test sets off a PHP warning when run. This is generally related to some kind of problem in the code which doesn't necessarily cause a test to fail, but indicates something to watch out for.
- R (Risky): Printed whenever a test is marked as "risky, meaning it might not be trustworthy or is likely to produce inconsistent results. This occurs

when the test is not fully implemented or when the environment is not set up properly.

- D (Deprecation): Printed when the test triggers a deprecation notice. This simply means that the test uses a feature which is currently deprecated and is likely to be removed from future versions of PHP.
- N (Notice): Printed when the test triggers a PHP notice. Notices are less serious than warnings or errors but still note a potential problem or minor flaw in the code.
- I (Incomplete): Printed when the test is marked incomplete. This occurs when a test method is not fully implemented, which is done on purpose, usually using a call to markTestIncomplete().
- S (Skipped): Printed when a test is skipped, either due to an explicit call to markTestSkipped() or because a condition for running the test was not met.

PART 8: Defect Processing

This process involves identifying, recording, prioritizing, and fixing software defects in an efficient manner while tracking them. During test execution in PHPUnit, defects frequently arise due to a failed test case, indicating that the expected result has not been achieved. Efficient defect processing will ensure that these issues are taken care of for proper maintenance of the quality of the software. [54]

8.1 Defect life cycles

The defect life cycle is also known as bug life cycle. It is the cycle in which a defect travels from its identification till it gets resolution. Hence, it ensures that defects are tracked and managed systematically [55][56]. Some stages, in general, include:

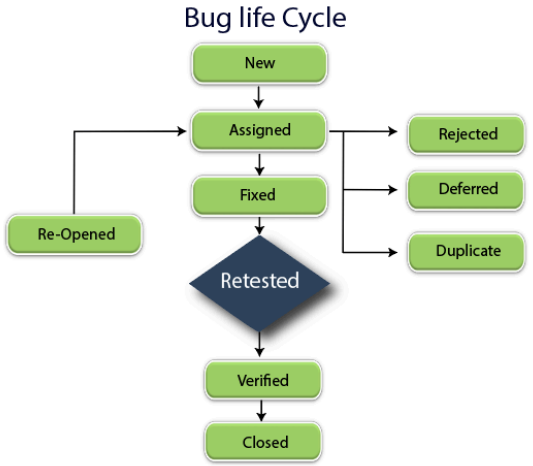


Figure 19: Bug Life Cycle [56]

1. **New:** The bug is identified and logged into the defect tracking system. A unique ID will be assigned to it for tracking, and it waits to be reviewed by the project team [55] [56].
2. **Assigned**: A team lead or project manager assigns it to a developer or specific team for resolution [55][56].
3. **Rejected**: A bug is marked as rejected if the reported issue is not valid or reproducible. Some

common reasons include invalid bugs, expected behavior, or testing misconfigurations [55] [56].

4. **Deferred**: If the bug is valid and not critical for the current release or milestone, it is simply resolved with the issue being deferred to be resolved in future releases. Mostly used for minor bugs or the ones which require a lot of changes [55] [56].
5. **Fixed**: The developer resolves the bug and changes the status of it to fixed, showing it has been implemented [55] [56].
6. **Retested**: The testing team verifies the correction by retesting the bug to the same reported conditions. If the defect persists, then the bug enters the reopened status [55] [56].
7. **Reopened**: After retesting also indicates an issue still prevails, the bug is moved to the reopened status and reassigned for resolution [55] [56].
8. **Duplicate**: In the case that the bug reported is an exact copy of an issue present in the defect tracking system, it is tagged as a duplicate. Duplicates are referenced to the original bug [55] [56].
9. **Verified**: In case the fix, when validated during retest passes with success then the bug is tagged verified, indicating that the bug itself is cured [55] [56].
10. **Closed**: The bug is officially closed, if the project team verifies that the bug no longer exists and all parties affected by it agree with the resolution [55] [56].

8.2 Severity and priority of defects

Severity and priority are critical attributes for classifying defects and determining their impact on the software:

- **Severity**: Reflects the impact of the defect on the application. Common levels include:
  - *Critical*: Prevents the system from functioning (e.g., crashes) [57].
  - *Major*: Causes significant functional issues [57].
  - *Minor*: Causes limited impact on functionality [57].
  - *Trivial*: Cosmetic issues or minor inconveniences [57].
- **Priority**: Determines the urgency of resolving the defect. Common levels include:
  - *High*: Requires immediate attention as it affects key functionalities [57].
  - *Medium*: Needs to be fixed but does not impede critical operations [57].
  - *Low*: Can be resolved in later releases [57].

  Understanding the distinction between severity and priority helps development teams allocate resources effectively [57].

8.3 Writing good defect reports

Clear and comprehensive defect reports are essential for effective communication between testers and developers [56] [58]. A good defect report should include:

1. **Unique Identifier**: A distinct ID for tracking. [56]
2. **Summary**: A concise title describing the defect [56] [58].
3. **Description**: A detailed explanation of the issue, including steps to reproduce, expected results, and actual results [56] [58].
4. **Severity and Priority**: Classification to indicate the defect's importance and urgency [56] [58].
5. **Environment Details**: Information about the testing environment (e.g., OS, browser version, application build) [56] [58].
6. **Attachments**: Screenshots, logs, or video recordings to provide additional context [56].

Example:

Title: "Login button unresponsive on mobile devices"
Steps to Reproduce:

1. Open the app on a mobile browser.
2. Enter valid credentials.
3. Tap the login button.

Expected Result: The user logs in successfully.
Actual Result: The button is unresponsive.

8.4 Efficient use of defect tracking tools

Defect tracking tools streamline the defect management process by providing a centralized platform for logging, tracking, and resolving issues [59]. Key practices for efficient use include:

1. **Standardization**: Define consistent guidelines for logging and updating defects [59].
2. **Integration**: Use tools that integrate seamlessly with other project management and testing platforms [59].
3. **Customization**: Leverage filters, tags, and dashboards to prioritize and monitor critical issues [59].
4. **Collaboration**: Enable real-time communication between testers and developers [59].

Popular defect tracking tools include ***JIRA***, ***Bugzilla***, ***Redmine***, and ***Trello***, each offering unique features to support defect management. Selecting the right tool depends on team size, workflow, and project requirements [59].

PART 9: Test Automation

9.1 Test automation approaches

Testing Automation is a process in which software applications are used to run a previously designed test case, check whether the expected results were achieved and notify the user about any errors found in the process, decreasing the need for human effort. This technique is highly useful in performing testing work where the same test has to be executed many times accurately and without errors. There are considerable advantages in using tools such as PHPUnit in this respect [60].

One of the most significant advantages of test automation especially with the aid of PHPUnit is reduction of human effort in repetitive testing tasks

especially those that require high level of accuracy and efficiency. In the case of web applications, for example, such systems are regularly updated and most recently done updates might conflict with old functionalities, say before and after login works or even the databases. Testing those features over and over again using human beings is bound to take so much time and contain a lot of errors. PHPUnit addresses this issue whereby the testers have to try to find the bugs over and over again by writing and running automated tests that can be reused.

Data-driven testing is also an essential aspect in the use of PHPUnit whereby a test script is able to check the function with many input datasets. For instance, in order to automatically check for all edge cases for testing login features one will test both right and wrong combinations of credentials. Functionality testing with the use of loaded mock objects is also possible in PHPUnit whereby developers can create mock API objects for other systems or databases to enable testing of components in a standalone manner. This is particularly advantageous since it allows for testing of units in isolation without having the need for the systems that possibly surround them. In addition, tests that resort to the use of PHPUnit can be easily incorporated in a continuous integration environment which allows for every build cycle to always incorporate the running of automated tests thus feedback on code quality is very quick. These methods limit the amount of energy expended on the manual work and greatly improve the efficiency and scope of testing software [61].

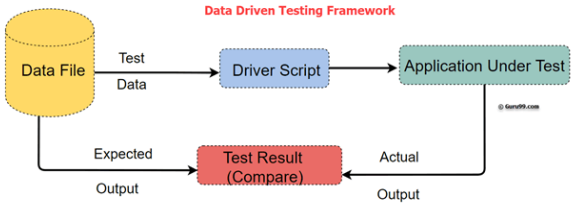


Figure 20: Data Driven Testing Framework

Automation testing means the use of tools such as PHPUnit to run the already created test cases, check the results against the expected ones, and log the differences which in turn reduces the human effort towards performing tedious and repetitive tasks. One practical example is the testing of the transactional handling capabilities of a banking application. For example, the “Transfer Funds” feature. It is imperative that transfer operations work well across all scenarios including valid account numbers, wrongly formatted account numbers, and even when an attempt is made to transfer zero or negative values. Utilizing PHPUnit, it is possible to implement a trial case to check the ability in functionality, that is,

```
Unset
/**
 * TransactionServiceTest tests the
 * TransactionService functionality
```

```
 *
 * @covers \App\TransactionService
 */
class TransactionServiceTest extends TestCase
{

  /**
   * Tests the processTransfer method with
various input scenarios
   *
   * @dataProvider transferDataProvider
   * @param array $input The transfer input data
containing 'from', 'to' and 'amount'
   * @param string $expected Expected result
('success' or 'error')
   * @param string $message Test assertion
message
   */

  #[DataProvider('transferDataProvider')]
  public function testProcessTransfer(array
$input, string $expected, string $message)
  {
    $transactionService = new
TransactionService();
    $this->assertEquals($expected,
$transactionService->processTransfer($input),
$message);
  }

  /**
```

```
   * Data provider for transfer test cases
   *
   * @return array Test cases with the following
structure:
   *               - input: array with 'from',
'to' and 'amount' keys
   *               - expected: string ('success'
or 'error')
   *               - message: string (test
description)
   */
  public static function transferDataProvider():
array
  {
    return [
      'successful_transfer' => [/* ... */],
      'zero_amount' => [/* ... */],
      'empty_from_account' => [/* ... */],
      'empty_to_account' => [/* ... */]
    ];
  }
}
```

Running the test file gives a success test case,

```
PS C:\Users\loudi\Desktop\softwaretest> vendor\bin\phpunit tests/TransactionServiceTest.php
PHPUnit 11.4.3 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12
Configuration: C:\Users\loudi\Desktop\softwaretest\phpunit.xml

.........                                                         9 / 9 (100%)

Time: 00:00.167, Memory: 8.00 MB

OK (9 tests, 9 assertions)
```

Result: TransactionServiceTest when executed

Here, we can use the data provider of PHPUnit as the unit testing is done with more than one values for example, valid transfers, invalid transfers like cases with missing account numbers or incorrect amounts amongst others, without code duplication. The anticipated output has results such as 'success' for valid values and 'error' for invalid values. By doing so, this testing part is made operative, allowing for quick and precise executing of repetitive verification processes and for upholding the quality with every alteration of the code. Furthermore, with the help of mock objects embedded in PHPUnit, it is possible to check transaction logic in isolation by faking outputs of third parties systems like account validation services [62].

9.2 Automation process

The start of the automation process entails correlating the actions involved in implementation with the strategy discussed in 9.1. For example, if data-driven testing is the methodological choice, the emphasis shifts towards testing several cases by altering input data sets. In the same vein, mocking and stubbing tactics focus on isolating the particular unit being tested from the external usage by creating 'fake' dependencies while regression testing is concerned with how the automated tests become part of continuous integration/continuous delivery (CI/CD) pipeline. The first stage in this sequence of events is proficiency in defining automation objectives. Covering things such as specifying the parameters of success enhancements of coverage, decrease of manual work, and quick turnaround time [63].

Having selected the approach, automated test cases are then created. In case of case driven testing, two or more different inputs can be provided to a single test case using data providers available in the PHPUnit allowing wider coverage and avoiding code duplication. For example, a functional login might be validated through data providers with varying degrees of validity in the user credentials for purposes of gaining access to the system. With regards to mocking, the MockObject framework of Unit Test works great in allowing testing of classes by creating fakes of an external service or an API and sub testing only the unit in question, where the response is predictable [64].

The code below represents an example of using the MockObject framework and Data providers, that is,

```
Unset
public function testUserNotification()
```

```
{
  $mailer = $this->createMock(Mailer::class);
  $mailer->expects($this->once())
         ->method('send');
  $service = new UserService($mailer);
  $service->notifyUser();
}

/** @dataProvider userDataProvider */
public function testUserValidation($input,
$expected)
{
  $result = $this->user->validate($input);
  $this->assertEquals($expected, $result);
}
```

As with any automation, debugging and reporting form a central part of the entire process. Given the framework of PHP, PHPUnit issues elaborate error reports alongside code coverage statistics that help search for testing inadequacies and trouble spots. The tests are kept up to date through maintenance refactoring during which they are modified according to how the application evolves, thus minimizing maintenance costs in the long run. Embedding these tests in a CI/CD pipeline supports their execution on a continuous basis thus supplying developers with feedback almost instantaneously and enhancing the quality of the code over time. This proper way of doing things incorporates automation, increases productivity, diminishes manual work, and takes care of the software reliability [65].

9.3 Unit test frameworks

PHPUnit is a unit testing framework specifically designed for the PHP programming language. It is an instance of the xUnit architecture for unit testing frameworks originated with SUnit and became popular with JUnit. PHPUnit was created by Sebastian Bergmann and its development is hosted on GitHub. The core components of PHPUnit is the systematic structure of test cases which is crucial in terms of larger scale projects.

Here is the typical structure of building a test cases in PHPUnit, that is,

```
Unset
<?php
abstract class BaseTestCase extends TestCase
{
  protected function setUp(): void
  {
    parent::setUp();
    $this->initializeTestEnvironment();
  }

  protected function initializeTestEnvironment():
void
  {
    // Set up test dependencies
  }
}
```

As mentioned earlier, PHPUnit is optimized for larger projects. This means that organization of tests is the top priority in either test-driven development or behavior-driven development. A typical directory structure for a PHP project looks like this:

```
Unset
- public
- src
- tests
  - unit
  - integration
  - end-to-end
  bootstrap.php
- phpunit.xml
- composer.json
```

The phpunit.xml involves configuration files for running PHPUnit at various options, that is,

```
Unset
<phpunit bootstrap="tests/bootstrap.php">
  <testsuites>
    <testsuite name="unit">
      <directory>tests/Unit</directory>
    </testsuite>
  </testsuites>
  <coverage>
    <include>
      <directory>src</directory>
    </include>
  </coverage>
</phpunit>
```

9.4 Automation of regression testing

Regression testing is one of the most common types of testing in software development. It requires going back, or "regressing," to existing code and ensuring it isn't negatively affected whenever new functionality, features, or updates are added [66].

Regression testing is the building blocks of continuous integration/continuous development (CI/CD). An example of CI/CD is presented in 2.4 Testing pipelines showcasing an automatic test after each code has been committed on a repository. However, to illustrate further, let's reuse the same repository together with the ci-php73.yml file. This can be found under,

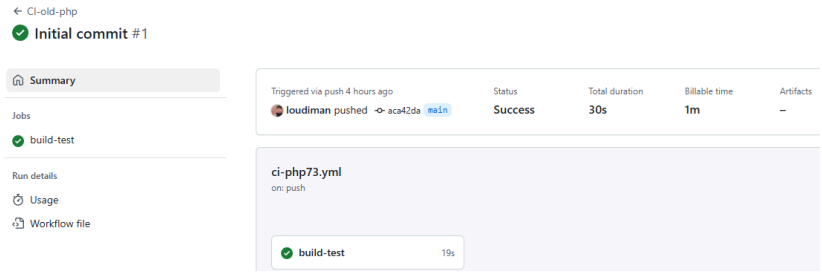
```
Unset
- .github
  - workflows
    - ci-php73.yml (contains configuration file
to run CI/CD tests on a codebase using older
version of PHP (PHP7.3)
```

The code within ci-php73.yml file looks like this,

```
Unset
name: CI-old-php

on: [push]

jobs:
  build-test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - uses: php-actions/composer@v6

      - name: PHPUnit Tests
        uses: php-actions/phpunit@v3
        env:
          TEST_NAME: Scarlett
        with:
          bootstrap: vendor/autoload.php
          configuration: test/phpunit.xml
          args: --coverage-text
          version: 8
          php_version: "7.3"
```

The essence of this workflow configuration is to guarantee that the codebase is compatible with PHP 7.3 whenever code is pushed to the repository.

Assuming that we pushed changes in the repository the ci-php73.yml executes and runs automated tests, that is,



Result: Test result summary of the execution of ci-php73.yml on GitHub

9.5 Minimizing test automation maintenance

Maintaining test automation encounters numerous challenges that can influence its efficiency and effectiveness. Among the issues in this process are fragile test scripts, where even slight alterations in the application under test can lead to breakage, demanding extensive updates and increased maintenance efforts.

Minimizing test automation maintenance is crucial for ensuring a reasonable cost leading to a sustained efficiency and effectiveness of automated testing processes. In the context of PHPUnit as a software testing automation tool, this can be accomplished by following best practices. This has a tendency to result in complexity in the initial stage but the outcome solidifies the cost-reduction of test automation maintenance.

The following are strategies to minimize test automation maintenance, that is,

1. Framework Architecture Best Practices - Utilizing common framework architecture for a system is the usual approach rather than reinventing the wheel.

```
Unset
project/
├── src/
├── tests/
│   ├── TestCase.php
│   ├── Traits/
│   ├── DataProviders/
│   └── Utilities/
├── config/
└── phpunit.xml
```

2. Recommended Tools & Libraries - Employing dependencies that are widely established is better than nothing.

```
Unset
{
    "require-dev": {
        "php-webdriver/webdriver": "^1.1",
        "phpunit/phpunit": "^11.4",
        "phpunit/php-code-coverage": "^11.0"
}
```

3. Documentation Requirements - A good documentation requirement improves the quality of a software product.

```
Unset
namespace Tests\Feature;

class UserTest extends TestCase
{
  use DatabaseTrait;

  /** @test */
  public function it_can_create_user(): void
  {
    $userData =
TestDataProvider::loadTestData('user');

    $response = $this->post('/api/users',
$userData);

    $response->assertStatus(201);
    $this->assertDatabaseHas('users', $userData);
  }
}
```

PART 10: Appendices
List of References

[1] Beizer, B. (1990). Software Testing Techniques (2nd ed.). Van Nostrand Reinhold.

[2] Sophia Ellis (2023, September 11). What are the Goals and Objectives of Software Testing? Explore from https://www.theknowledgeacademy.com/blog/objectives-of-software-testing

[3] Robert T. Hughes, (2003) quality of software, from https://www.sciencedirect.com/topics/computer-science/quality-of-software

[4] Westland, J. (2024, November 7). The Triple Constraint in Project Management: Time, Scope & Cost. ProjectManager. From https://www.projectmanager.com/blog/triple-constraint-project-management-time-scope-cost

[5] Swail, P. (2020, October 8). The testing trade-off triangle. Serverless First. From https://serverlessfirst.com/testing-tradeoff-triangle

[6] Sharma, L. (2024, November 22). What is the Difference between Error, Defect, and Failure? TOOLSQA. From https://toolsqa.com/software-testing/istqb/error-defect-failure

[7] Shiva, (Mar 27, 2924) Difference Between Bug, Defect, Error, Fault and Failure From https://www.naukri.com/code360/library/difference-between-bug-defect-error-fault-and-failure

[8] Admin, & Admin. (2024, September 26). Difference between Error, Fault and Failure. TestingDocs.com. From https://www.testingdocs.com/difference-between-error-fault-and-failure

[9] Cser, T. (2023, February 12). The cost of finding bugs later in the SDLC From https://www.functionize.com/blog/the-cost-of-finding-bugs-later-in-the-sdlc

[10] Geeksforgeeks (2024, September, 26) What is Software Testing https://www.geeksforgeeks.org/software-testing-basics/

[11] Test Yandra Global (2024,August 4) Software Testing Life Cycle : Complete Guide https://www.linkedin.com/pulse/software-testing-life-cycle-complete-guide-test-yantra-global-qicpc

[12] Nestify (2023, June 30) PHP Unit Testing Using PHPUnit Framework: A Comprehensive Guide https://nestify.io/blog/php-unit-testing-using-phpunit-framework/

[13] GeeksforGeeks. (2024, November 20) Software Development Life Cycle (SDLC) https://www.geeksforgeeks.org/software-development-life-cycle-sdlc/

[14] Jacobs, J. (2007, July) Identification of factors that influence defect injection and detection in development of software intensive products https://www.sciencedirect.com/science/article/abs/pii/S0950584906001200

[15]]Imperva (n. d.) https://www.imperva.com/learn/application-security/php-injection/

[16] Alexkondov. (2022, July 29) A Testing Philosophy https://alexkondov.com/a-testing-philosophy/

[17] Demir, K. (2023, December) The Role of Pipeline in Software Processes: A Software Engineer's Perspective https://medium.com/huawei-developers/the-role-of-pipeline-in-software-processes-a-software-engineers-perspective-c84bf763a55e

[18] Ramesh, R. (2023, September 15) How to Attain Business Success with CI/CD Pipeline Automation Testing https://www.headspin.io/blog/why-you-should-consider-ci-cd-pipeline-automation-testing

[19] Crudu, V. (2024, August 24) How to automate PHP unit testing in a continuous integration pipeline? https://moldstud.com/articles/p-how-to-automate-php-unit-testing-in-a-continuous-integration-pipeline

[20] Castrovillari, T. (2018, September 18) The Deployment Pipeline: The Basis for Successful Software Development https://blogs.itemis.com/en/the-deployment-pipeline-the-basis-for-successful-software-development

[21] Bowler, G., & Clauss, C. (n.d.). An example project that uses php-actions/phpunit. https://github.com/php-actions/example-phpunit

[22] Kent, R. (2016). https://www.sciencedirect.com/book/9780081020821/quality-management-in-plastics-processing. Elsevier. https://www.sciencedirect.com/science/article/abs/pii/B9780081020821500058

[23] Sommerville, I. (1988) Software Engineering 9th Edition by Ian Sommerville

[24] Naik, K., & Tripathy, P. (2008). Software Testing and Quality Assurance: Theory and Practice. John Wiley & Sons, inc.

https://www.softwaretestinggenius.com/download/staqtpsn.pdf

[25] Dooley, K. (1996, July). (PDF) The Dimensions of Software Quality. ResearchGate. Retrieved November 24, 2024, from https://www.researchgate.net/publication/249863335_The_Dimensions_of_Software_Quality

[26] Kratikal. (August 17, 2022). Types of testing techniques: Black, white, and grey box. https://kratikal.com/blog/types-of-testing-techniques-black-white-and-grey-box/

[27] TestFort. (June 9, 2023). Static vs. dynamic testing: Definitions, differences, and business considerations. https://testfort.com/blog/static-vs-dynamic-testing-definitions-differences-and-business-considerations

[28] Paspelava, D. (2023). 4 levels of software testing: How to develop a reliable product. Retrieved November 21, 2024, from https://www.exposit.com/blog/4-levels-software-testing-how-develop-reliable-product/

[29] Bergmann, S(2024). Raytracer [Software]. GitHub. https://github.com/sebastianbergmann/raytracer

[30] Myers, G. J., Sandler, C., & Badgett, T. (2011). The Art of Software Testing (3rd ed.). John Wiley & Sons.

[31] Sommerville, I. (2016). Software Engineering (10th ed.). Pearson.

[32] Burnstein, I. (2003). Practical Software Testing: A Process-Oriented Approach. Springer Science & Business Media.

[33] Beizer, B. (1990). Software Testing Techniques (2nd ed.). Van Nostrand Reinhold.

[34] Terra, J. (2024, April 11). System Testing in Software Testing: Definition, Types, and Tips. Caltech Bootcamps. Retrieved November 21, 2024, from https://pg-p.ctme.caltech.edu/blog/coding/system-testing-in-software-testing-types-tips

[35] Sourojit, D. (2024, October 7). What is System Testing? (Examples, Use Cases, Types). BrowserStack. Retrieved November 21, 2024, from https://www.browserstack.com/guide/what-is-system-testing

[36] Stanford UIT. (n.d.). User Acceptance Testing | University IT. University IT. Retrieved November 21, 2024, from https://uit.stanford.edu/pmo/UAT

[37] Elazar, E. (2018, April 23). What is User Acceptance Testing (UAT)? Experts Explain Simply. Panaya. Retrieved November 21, 2024, from https://www.panaya.com/blog/testing/what-is-uat-testing/#h-uat-prerequisites

[38] Javatpoint. (n.d.). Cause-Effect Graph Technique in Black Box Testing - javatpoint. Javatpoint. Retrieved November 21, 2024, from https://www.javatpoint.com/cause-and-effect-graph-technique-in-black-box-testing

[39] GeeksForGeeks. (n.d.). What is Usability Testing? — updated 2024 | IxDF. The Interaction Design Foundation. Retrieved November 22, 2024, from https://www.interaction-design.org/literature/topics/usability-testing#what_is_usability_testing?-0

[40] Interaction Design Foundation. (n.d.). What is Usability Testing? — updated 2024 | IxDF. The Interaction Design Foundation. Retrieved November 22, 2024, from https://www.interaction-design.org/literature/topics/usability-testing#what_is_usability_testing?-0

[41] Pressman, R. S. (2001). Software engineering : a practitioner's approach (5th ed.). McGraw-Hill. https://repository.unikom.ac.id/45085/1/Software%20Engineering%20-%20Roger%20S%20Pressman%20%5B5th%20edition%5D.pdf

[42] TMAP Sogeti. (n.d.). Performance Testing. tmap.net. https://www.tmap.net/building-blocks/performance-testing

[43] Xu, A. (2023, October 28). EP83: Explaining 9 Types of API Testing. ByteByteGo Newsletter. Retrieved November 24, 2024, from https://blog.bytebytego.com/p/ep83-explaining-9-types-of-api-testing

[44] ISO25000. (n.d.). Reliability - ISO 25000 STANDARDS. ISO/IEC 25000. Retrieved November 24, 2024, from https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/62-reliability

[45] BrowserStack. (n.d.). What is Test Execution: Importance, Process. What is Test Execution: Importance, Process. https://www.browserstack.com/test-management/features/test-run-management/what-is-test-execution

[46] GeeksforGeeks. (n.d.). Test Execution For Software Testing. GeeksforGeeks. https://www.geeksforgeeks.org/test-execution-for-software-testing/

[47] Najafi, A., Shang, W., & Rigby, P. C. (2019, May). Improving test effectiveness using test executions history: An industrial experience report. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) (pp. 213-222). IEEE.

[48] SmartBear. (n.d.). Test Cycles (Overview) | Zephyr Scale Cloud Documentation. SmartBear Support. https://support.smartbear.com/zephyr-scale-cloud/docs/en/test-cycles/test-cycles--overview-.html

[49] PHPUnit. (n.d.). PHPUnit Manual. PHPUnit Manual — PHPUnit 11.4 Manual. https://docs.phpunit.de/en/11.4/

[50] GeeksforGeeks. (2024, March 22). What is Test Data in Software Testing? GeeksforGeeks. https://www.geeksforgeeks.org/what-is-test-data-in-software-testing/

[51] Kualitee. (2023, October 19,). Test Case Management: How to Track and Report on Test Case Results. Kualitee. https://www.kualitee.com/blog/test-case-management/how-to-track-and-report-on-test-case-results/

[52] LambdaTest. (2021, January 27). How To Generate PHPUnit Coverage Report In HTML and XML? LambdaTest. https://www.lambdatest.com/blog/phpunit-code-coverage-report-html/

[53] TypeError. (n.d.). Code Coverage Analysis. TypeError. https://www.typeerror.org/docs/phpunit~8/code-coverage-analysis

[54] GeeksforGeeks. (n.d.). Defect Management Process GeeksforGeeks. https://www.geeksforgeeks.org/defect-management-process/

[55] Jira Bug Life Cycle. (n.d.). Javatpoint. https://www.javatpoint.com/jira-bug-life-cycle

[56] Mutalik, D. (2021, July 7). Defect Life Cycle. What is the Defect/Bug Life Cycle in Software Testing? https://www.toolsqa.com/software-testing/defect-life-cycle/

[57] Brooks, H., & Chatterjee, A. (2023, July 19). Mastering Defect Management: Understanding Severity vs. Priority in the Life Cycle. testRigor. https://testrigor.com/blog/mastering-defect-management-understanding-severity-vs-priority-in-the-life-cycle/

[58] Defect Report — Learn with examples. (n.d.). Tuskr. https://tuskr.app/learn/defect-report

[59] Kramer, N. (2024, September 11). Defect Tracking Best Practices for Software QA. Defect Tracking Best Practices for Software QA. https://daily.dev/blog/defect-tracking-best-practices-for-software-qa

[60] Kristina Berga (2024, July 3). What is Automated Testing and how Does it work? from https://www.testdevlab.com/blog/what-is-automated-testing-and-how-does-it-work-with-example

[61] Maria Homann (2024, July 1). An Introduction to data-driven testing. from https://www.leapwork.com/blog/a-short-introduction-to-data-driven-testing

[62] Sebastian Bergmann (2024) Writing Tests for PHPUnit from docs.phpunit.de/en/10.5/writing-tests-for-phpunit.html?

[63] Krishna, V (2024, October 22). Implementing a Data-Driven Testing Approach: A Comprehensive Guide. From https://www.linkedin.com/pulse/implementing-data-driven-testing-approach-guide-vijaya-krishna-0vbmc/

[64] Amin, A. (2024, September 22) Understanding Mock objects in PHPUnit Testing. DEV Community. From https://dev.to/ialaminpro/understanding-mock-objects-in-phpunit-testing-c55

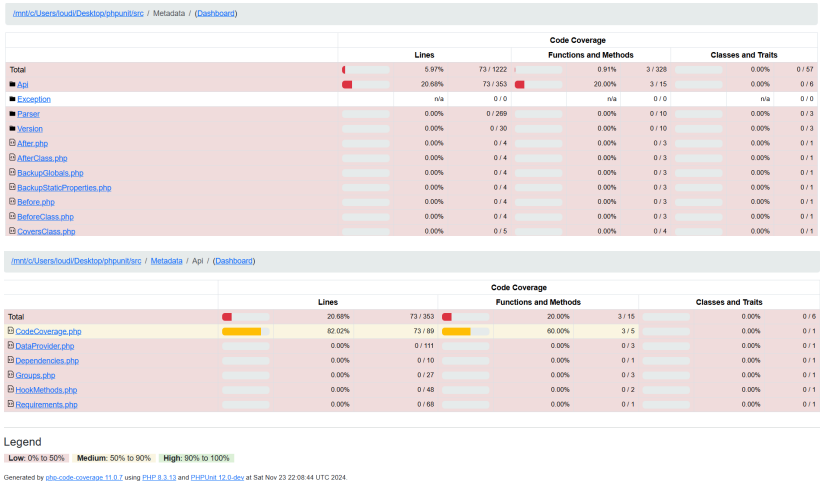
[65] Umesh, S (2023, February 11) PHP Testing and Debugging Techniques: A comprehensive guide to ensuring code quality and debugging efficiently. Medium. From https://umeshsl.medium.com/php-testing-and-debugging-techniques-a-comprehensive-guide-to-ensuring-code-quality-and-debugging-e72549a30e77

[66] Khanam, I. A. (2024, September 13). What is Regression Testing & Why Should it Be Automated? Opkey. Retrieved November 24, 2024, from https://www.opkey.com/blog/what-is-regression-testing-why-should-it-be-automated

Screen Shots

Part 3.1 Test coverage and Testing Dimensions

[Screenshot: HTML Coverage reports]



/mnt/c/Users/loudi/Desktop/phpunit/src / Metadata / (Dashboard)

Code Coverage

	Lines		Functions and Methods		Classes and Traits	
Total	5.97%	73 / 1222	0.91%	3 / 328	0.00%	0 / 57
Api	20.68%	73 / 353	20.00%	3 / 15	0.00%	0 / 6
Exception	n/a	0 / 0	n/a	0 / 0	n/a	0 / 0
Parser	0.00%	0 / 269	0.00%	0 / 10	0.00%	0 / 3
Version	0.00%	0 / 30	0.00%	0 / 10	0.00%	0 / 3
After.php	0.00%	0 / 4	0.00%	0 / 3	0.00%	0 / 1
AfterClass.php	0.00%	0 / 4	0.00%	0 / 3	0.00%	0 / 1
BackupGlobals.php	0.00%	0 / 4	0.00%	0 / 3	0.00%	0 / 1
BackupStaticProperties.php	0.00%	0 / 4	0.00%	0 / 3	0.00%	0 / 1
Before.php	0.00%	0 / 4	0.00%	0 / 3	0.00%	0 / 1
BeforeClass.php	0.00%	0 / 4	0.00%	0 / 3	0.00%	0 / 1
CoversClass.php	0.00%	0 / 5	0.00%	0 / 4	0.00%	0 / 1

/mnt/c/Users/loudi/Desktop/phpunit/src / Metadata / Api / (Dashboard)

Code Coverage

	Lines		Functions and Methods		Classes and Traits	
Total	20.68%	73 / 353	20.00%	3 / 15	0.00%	0 / 6
CodeCoverage.php	82.02%	73 / 89	60.00%	3 / 5	0.00%	0 / 1
DataProvider.php	0.00%	0 / 111	0.00%	0 / 3	0.00%	0 / 1
Dependencies.php	0.00%	0 / 10	0.00%	0 / 1	0.00%	0 / 1
Groups.php	0.00%	0 / 27	0.00%	0 / 3	0.00%	0 / 1
HookMethods.php	0.00%	0 / 48	0.00%	0 / 2	0.00%	0 / 1
Requirements.php	0.00%	0 / 68	0.00%	0 / 1	0.00%	0 / 1

Legend

Low: 0% to 50% Medium: 50% to 90% High: 90% to 100%

Generated by php-code-coverage 11.0.7 using PHP 8.3.13 and PHPUnit 12.0-dev at Sat Nov 23 22:08:44 UTC 2024.

Test Case Examples

Part 5.1 Cross Feature Testing

[Test case: secondary code used in 5.1]

```
Unset
class WikipediaCrossFeatureTest extends TestCase
{
  private $driver;
  private $wait;
  private $logger;

  const BASE_URL = 'https://www.wikipedia.org';
            const       SEARCH_BOX_SELECTOR       =
'input#searchInput';
            const      SEARCH_BUTTON_SELECTOR      =
'button.pure-button.pure-button-primary-progressi
ve';
  const SEARCH_TERM = 'Software engineering';
            const      EXPECTED_URL_CONTAINS      =
'Software_engineering';

  protected function setUp(): void
  {
    $options = new ChromeOptions();
       $options->addArguments(['--headless']);  //
Run tests in headless mode
                              $capabilities       =
DesiredCapabilities::chrome();

$capabilities->setCapability(ChromeOptions::CAPAB
ILITY, $options);

                              $this->driver       =
RemoteWebDriver::create('http://localhost:65176',
$capabilities); // Update with your WebDriver URL
                        $this->wait      =      new
WebDriverWait($this->driver, 10);

    // Set up logger
    $logFilePath = __DIR__ . '/test.log';
                       $this->logger      =      new
Logger('wikipedia_cross_feature_test');
                    $this->logger->pushHandler(new
StreamHandler($logFilePath, Logger::DEBUG));
```

```
    // Ensure the log file is created
    if (!file_exists($logFilePath)) {
      touch($logFilePath);
    }
  }

  public function testCrossFeatureIntegration()
  {
    $this->navigateToHomePage();
    $this->searchForTerm(self::SEARCH_TERM);

$this->verifySearchResults(self::EXPECTED_URL_CON
TAINS);
  }

  private function navigateToHomePage()
  {
    $startTime = microtime(true);
    $this->driver->get(self::BASE_URL);
    $endTime = microtime(true);
      $this->logger->info('Navigated to Wikipedia
homepage', [
      'url' => self::BASE_URL,
      'response_time' => $endTime - $startTime
    ]);
  }

  private function searchForTerm($term)
  {

$this->waitForElement(self::SEARCH_BOX_SELECTOR);
                                $searchBox       =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BOX_SELECTOR));
      $this->assertTrue($searchBox->isDisplayed(),
"Search bar is not visible.");
    $searchBox->sendKeys($term);
       $this->logger->info('Entered search term',
['term' => $term]);

    $startTime = microtime(true);
                               $searchButton       =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BUTTON_SELECTOR));
    $searchButton->click();
    $endTime = microtime(true);
     $this->logger->info('Clicked search button',
[
      'response_time' => $endTime - $startTime
    ]);
  }

                         private           function
verifySearchResults($expectedUrlContains)
  {
    $startTime = microtime(true);
    try {

$this->waitForUrlContains($expectedUrlContains);
                                     }        catch
(\Facebook\WebDriver\Exception\TimeoutException
$e) {
```

```
                                $currentURL     =
$this->driver->getCurrentURL();
        $this->logger->error('Timeout waiting for
URL to contain expected string', [
        'current_url' => $currentURL,
                        'expected_url_contains'   =>
$expectedUrlContains
      ]);
      throw $e;
    }
    $endTime = microtime(true);
    $currentURL = $this->driver->getCurrentURL();
     $this->logger->info('Search results loaded',
[
      'url' => $currentURL,
      'response_time' => $endTime - $startTime
    ]);

$this->assertStringContainsString($expectedUrlCon
tains, $currentURL, "Search result page URL is
not as expected.");
  }

  private function waitForElement($selector)
  {

$this->wait->until(WebDriverExpectedCondition::pr
esenceOfElementLocated(WebDriverBy::cssSelector($
selector)));
  }
```

```
  private function waitForUrlContains($text)
  {

$this->wait->until(WebDriverExpectedCondition::ur
lContains($text));
  }

  protected function tearDown(): void
  {
    $this->driver->quit();
  }
}
```

Part 5.2 Cause-effect graphing
[Test case: secondary code used in 5.2]

Unset

```
<?php
class WikipediaCauseEffectTest extends TestCase
{
  private $driver;
  private $wait;

  const BASE_URL = 'https://www.wikipedia.org';
            const      SEARCH_BOX_SELECTOR      =
'input#searchInput';
          const     SEARCH_BUTTON_SELECTOR     =
'button.pure-button.pure-button-primary-progressi
ve';
```

```
            const      NO_RESULTS_SELECTOR      =
'.mw-search-results-info .mw-search-nonefound';
  const PAGE_TITLE_SELECTOR = 'h1#firstHeading';

  protected function setUp(): void
  {
    $options = new ChromeOptions();
    $options->addArguments(['--headless']);
                            $capabilities       =
DesiredCapabilities::chrome();

$capabilities->setCapability(ChromeOptions::CAPAB
ILITY, $options);

                            $this->driver       =
RemoteWebDriver::create('http://localhost:57585',
$capabilities);
                        $this->wait      =      new
WebDriverWait($this->driver, 20);
  }

  /**
   * @dataProvider causeEffectDataProvider
   */
   public  function  testCauseEffectSearch($input,
$expectedOutcome)
  {
    $this->driver->get(self::BASE_URL);

$this->waitForElement(self::SEARCH_BOX_SELECTOR);
```

```
                                  $searchBox       =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BOX_SELECTOR));
    $searchBox->sendKeys($input);
                               $searchButton       =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BUTTON_SELECTOR));
    $searchButton->click();

    if ($expectedOutcome === 'success') {
      try {

$this->waitForElement(self::PAGE_TITLE_SELECTOR);
                                     $pageTitle    =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::PAGE_TITLE_SELECTOR));
         $this->assertStringContainsString($input,
$pageTitle->getText(),   "Page   title   does   not
contain the search term.");
                                        }      catch
(\Facebook\WebDriver\Exception\TimeoutException
$e) {
                                    $currentURL    =
$this->driver->getCurrentURL();
           echo  "Current  URL:  "  .  $currentURL  .
"\n";
                                    $pageSource    =
$this->driver->getPageSource();
           echo  "Page  Source:  "  .  $pageSource  .
"\n";
        throw $e;
```

```
      }
    } else {
      try {

$this->waitForElement(self::NO_RESULTS_SELECTOR);
                            $noResultsMessage    =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::NO_RESULTS_SELECTOR));

$this->assertTrue($noResultsMessage->isDisplayed(
), "No results message is not displayed.");
                                      }      catch
(\Facebook\WebDriver\Exception\TimeoutException
$e) {
                                  $currentURL    =
$this->driver->getCurrentURL();
           echo "Current URL: " . $currentURL .
"\n";
                                  $pageSource    =
$this->driver->getPageSource();
           echo "Page Source: " . $pageSource .
"\n";
        throw $e;
      }
    }
  }

               public       static       function
causeEffectDataProvider()
  {
    return [
      ['Software engineering', 'success'],
      ['Nonexistentsearch', 'error']
    ];
  }

  private function waitForElement($selector)
  {

$this->wait->until(WebDriverExpectedCondition::pr
esenceOfElementLocated(WebDriverBy::cssSelector($
selector)));
  }

  protected function tearDown(): void
  {
    $this->driver->quit();
  }
}
```

Part 5.3 Scenario Testing
[Test case: secondary code used in 5.3]

```
Unset
<?php
class AmazonScenarioTest extends TestCase
{
  private $driver;
  private $wait;
```

```
  private $logger;

   const BASE_URL = 'https://www.amazon.com'; //
Amazon base URL
            const      SEARCH_BOX_SELECTOR      =
'input#twotabsearchtextbox';
           const     SEARCH_BUTTON_SELECTOR     =
'input#nav-search-submit-button';
             const       PRODUCT_SELECTOR       =
'div.a-section.aok-relative.s-image-square-aspect
img.s-image';
            const      ADD_TO_CART_SELECTOR      =
'div.a-button-stack
input[id="add-to-cart-button"]';
  const CART_BUTTON_SELECTOR = 'a#nav-cart';
          const     CHECKOUT_BUTTON_SELECTOR     =
'input[name="proceedToRetailCheckout"]';
           const     SIGN_IN_EMAIL_SELECTOR     =
'input#ap_email';
         const     SIGN_IN_CONTINUE_SELECTOR     =
'input#continue';
         const     SIGN_IN_PASSWORD_SELECTOR     =
'input#ap_password';
          const     SIGN_IN_SUBMIT_SELECTOR     =
'input#signInSubmit';

   // Credentials (replace with valid credentials
for testing)
  const TEST_EMAIL = 'pamintabilog123@gmail.com';
  const TEST_PASSWORD = 'meowmeow';
```

```
  protected function setUp(): void
  {
    $options = new ChromeOptions();
       $options->addArguments(['--headless']);  //
Run tests in headless mode
                              $capabilities       =
DesiredCapabilities::chrome();

$capabilities->setCapability(ChromeOptions::CAPAB
ILITY, $options);

                              $this->driver       =
RemoteWebDriver::create('http://localhost:65176',
$capabilities); // WebDriver URL
                         $this->wait      =      new
WebDriverWait($this->driver, 10);

    // Set up logger
                  $logFilePath    =    __DIR__    .
'/logs/scenario_test.log';
                       $this->logger     =     new
Logger('amazon_scenario_test');
                     $this->logger->pushHandler(new
StreamHandler($logFilePath, Logger::DEBUG));
  }

  public function testAmazonScenario()
  {
     $this->logger->info('Starting Amazon scenario
test');
```

```
    $this->driver->get(self::BASE_URL);

$this->waitForElement(self::SEARCH_BOX_SELECTOR);
                                $searchBox       =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BOX_SELECTOR));
    $searchBox->sendKeys('Fan');
                              $searchButton      =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BUTTON_SELECTOR));
    $searchButton->click();
       $this->logger->info('Performed  search  for
Fan');

    $this->selectProduct();
    $this->addToCart();
    $this->proceedToCheckout();
    $this->signIn();
    $this->verifyCheckoutRedirection();
  }

  private function selectProduct()
  {

$this->waitForElement(self::PRODUCT_SELECTOR);
                                  $product        =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::PRODUCT_SELECTOR));
    $product->click();
    $this->logger->info('Selected product');
  }
```

```
  private function addToCart()
  {

$this->waitForElement(self::ADD_TO_CART_SELECTOR)
;
                            $addToCartButton      =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::ADD_TO_CART_SELECTOR));
    $addToCartButton->click();
    $this->logger->info('Added product to cart');
  }

  private function proceedToCheckout()
  {

$this->waitForElement(self::CART_BUTTON_SELECTOR)
;
                               $cartButton       =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::CART_BUTTON_SELECTOR));
    $cartButton->click();
    $this->logger->info('Navigated to cart');


$this->waitForElement(self::CHECKOUT_BUTTON_SELEC
TOR);
                             $checkoutButton      =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::CHECKOUT_BUTTON_SELECTOR));
    $checkoutButton->click();
```

```
    $this->logger->info('Proceeded to checkout');
  }

  private function signIn()
  {

$this->waitForElement(self::SIGN_IN_EMAIL_SELECTO
R);
                              $emailField       =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SIGN_IN_EMAIL_SELECTOR));
    $emailField->sendKeys(self::TEST_EMAIL);
    $this->logger->info('Entered email');

                           $continueButton      =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SIGN_IN_CONTINUE_SELECTOR));
    $continueButton->click();
           $this->logger->info('Clicked   continue
button');


$this->waitForElement(self::SIGN_IN_PASSWORD_SELE
CTOR);
                            $passwordField      =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SIGN_IN_PASSWORD_SELECTOR));

$passwordField->sendKeys(self::TEST_PASSWORD);
    $this->logger->info('Entered password');
```

```
                              $signInButton       =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SIGN_IN_SUBMIT_SELECTOR));
    $signInButton->click();
            $this->logger->info('Clicked   sign-in
button');
  }

  private function verifyCheckoutRedirection()
  {
    $this->waitForUrlContains('addressselect');
    $currentURL = $this->driver->getCurrentURL();
          $this->logger->info('Verified   checkout
redirection', ['url' => $currentURL]);

$this->assertStringContainsString('addressselect'
,  $currentURL,  "Failed  to  reach  the  checkout
page.");
  }

     private  function  waitForElement($selector,
$type = 'css')
  {
              $by    =    $type    ===    'css'    ?
WebDriverBy::cssSelector($selector)             :
WebDriverBy::id($selector);

$this->wait->until(WebDriverExpectedCondition::pr
esenceOfElementLocated($by));
  }
```

```
  private function waitForUrlContains($text)
  {

$this->wait->until(WebDriverExpectedCondition::ur
lContains($text));
  }

  protected function tearDown(): void
  {
    $this->driver->quit();
  }
}
```

Part 5.4 Usability Testing
[Test case: secondary code used in 5.4]

Unset

```
<?php
class AmazonUsabilityTest extends TestCase
{
  private $driver;
  private $wait;
  private $logger;

  const BASE_URL = 'https://www.amazon.com';
            const      SEARCH_BOX_SELECTOR      =
'input#twotabsearchtextbox';
           const     SEARCH_BUTTON_SELECTOR     =
'input#nav-search-submit-button';
  const SIGN_IN_LINK_ID = 'nav-link-accountList';
  const EMAIL_FIELD_ID = 'ap_email';
         const    CATEGORY_DROPDOWN_SELECTOR    =
'select.nav-search-dropdown';
        const    ELECTRONICS_OPTION_SELECTOR    =
'option[value="search-alias=electronics-intl-ship
"]';

  protected function setUp(): void
  {
    $options = new ChromeOptions();
       $options->addArguments(['--headless']);  //
Run tests in headless mode
                              $capabilities       =
DesiredCapabilities::chrome();

$capabilities->setCapability(ChromeOptions::CAPAB
ILITY, $options);

                              $this->driver       =
RemoteWebDriver::create('http://localhost:60537',
$capabilities); // Update with your WebDriver URL
                         $this->wait      =      new
WebDriverWait($this->driver, 10);

    // Set up logger
    $logFilePath = __DIR__ . '/test.log';
                      $this->logger     =     new
Logger('amazon_usability_test');
                    $this->logger->pushHandler(new
StreamHandler($logFilePath, Logger::DEBUG));
```

```
    // Ensure the log file is created
    if (!file_exists($logFilePath)) {
      touch($logFilePath);
    }
  }

  public function testUsabilityFeatures()
  {
    $this->driver->get(self::BASE_URL);

$this->waitForElement(self::SEARCH_BOX_SELECTOR);
                              $searchBox       =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BOX_SELECTOR));
    $searchBox->sendKeys('Laptop');
                            $searchButton      =
$this->driver->findElement(WebDriverBy::cssSelect
or(self::SEARCH_BUTTON_SELECTOR));
    $searchButton->click();

    $currentURL = $this->driver->getCurrentURL();
     $this->logger->info('Search results loaded',
[
      'url' => $currentURL,
      'response_time' => $endTime - $startTime
    ]);
       $this->assertStringContainsString('Laptop',
$currentURL, "Search functionality is not working
as expected.");
```

```
    $startTime = microtime(true);
    $this->driver->navigate()->back();
    $endTime = microtime(true);
         $this->logger->info('Navigated  back  to
homepage', [
      'response_time' => $endTime - $startTime
    ]);

      $this->waitForElement(self::SIGN_IN_LINK_ID,
'id');
                                $signInLink       =
$this->driver->findElement(WebDriverBy::id(self::
SIGN_IN_LINK_ID));
    $signInLink->click();

       $this->waitForElement(self::EMAIL_FIELD_ID,
'id');
                                $emailField       =
$this->driver->findElement(WebDriverBy::id(self::
EMAIL_FIELD_ID));
     $this->assertTrue($emailField->isDisplayed(),
"Sign-in page did not load correctly.");
  }

     private  function  waitForElement($selector,
$type = 'css')
  {
              $by   =   $type   ===   'css'   ?
WebDriverBy::cssSelector($selector)            :
WebDriverBy::id($selector);
```

```
$this->wait->until(WebDriverExpectedCondition::pr
esenceOfElementLocated($by));
  }

  private function waitForUrlContains($text)
  {

$this->wait->until(WebDriverExpectedCondition::ur
lContains($text));
  }

  protected function tearDown(): void
  {
    $this->driver->quit();
  }
}
```

Part 6.1 Smoke Testing
[Test case: secondary code used in 6.1]

```
Unset
class GitHubLoginSmokeTest extends TestCase
{
  private $driver;

  protected function setUp(): void
  {
       // Use absolute path without environment
variable
    $options = new ChromeOptions();
    $options->addArguments([
      '--no-sandbox',
      '--disable-dev-shm-usage'
    ]);

                              $capabilities       =
DesiredCapabilities::chrome();

$capabilities->setCapability(ChromeOptions::CAPAB
ILITY, $options);

    // Connect to running ChromeDriver instance
    $this->driver = RemoteWebDriver::create(
      'http://localhost:9515',
      $capabilities
    );
  }

  public function testGitHubLoginForm()
  {
    // Code is in Part 6.1 Smoke Testing
  }

  protected function tearDown(): void
  {
    $this->driver->quit();
  }
}
```

Part 6.2 Performance Testing
[Test case: secondary code used in 6.2]

```
Unset
class GitHubHomePagePerformanceTest extends
TestCase
{
  private $webDriver;
  private $metrics = [];
  private $startTime;
  private const MAX_RETRIES = 3;
  private const TIMEOUT = 60; // seconds


  protected function setUp(): void
  {
    $options = new ChromeOptions();
    $options->addArguments([
      '--headless',
      '--disable-gpu',
      '--no-sandbox',
      '--disable-dev-shm-usage',
      '--ignore-certificate-errors'
    ]);


    $capabilities =
DesiredCapabilities::chrome();

$capabilities->setCapability(ChromeOptions::CAPAB
ILITY, $options);
    // Initialize WebDriver with retry logic
    $this->initializeWebDriver($capabilities);
    $this->startTime = microtime(true);
  }


  private function
initializeWebDriver($capabilities)
  {
    $retry = 0;
    while ($retry < self::MAX_RETRIES) {
      try {
        $this->webDriver =
RemoteWebDriver::create(
          'http://localhost:9515',
          $capabilities,
          self::TIMEOUT * 1000, // connection
timeout in milliseconds
          self::TIMEOUT * 1000  // request
timeout in milliseconds
        );
        return;
      } catch (\Exception $e) {
        $retry++;
        if ($retry === self::MAX_RETRIES) {
          throw new RuntimeException(
            "Failed to initialize WebDriver after
{$retry} attempts: " . $e->getMessage()
          );
        }
        sleep(2); // Wait before retry
```

```
      }
    }
  }


  protected function tearDown(): void
  {
    $totalTime = microtime(true) -
$this->startTime;
    $this->metrics['total_time'] =
round($totalTime, 2);


    file_put_contents(
      __DIR__ .
'/output/homepage_performance_report.json',
      json_encode($this->metrics,
JSON_PRETTY_PRINT)
    );


    if ($this->webDriver) {
      $this->webDriver->quit();
    }
  }
}
```

Part 6.3 Stress Testing
[Test case: secondary code used in 6.3]

```
Unset
class GitHubStressTest extends TestCase
{
    private $driver;
    private const LOGIN_URL =
'https://github.com/login';
    private const MAX_RESPONSE_TIME = 5.0; //
seconds

    protected function setUp(): void
    {
        $options = new ChromeOptions();
        $options->addArguments(['--no-sandbox',
'--headless']);
        $capabilities =
DesiredCapabilities::chrome();

$capabilities->setCapability(ChromeOptions::CAPAB
ILITY, $options);
$this->driver=
RemoteWebDriver::create('http://localhost:9515',
$capabilities);
}
```

Part 6.4 Reliability Testing
[Test case: secondary code used in 6.4]

```
Unset
class GitHubReliabilityTest extends TestCase
{
  private $driver;
  private const LOGIN_URL =
'https://github.com/login';
  private const CHECK_INTERVAL = 60; // 1 minute

  protected function setUp(): void
  {
    $options = new ChromeOptions();
    $options->addArguments(['--no-sandbox',
'--headless']);
    $capabilities =
DesiredCapabilities::chrome();

$capabilities->setCapability(ChromeOptions::CAPAB
ILITY, $options);

    $this->driver =
RemoteWebDriver::create('http://localhost:9515',
$capabilities);
  }

  private function assertElementsPresent(): void
  {
    $elements = [
      'login_field' =>
WebDriverBy::id('login_field'),
      'password' => WebDriverBy::id('password'),
      'submit' =>
WebDriverBy::cssSelector('input[type="submit"]')
    ];

    foreach ($elements as $name => $locator) {
      $this->assertTrue(

$this->driver->findElement($locator)->isDisplayed
(),
        "Element '$name' not found"
      );
    }
  }

  private function testFormValidation(): void
  {
    $loginField =
$this->driver->findElement(WebDriverBy::id('login
_field'));
    $loginField->sendKeys('test@example.com');

    $passwordField =
$this->driver->findElement(WebDriverBy::id('passw
ord'));
    $passwordField->sendKeys('invalid');

    $submit =
$this->driver->findElement(WebDriverBy::cssSelect
or('input[type="submit"]'));
    $submit->click();
```

```
    // Verify error message appears
    $this->driver->wait(10)->until(

WebDriverExpectedCondition::presenceOfElementLoca
ted(
        WebDriverBy::cssSelector('.flash-error')
      )
    );
  }

  private function measureResponseTime(): float
  {
    $start = microtime(true);
    $this->driver->get(self::LOGIN_URL);
    return microtime(true) - $start;
  }

  private function generateReport(array $checks,
array $errors): void
  {
    $report = [
      'total_checks' => count($checks),
      'successful_checks' => count($checks) -
count($errors),
      'errors' => count($errors),
      'reliability_rate' => ((count($checks) -
count($errors)) / count($checks)) * 100,
      'average_response_time' =>
array_sum(array_column($checks, 'response_time'))
/ count($checks),
      'error_details' => $errors
    ];

    file_put_contents(
      'output/reliability_test_report.json',
      json_encode($report, JSON_PRETTY_PRINT)
    );
  }

  public static function
reliabilityTestDataProvider(): array
  {
    return [
      '1 hour test' => [3600]
    ];
  }

  protected function tearDown(): void
  {
    if ($this->driver) {
      $this->driver->quit();
    }
  }
}
```

Test Results

Part 6.3: Stress Testing

[Test result: secondary result in 6.3]

output > stress_test_metrics.log


```
{"iteration":69,"timestamp":"2024-11-21 16:05:14","response_time":0.845,"memory_usage":8388608}
{"iteration":70,"timestamp":"2024-11-21 16:05:17","response_time":1.09,"memory_usage":8388608}
{"iteration":71,"timestamp":"2024-11-21 16:05:18","response_time":0.759,"memory_usage":8388608}
{"iteration":72,"timestamp":"2024-11-21 16:05:20","response_time":0.554,"memory_usage":8388608}
{"iteration":73,"timestamp":"2024-11-21 16:05:22","response_time":0.554,"memory_usage":8388608}
{"iteration":74,"timestamp":"2024-11-21 16:05:23","response_time":0.537,"memory_usage":8388608}
{"iteration":75,"timestamp":"2024-11-21 16:05:28","response_time":4.166,"memory_usage":8388608}
{"iteration":76,"timestamp":"2024-11-21 16:05:30","response_time":0.835,"memory_usage":8388608}
{"iteration":77,"timestamp":"2024-11-21 16:05:33","response_time":1.281,"memory_usage":8388608}
{"iteration":78,"timestamp":"2024-11-21 16:05:36","response_time":2.496,"memory_usage":8388608}
{"iteration":79,"timestamp":"2024-11-21 16:05:38","response_time":1.326,"memory_usage":8388608}
{"iteration":80,"timestamp":"2024-11-21 16:05:40","response_time":0.803,"memory_usage":8388608}
{"iteration":81,"timestamp":"2024-11-21 16:05:42","response_time":0.572,"memory_usage":8388608}
{"iteration":82,"timestamp":"2024-11-21 16:05:43","response_time":0.541,"memory_usage":8388608}
{"iteration":83,"timestamp":"2024-11-21 16:05:45","response_time":0.553,"memory_usage":8388608}
{"iteration":84,"timestamp":"2024-11-21 16:05:47","response_time":1.092,"memory_usage":8388608}
{"iteration":85,"timestamp":"2024-11-21 16:05:50","response_time":1.551,"memory_usage":8388608}
{"iteration":86,"timestamp":"2024-11-21 16:05:52","response_time":1.02,"memory_usage":8388608}
{"iteration":87,"timestamp":"2024-11-21 16:05:54","response_time":0.882,"memory_usage":8388608}
{"iteration":88,"timestamp":"2024-11-21 16:05:56","response_time":1.681,"memory_usage":8388608}
{"iteration":89,"timestamp":"2024-11-21 16:05:58","response_time":0.948,"memory_usage":8388608}
{"iteration":90,"timestamp":"2024-11-21 16:06:00","response_time":0.983,"memory_usage":8388608}
{"iteration":91,"timestamp":"2024-11-21 16:06:03","response_time":1.22,"memory_usage":8388608}
{"iteration":92,"timestamp":"2024-11-21 16:06:05","response_time":0.767,"memory_usage":8388608}
{"iteration":93,"timestamp":"2024-11-21 16:06:06","response_time":0.65,"memory_usage":8388608}
{"iteration":94,"timestamp":"2024-11-21 16:06:08","response_time":0.538,"memory_usage":8388608}
{"iteration":95,"timestamp":"2024-11-21 16:06:09","response_time":0.518,"memory_usage":8388608}
{"iteration":96,"timestamp":"2024-11-21 16:06:12","response_time":1.442,"memory_usage":8388608}
{"iteration":97,"timestamp":"2024-11-21 16:06:14","response_time":0.781,"memory_usage":8388608}
{"iteration":98,"timestamp":"2024-11-21 16:06:15","response_time":0.516,"memory_usage":8388608}
{"iteration":99,"timestamp":"2024-11-21 16:06:17","response_time":0.778,"memory_usage":8388608}
```


Part 6.4 Reliability Testing
[Test result: secondary result in 6.4]
Reliability testing 1 hour:

```
Response time: 1.16s
Completed: 80%.
Response time: 0.99s
Completed: 82%.
Response time: 1.40s
Completed: 83%.
Response time: 1.02s
Completed: 85%.
Response time: 1.17s
Completed: 87%.
Response time: 1.22s
Completed: 89%.
Response time: 1.22s
Completed: 90%.
Response time: 1.40s
Completed: 92%.
Response time: 1.25s
Completed: 94%.
Response time: 0.99s
Completed: 95%.
Response time: 1.40s
Completed: 97%.
Response time: 1.13s
Completed: 99%
Test completed. Duration: 3600 seconds
D                                                                    1 / 1 (100%)

Time: 01:00:16.051, Memory: 8.00 MB

OK, but there were issues!
Tests: 1, Assertions: 178, Deprecations: 1, PHPUnit Deprecations: 1.
```

Part 6.4 Reliability Testing
[Test result: secondary result in 6.4]
Reliability testing 4 hours:

```
Completed: 98%.
Response time: 0.99s
Completed: 97%.
Response time: 1.20s
Completed: 98%.
Response time: 0.99s
Completed: 97%.
Response time: 0.99s
Completed: 97%.
Response time: 0.99s
Response time: 0.99s
Response time: 0.99s
Response time: 0.99s
Completed: 97%.
Response time: 1.20s
Completed: 98%.
Response time: 1.00s
Completed: 98%.
Response time: 1.18s
Completed: 99%.
Response time: 1.20s
Completed: 99%.
Response time: 1.46s
Completed: 100%.
Response time: 1.00s
Completed: 100%
Test completed. Duration: 14400 seconds
```

```
Completed: 97%.
Response time: 1.58s
Completed: 98%.
Response time: 1.39s
Completed: 98%.
Response time: 1.62s
Completed: 98%.
Response time: 1.01s
Completed: 98%.
Response time: 1.58s
Completed: 98%.
Response time: 1.05s
Completed: 99%.
Response time: 1.58s
Completed: 99%.
Response time: 1.00s
Completed: 99%.
Response time: 1.68s
Completed: 99%.
Response time: 1.15s
Completed: 99%.
Response time: 1.65s
Completed: 100%.
Response time: 1.40s
Completed: 100%
Test completed. Duration: 28800 seconds
D                                                                    3 / 3 (100%)

Time: 13:01:59.074, Memory: 8.00 MB

OK, but there were issues!
Tests: 3, Assertions: 2298, Deprecations: 1, PHPUnit Deprecations: 1.
```

Part 6.4 Reliability Testing
[Test result: secondary result in 6.4]
Reliability Testing for 8 hours:

List of Figures

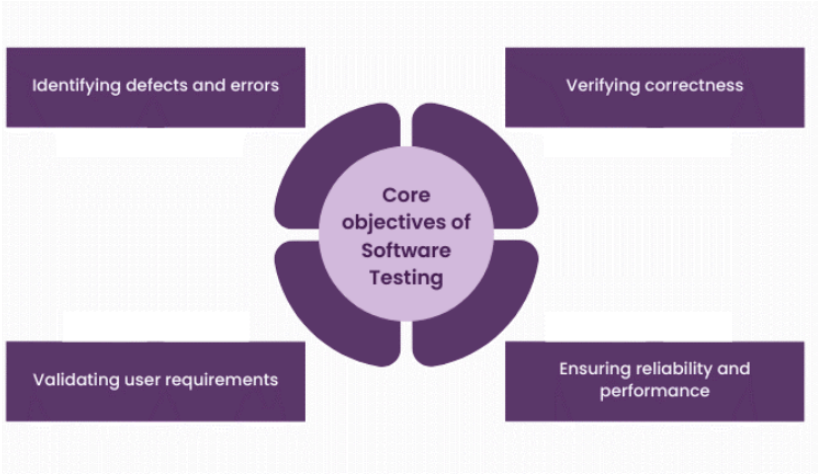


Figure 1: Core Objectives of software Testing

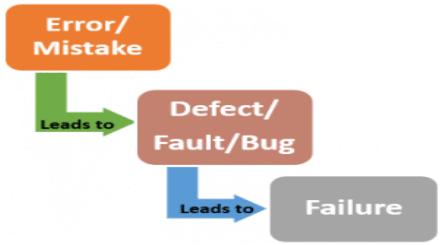


Figure 2: Defects: Faults and failure

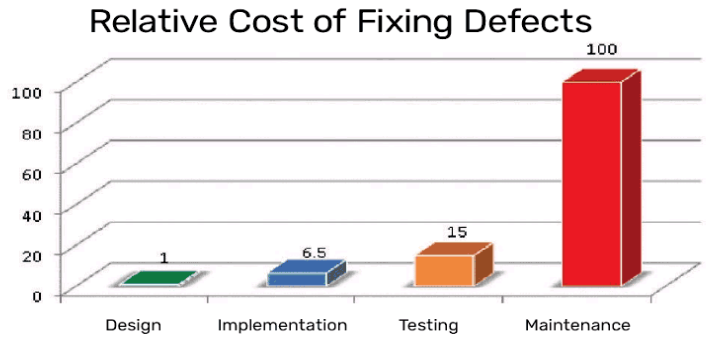


Figure 3: Relative Cost of Fixing Defects

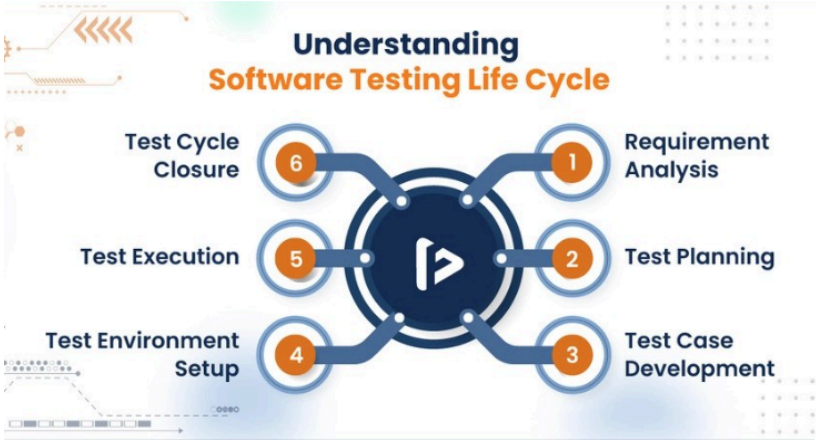


Figure 4: Understanding Software Testing Life Cycle.

Part 2.2 Defect injections and defect causes

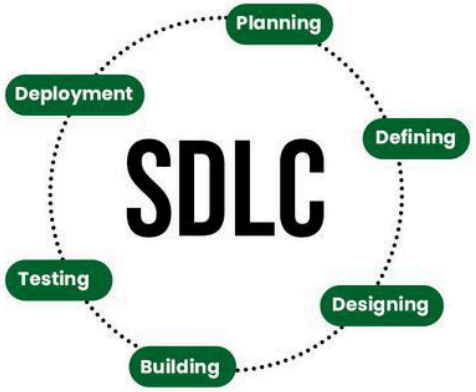


Figure 5: Software Development Life Cycle,

Part 2.4 Testing pipeline

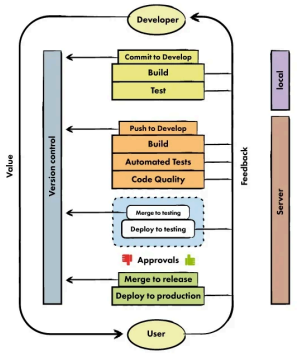


Figure 6: Simplified deployment pipeline can look like this

Part 3.2 White-box versus Black-box testing versus Gray-box testing

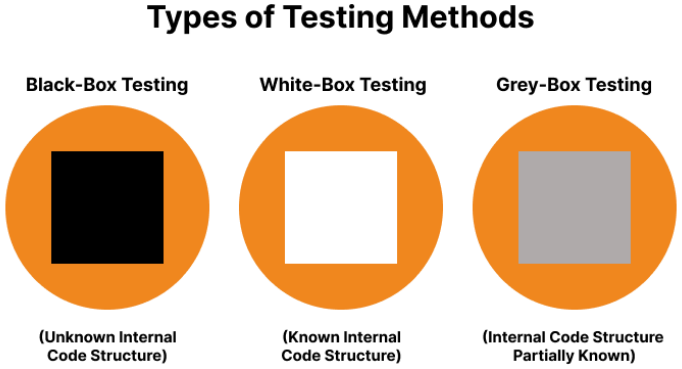


Figure 7: Types of Testing Methods

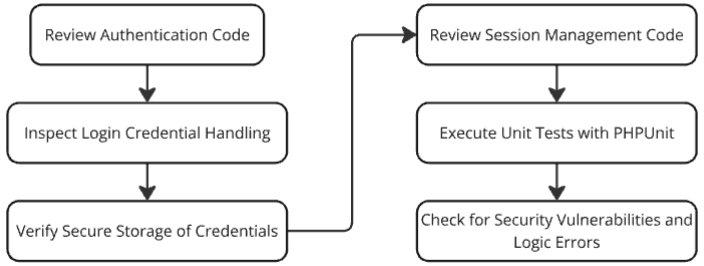


Figure 8: Testing the Authentication Process

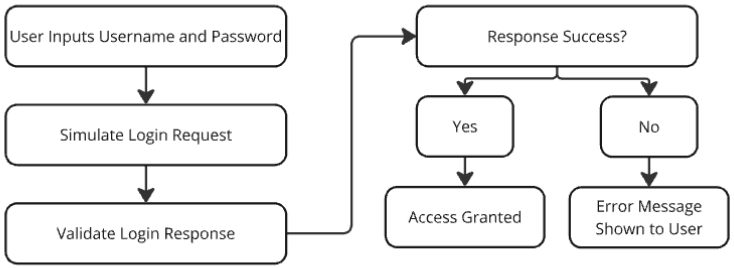


Figure 9: Testing a Login Functionality

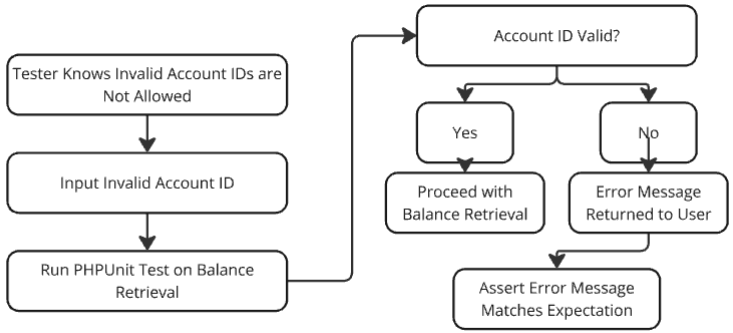


Figure 10: Testing for Invalid Account ID Handling

PART 11: Unit & Integration Testing

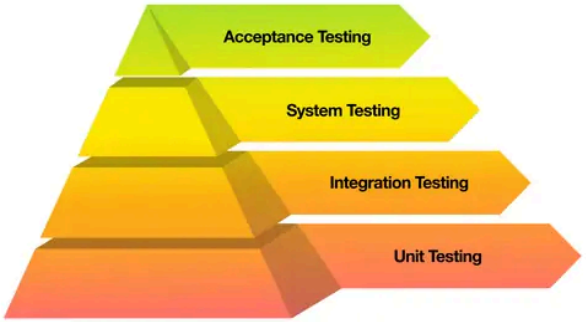


Figure 11: Levels of testing

Part 5.2 Cause-effect graphing

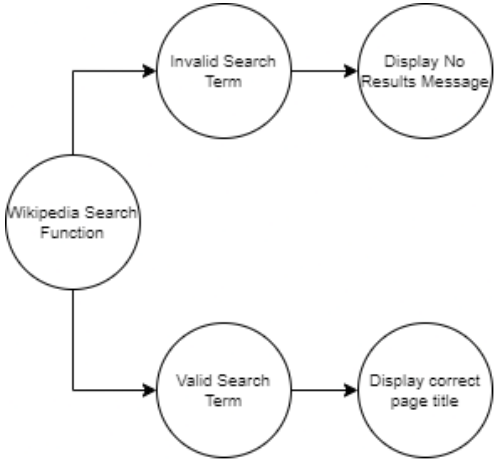


Figure 12: Cause-effect graph of the test case

Part 5.3 Scenario testing

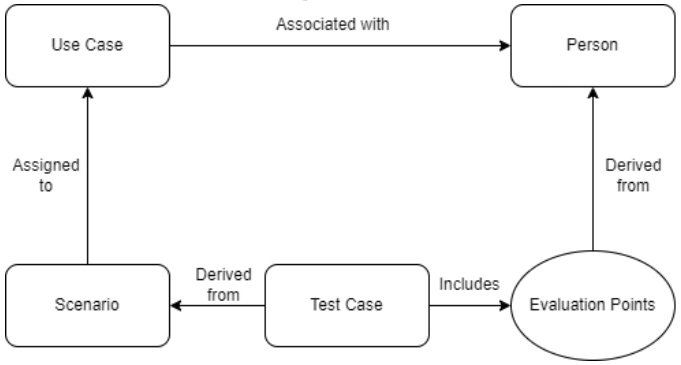


Figure 13: Scenario Testing Process

Part 6.1 Smoke Testing

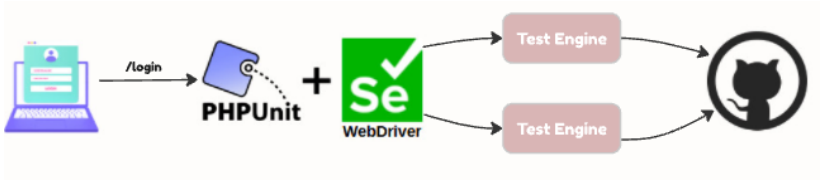


Figure 14: Endpoint verification

Part 6.2 Performance Testing

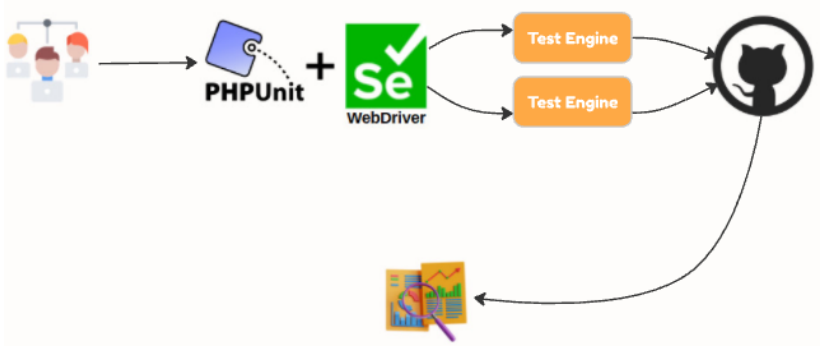


Figure 15: Performance metrics process

Part 6.3 Stress Testing

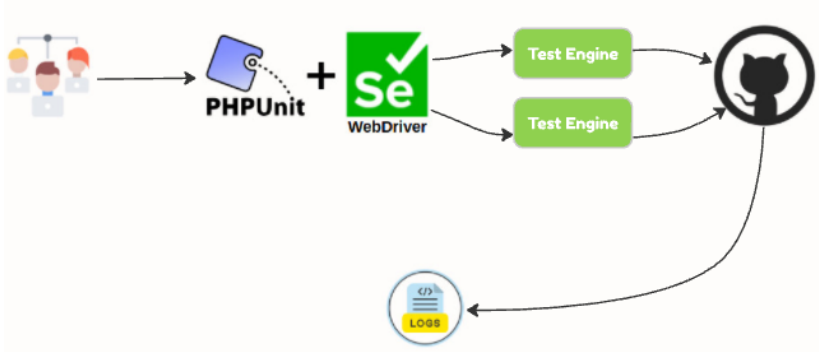


Figure 16: Evaluating log in load time process

Part 6.4 Reliability Testing

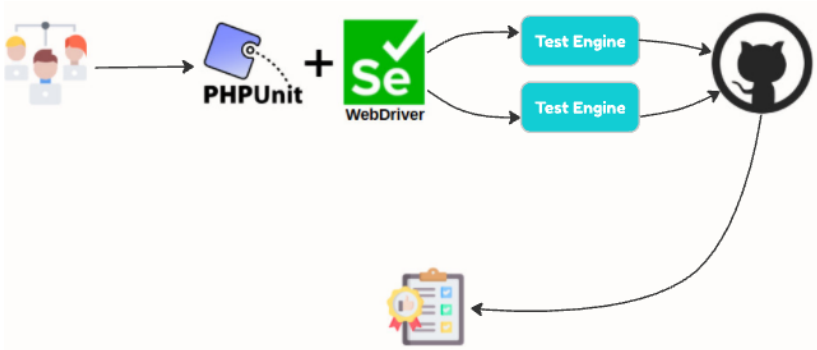


Figure 17: Reliability testing process

Part 6.5 Acceptance Testing

Figure 18: Online shopping purchasing process

Part 8.1 Defect life cycles

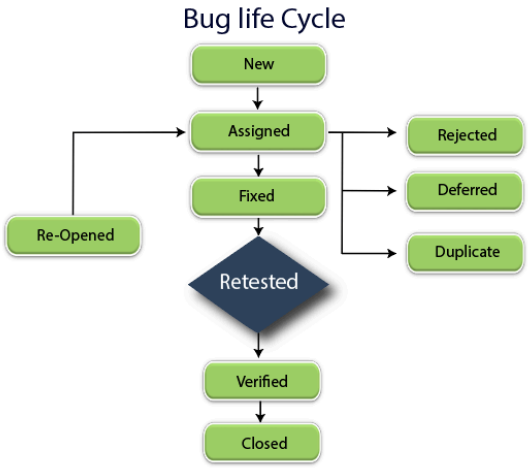


Figure 19: Bug Life Cycle

Part 9.1 Test automation approaches

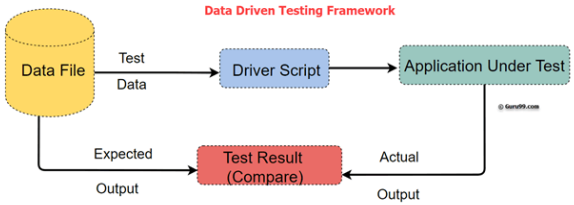


Figure 20: Data Driven Testing Framework

List of Tables

4.3 Equivalence partitioning

Input Range	**Representative Value**	**Expected Output**
Valid (Square matrices of same dimensions)	Matrix A = [[1.0, 2.0], [3.0, 4.0]] Matrix B = [[2.0, 1.0], [1.0, 2.0]]	Valid (Returns [[4.0, 5.0], [10.0, 11.0]])
Invalid (Non-square matrices)	Matrix A = [[1.0, 2.0], [3.0, 4.0]] Matrix B = [[2.0, 1.0],	Invalid (Throws InvalidArgumentException)

	[1.0, 2.0], [3.0, 4.0]]	
Invalid (Different dimensions)	Matrix A = [[1.0, 2.0]] Matrix B = [[1.0], [2.0]]	Invalid (Throws InvalidArgumentException)

Table 1: Test Case Partitions Table

4.5 Decision tables

Parameter	Value
t	4.0
Ray Origin	Tuple::point(0, 0, -5)
Ray Direction	Tuple::vector(0, 0, 1)
Expected Point	Tuple::point(0, 0, -1)
Expected Eye	Tuple::vector(0, 0, -1)
Expected Normal	Tuple::vector(0, 0, -1)
Expected Inside	false
Condition	Ray starts outside the sphere, pointing inward

Table 2: Test Case 1

Parameter	Value
t	1.0
Ray Origin	Tuple::point(0, 0, 0)
Ray Direction	Tuple::vector(0, 0, 1)
Expected Point	Tuple::point(0, 0, 1)
Expected Eye	Tuple::vector(0, 0, -1)
Expected Normal	Tuple::vector(0, 0, -1)
Expected Inside	true
Condition	Ray starts inside the sphere, pointing outward.

Table 3: Test Case 2