NORTH ISLAND COLLEGE

Programming practice article

25.01.2024

—

Yogesh Manni
DGL-104-DLU1L
n0205600
Programming Article

A Developer's Guide to Effective Code Testing

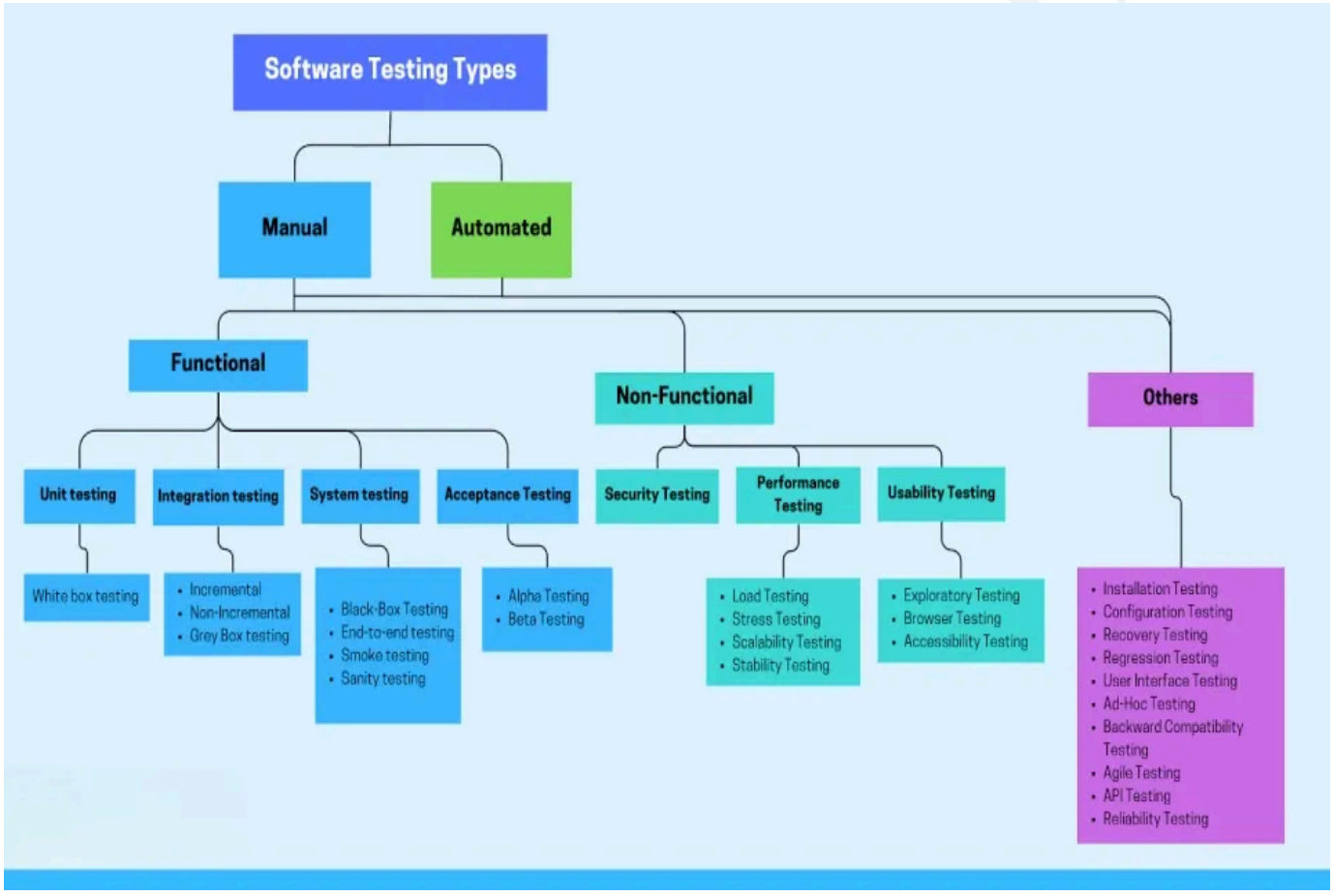
Introduction:

In the ever-evolving realm of software development, the art of code testing stands out as a critical practice. For budding developers, mastering different testing approaches can significantly enhance programming skills and contribute to project success. Different testing methods help us look at different parts of the code, ensuring everything fits together smoothly. In this article, we'll dive into the world of code testing, exploring its various approaches and strategies, significance, and how to seamlessly integrate them into your coding journey.

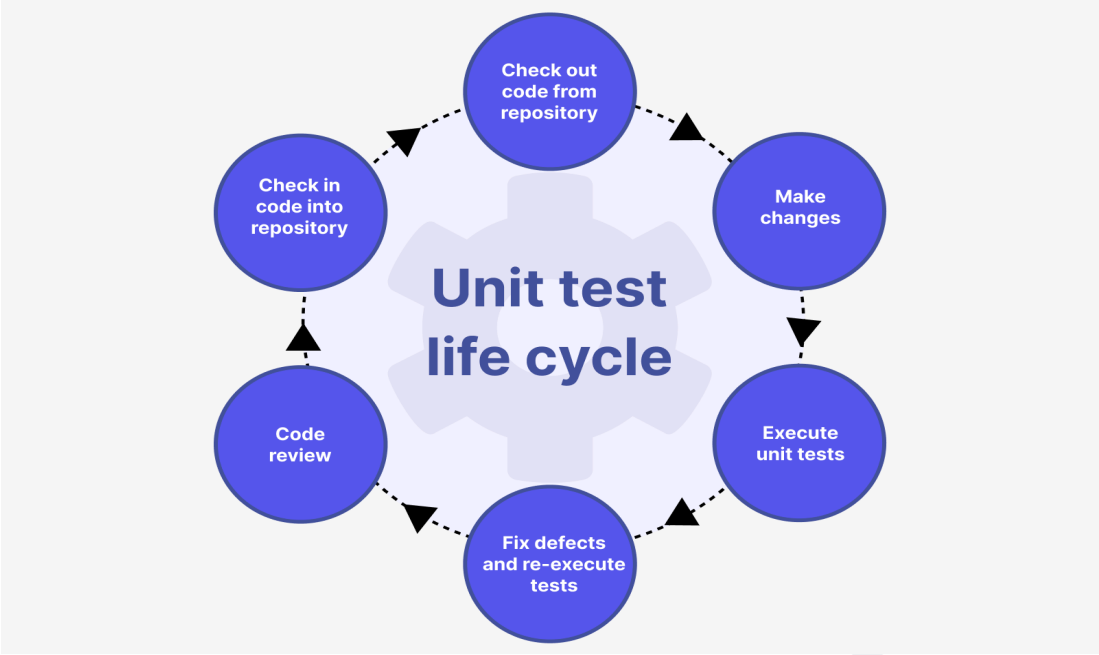
Understanding the Essence of Code Testing:

Code testing is not just a checkbox on the development checklist, it's a vital step that ensures the reliability and resilience of an application. It acts as a safety net, catching potential issues before they impact end-users. Embracing a holistic testing approach lays the groundwork for constructing dependable and sustainable software solutions.

Exploring Different Code Testing Approaches:



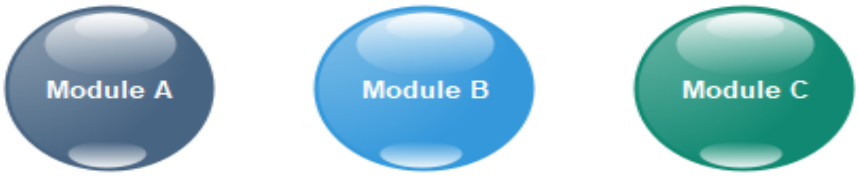
1. Unit Testing:



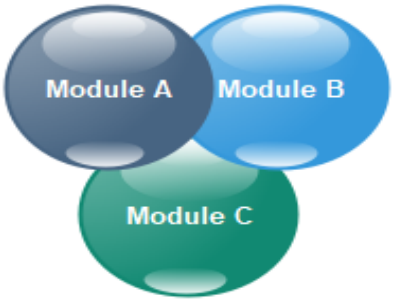
At its core, unit testing involves evaluating individual components or units of code in isolation. This practice is a proactive measure, catching bugs early in the development phase. This type of testing is performed by developers before the setup is handed over to the testing team to formally execute the test cases. Unit testing is performed by the respective developers on the individual units of source code assigned areas. The developers use test data that is different from the test data of the quality assurance team.Considera simple Python example, here we are testing the addition of two numbers and evaluating if the result will be correct :

```
def add_numbers(a, b):
    return a + b

 # Unit Test
 def test_add_numbers():
    assert add_numbers(2, 3) == 5
    assert add_numbers(-1, 1) == 0
```

2. Integration Testing:



Tested in Unit Testing



Under Integration Testing

Integration testing steps beyond individual units, examining how different modules interact. It ensures that integrated components function seamlessly together. In a web application, this could involve checking the smooth communication between the frontend and backend.

The integration test is like making sure all the parts of a machine not only work on their own but also work well together. Sometimes, when there are problems with two or more parts, these tests help find and fix those issues. They are a usual type of testing and are mostly done automatically. Integration tests make sure that information and commands flow smoothly between different parts. This is important because, in the end, all the pieces must work together as one complete software.

```
  # Integration Test - includes testing of various modules

// test authentication
   def test_user_authentication():
       # Simulate user login process
       assert login(username="user1", password="pass123") == True

// test user data
 def test_user_data():
        # Simulate user data checking process
        assert checkuserdata(data) == someObj;
```

3. End-to-End Testing:

This form of testing scrutinizes the complete workflow of an application, mirroring the user's experience. It identifies issues arising from the integration of different modules. It involves thoroughly testing a software component from its initiation to its completion, simulating the actual usage by end users. In the context of a web application, this entails launching a browser, accessing the appropriate URL, engaging with the application as intended, and validating its behavior. Similarly, for a desktop application, the process involves initiating the application, utilizing its features, and confirming its behavior. When conducting E2E testing for an API, the approach includes making calls in a manner reflective of how actual clients would interact with it.

A simple example in a JavaScript testing framework:

```
// End-to-End Test, testing the api to create a new user
   describe('User Registration', () => {
       it('should register a new user', () => {
           // Simulate user registration process
           expect(registerUser('JohnDoe',
'pass123')).to.equal('Registration successful');
```

```
      });
```

3. Component Testing:

When it comes to testing software, Component Testing is like giving each part a close look. It's when developers check how well individual pieces work on their own. They want to make sure that each part does what it's supposed to do and is easy for users to handle. Imagine a part being something that takes in information and gives out specific results, like a web page or a screen in the software.

One way is by checking how the software looks and how easy it is for users. Another thing to test is how quickly a page loads – that's also a part of Component Testing. Using a tool called SQL, testers also look at how safe the user interface part is. Testers play around with the page, putting in both right and wrong information, to see how well it responds. So, Component Testing is like taking each piece of software and making sure it not only works well on its own but also fits perfectly into the bigger picture.

Below is a React component which we can test individually. Components like these should be tested to ensure proper working of the whole app. The following component is the implementation of a counter which can be tested manually or with automatic tools.

```
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount(count + 1);
  };
  const decrement = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <p data-testid="count-display">Count: {count}</p>
      <button data-testid="increment-button" onClick={increment}>
        Increment
```

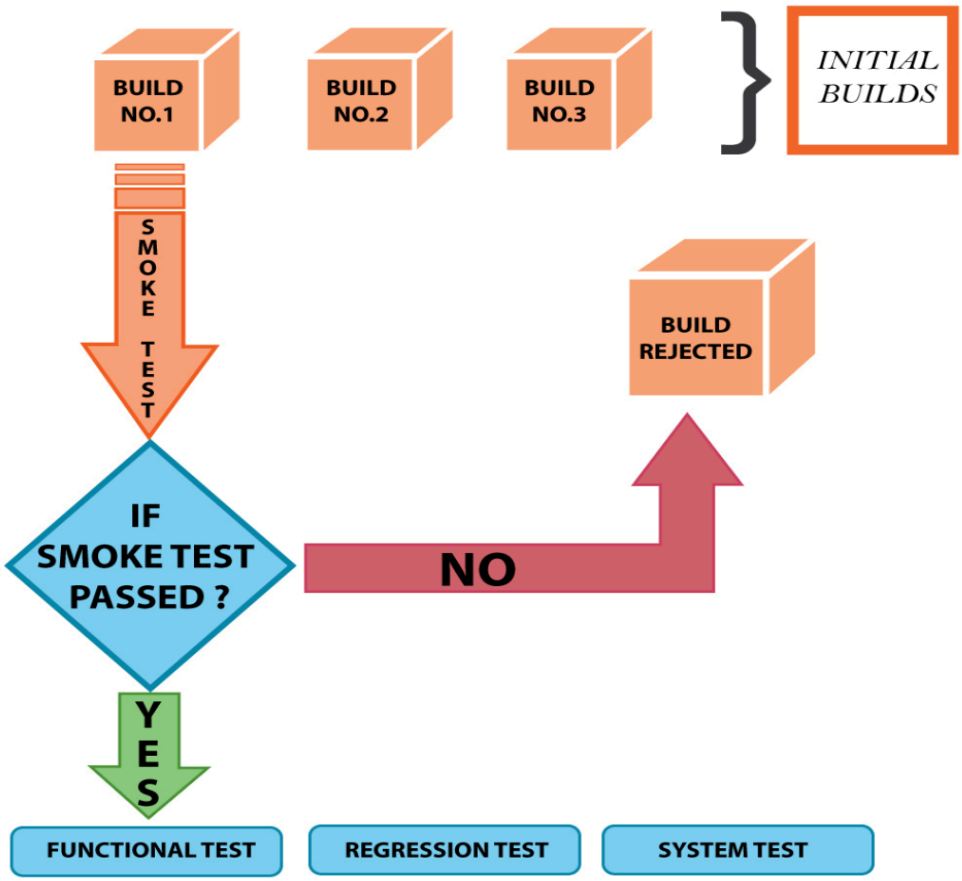
```
      </button>
      <button data-testid="decrement-button" onClick={decrement}>
        Decrement
      </button>
    </div>
  );
};

export default Counter;
```

4. Smoke testing

One of the most important types of testing that needs to be done first for every new release is called smoke testing. When a new version is ready, we check its basic functions right away. Only if it passes this initial check, more detailed testing by QA (Quality Assurance) is done. To make testing less tiring, the developer should focus on testing only the major features. The main goal is to look closely at the most important features of the software as soon as possible.

If a version doesn't pass the smoke test, it goes back to the developer for fixing. Versions that pass all the tests and work well are called stable releases. These stable releases might go through more testing depending on the situation.

For example, let's say a web application for an insurance company adds a new page to check the status of insurance claims. Testers would do smoke tests to make sure the existing version is working at a basic level. They'd check things like whether a user can log in, access the claims status page, and see the status of a specific claim without the app freezing or having issues.



5. System testing

System testing is like checking if the entire software, all its parts, works correctly together. The main aim of this testing is to make sure the software does what it's supposed to do.

People can do this testing by hand, or they can use special tools that do it automatically. Some examples of what can be tested include how the software looks to users or how fast and safe it is.

Here's an example of system testing in a simpler way. Imagine you're checking a website. You want to make sure that when you ask for a list of users, the website gives you the right

list with all the correct details. Then, you also want to check if adding a new user works properly. It's like making sure the different parts of the website play well together.

To do this checking, in the below code example, we use a special library called Supertest with JavaScript. It helps us make requests to the website and see if everything is working as expected. For example, we can test if the website gives us the right list of users when we ask for it or if it adds a new user correctly when we tell it to.

```
const assert = require('chai').assert;
const request = require('supertest');
const app = require('../app');

describe('System Tests', function () {

// getting a list of users
  describe('GET /users', function () {
    it('should return a list of users', function (done) {
      request(app)
        .get('/users')
        .expect(200)
        .end(function (err, res) {
          if (err) return done(err);
          assert.isArray(res.body, 'response should be an
array');
          assert.property(res.body[0], 'name', 'user has a
name');
          assert.property(res.body[0], 'email', 'user has an
email');
          done();
        });
    });
  });

// adding a new user
```

```
  describe('POST /users', function () {
    it('should add a new user', function (done) {
      request(app)
        .post('/users')
        .send({ name: 'John', email: 'john@example.com' })
        .expect(201)
        .end(function (err, res) {
          if (err) return done(err);
          assert.property(res.body, 'id', 'user has an id');
          assert.equal(res.body.name, 'John', 'user has the
correct name');
          assert.equal(res.body.email, 'john@example.com',
'correct email');
          done();
        });
    });
  });

});
```

6. Acceptance Testing

Acceptance Testing is a kind of functional testing used to check if an application or product meets the criteria and requirements set by stakeholders or customers. The focus is on testing the application in various scenarios to ensure it aligns with both functional and non-functional requirements, meeting the expectations and needs of users. It is further categorized in two steps as given below.

Alpha Testing :

Alpha Testing belongs to the software testing types conducted internally by developers or testers before releasing an application to the public. It includes checking the application for any problems or bugs and making necessary fixes before the official release.

Example of Alpha Testing:

Imagine a software development team creating a new video editing application. During Alpha Testing, the team uses the application extensively, trying out various features and tools to identify any issues. This ensures that the application is refined and polished before reaching the hands of the general public.

Beta Testing:

Beta Testing is a testing type carried out by a group of end-users or customers before the official release but in a real-world environment. It gathers information on user experience, functionality, and application performance. Feedback obtained during Beta Testing is valuable for addressing issues or bugs to enhance the overall user experience.

Example of Beta Testing:

Consider a social media platform introducing a new feature for sharing live videos. Beta Testing involves allowing a selected group of users to try out this feature before it's available to everyone. The feedback collected during this phase helps the development team refine the feature, ensuring it performs well and meets user expectations when released to the wider audience.

7. Security Testing

Security testing aims to pinpoint vulnerabilities and potential threats within a software application. This involves scrutinizing the application for weaknesses like SQL Injection, Cross-Site Scripting (XSS), and other security risks. The testing process encompasses both manual examination and automated techniques, such as penetration testing, vulnerability scanning, and code reviews.

Let's consider a simplified example in a web application where user input is used in constructing SQL queries without proper validation, leading to a potential SQL Injection vulnerability.

```
// Node.js and SQL example using the 'mysql' library
const mysql = require('mysql');

// User input (in a real-world scenario, this might come from user input in
a web form)
const userInput = "'; DROP TABLE Users; --";

// Creating a SQL query without proper validation (vulnerable to SQL
Injection)
const vulnerableQuery = `SELECT * FROM Users WHERE username =
'${userInput}'`;

// Creating a SQL query with proper parameterization to prevent SQL
Injection
const secureQuery = 'SELECT * FROM Users WHERE username = ?';

// Creating a connection to a MySQL database (replace with your actual
database connection details)
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'your_username',
  password: 'your_password',
  database: 'your_database'
});

// Attempting to execute the vulnerable query
connection.query(vulnerableQuery, (err, results) => {
```

```
  if (err) {
    console.error('Error executing vulnerable query:', err);
  } else {
    console.log('Results of vulnerable query:', results);
  }
});

// Executing the secure query using parameterization to prevent SQL
Injection
connection.query(secureQuery, [userInput], (err, results) => {
  if (err) {
    console.error('Error executing secure query:', err);
  } else {
    console.log('Results of secure query:', results);
  }
});

// closing  the database connection
 connection.end();
```

In this example, the vulnerableQuery is susceptible to SQL Injection because it directly incorporates user input into the SQL query string. The secureQuery uses parameterization, a secure coding practice, to prevent SQL Injection. Always employ parameterized queries or prepared statements to mitigate the risk of SQL Injection in real-world scenarios.

8. Performance Testing

Performance testing, a non-functional testing type, evaluates how a software application performs under diverse conditions. Several techniques are employed for this assessment. Some of them are given below.

Load Testing

Load testing focuses on app performance under both typical and peak loads. The primary objective is to ensure the application handles anticipated loads without encountering performance issues.

Stress Testing

Stress testing is employed to assess how an application performs under extreme loads or adverse conditions. This helps identify the application's breaking points and examines its capability to manage unexpected or abnormal loads.

Scalability Testing

Scalability testing gauges the application's capacity to adapt to increased workloads or user numbers. The aim is to determine if the application can seamlessly adjust to varying workloads, either scaling up or down as required.

Stability Testing

Stability testing is applied to ascertain the reliability and consistency of an application over an extended period. The goal is to guarantee the application maintains consistent performance over time, devoid of degradation or failures.

Automatic Testing tools:

Automatic testing tools are crucial in software development for their efficiency, consistency, and repeatability. They execute tests quickly, ensuring uniformity and reliable results. The repeatability of automated tests is valuable in agile environments, where code changes frequently. While there's an initial setup cost, automated testing proves cost-effective by saving time in the long run. It excels in regression testing, detecting issues early and supporting parallel execution for faster results. Overall, these tools contribute to enhanced code quality and accelerated development processes. Some of the well known auto-testing tools are -

1) Selenium:

Type: Web Application Testing

Selenium is widely used for automating web browsers. It supports multiple programming languages and browsers, making it a versatile choice for web application testing.

2) JUnit:

Type: Unit Testing (Java)

JUnit is a popular testing framework for Java that facilitates unit testing. It provides annotations for test methods, assertions, and test fixtures.

3) Appium:

Type: Mobile Application Testing

Appium is an open-source tool for automating mobile applications on Android and iOS platforms. It supports native, hybrid, and mobile web applications.

4) Cypress:

Type: End-to-End Testing (Web)

Cypress is a JavaScript-based end-to-end testing framework for web applications. It provides real-time reloading, fast execution, and supports various browsers.

5) Postman:

Type: API Testing

Postman is a widely used tool for API testing. It allows you to create and send HTTP requests to test APIs, automate workflows, and analyze responses.

6) TestNG:

Type: Testing Framework (Java)

TestNG is a testing framework inspired by JUnit and NUnit but designed to be more powerful. It supports parallel execution, test parameterization, and grouping.

Tips :

Developing a robust test suite involves a strategic blend of different testing approaches. Keep the following tips in mind:

1. Prioritize Test Cases:

Identify critical functionalities and prioritize writing test cases for those scenarios. This ensures that crucial parts of your code undergo thorough testing.

2. Automate Your Tests:

Automation is the linchpin for maintaining a consistent and efficient testing process. Utilize tools like JUnit, Pytest, or Jasmine to automate test case execution, saving time and minimizing errors.

Conclusion:

In wrapping up, mastering code testing approaches is paramount for any aspiring developer aiming to produce reliable and high-quality software. By delving into testing approaches we lay a solid foundation for our projects. Remember, testing is not a one-off task but an ongoing practice that contributes to the enduring success of your applications. Testing code is like putting together puzzle pieces. Unit testing looks at each tiny piece, integration testing checks how the pieces fit, and system testing looks at the whole picture. By using a mix of manual testing and automatic tools, like checking for security or how fast the software runs, we make sure our software is safe and works really well. Testing is like a journey that helps us create software that not only works but goes beyond what people expect. As we keep learning about making software, having a good plan for testing remains super important.

Bibliography:

1. Atlassian. (n.d.). "Types of Software Testing." Retrieved from https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

2. OpenAI. (n.d.). Used ChatGPT and my previous projects for code examples.

3. Hunt, A., & Thomas, D. (2013). "The Pragmatic Programmer: Your Journey to Mastery." Pragmatic Bookshelf.

4. Parasoft. (n.d.). "Software Testing Methodologies Guide: A High-Level Overview." Retrieved from https://www.parasoft.com/blog/software-testing-methodologies-guide-a-high-level-overview/