Test Automation Overview

Efficiency and Accuracy in Software Testing

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

sli.do

#QA-Auto-BackEnd

Table of Contents

1. What is test automation?
2. Test Automation Evolution
3. When to Automate?
4. Test Automation Frameworks
5. Back-End testing and tools
6. CI / CD pipeline

What is Test Automation?

Test Automation Defined

What is Test Automation?

- **Utilizing software** to **configure** and **execute test scenarios** based on **predetermined conditions**, to **verify** the consistency of **actual results** with their **expected counterparts**
- **Functionality**:
  - The automation software is designed to **input test data** into the System Under Test
  - Perform **comparisons** between **expected** and **actual outcomes**
  - Produce **comprehensive test reports**

The Need for Automation Testing

- **Speed:** Automation Scripts are **fast** when compared to manual testing efforts
- **Reliable:** Tests perform **precisely** the same operations each time they are run, there by eliminating human error
- **Repeatable:** Tests can be **repeated n number** of times for execution of the same operation
- **Coverage:** Automated tests **increase coverage**
- **Reusable:** Tests can be **reused on different versions** of an application, even if the user interface changes

Automated testing vs. Manual testing

- Pros of **Automated testing:**
  - Saves time with repetitive test runs
  - Quickly detects regressions in changing code
  - Increases test coverage across different environments
  - Allows parallel testing on multiple devices
  - Frees testers for in-depth analysis and complex tasks

- Pros of **Manual Testing:**
  - Cost-effective for tests run only a few times
  - Enables ad-hoc testing to uncover unexpected bugs
  - Increases chances of finding real user issues through exploratory testing
  - More tester interactions leads to discovering more real-world issues

Automated testing vs. Manual testing

- Cons of **Automated testing**:
  - Initial setup and scripting are costly
  - Some complex tests can't be automated
  - Maintenance of test scripts over time

- Cons of **Manual testing:**
  - Manual execution is slower and labor-intensive
  - More human resources and physical hardware needed
  - Repetitive tests can lead to tester fatigue and boredom

When to Automate Tests?

- **Regression Testing:** When the software application is fairly stable and only regression tests need to be executed
- **Smoke Testing:** For getting a quick high-level assessment on the quality of a build and making go / no-go decision on further testing
- **Static & Repetitive Tests:** For automating testing tasks that are repetitive and relatively unchanging from one test cycle to the next
- **Data Driven Testing:** For testing application functions where the same functions needs to be validated with lots of different inputs & large data sets (i.e. login, search)
- **Load & Performance Testing:** No viable manual alternative exists

When Test Automation is not the best solution?

- **Frequent Changes:** For applications still under development, or frequently changing UI, creating automated test scripts may be a waste of time
- **Subjectiveness**: For application functions that require subjective validation such as usability, simplicity or look-and-feel, manual testing is more appropriate
- **Localization:** Testing localized content requires an understanding of the language, culture and local norms. These are best performed manually
- **One-timers:** The investment in developing test scripts pays of, if the test is repeated many times. It may not be worthwhile for one timers

Test Automation Evolution

Brief History

Evolution of Test Automation

- **Early Days of Test Automation (1960s-1980s)**
  - Originated with mainframe computing, focusing on automating basic tasks
  - Primarily command-line based, limited in capabilities, and high maintenance
- **Script-Based Testing (Late 1980s to 1990s)**
  - Manual creation of test scripts, with a focus on record-and-playback techniques.
  - Early versions of tools like HP QuickTest Professional (QTP), now UFT

Evolution of Test Automation

- **1990s and the Rise of Graphical User Interfaces (GUIs)**
  - Challenges arose with the advent of GUIs, requiring tools for simulating user interactions
  - Development of GUI automation tools
- **Framework Development Era (Early 2000s)**
  - From ad-hoc scripting to structured test automation frameworks
  - Development of modular, data-driven, and keyword-driven frameworks
  - Selenium, TestNG, and JUnit gaining popularity

Evolution of Test Automation

- **Integration with Development Practices (Mid to Late 2000s)**
  - Closer integration with Agile methodologies
  - Continuous testing as part of Continuous Integration (CI)
  - Introduction of Jenkins for CI
- **Behavior-Driven Development (BDD) and Test-Driven Development (TDD) (2010s)**
  - Emphasis on BDD and TDD, writing tests before code
  - Adoption of Cucumber for BDD, NUnit for TDD

Evolution of Test Automation

- **Rise of Mobile and Cross-Platform Testing (2010s)**
  - Diverse testing strategies for the explosion of mobile devices
  - Introduction of Appium for mobile automation
- **AI and Machine Learning Integration (Late 2010s to Present)**
  - AI and ML for predictive analytics, test maintenance, smarter test generation, and anomaly detection
  - Enhanced efficiency, reduced maintenance costs, and improved test coverage

Evolution of Test Automation

- **DevOps and Continuous Testing (2020s)**
  - Test automation integral to DevOps, focusing on continuous testing
  - Integration into CI/CD pipelines
  - Advanced CI/CD tools like GitLab CI, CircleCI
- **Current Landscape and Future Trends**
  - Test automation as a critical part of the software development lifecycle
  - AI-powered automation, API automation, continuous testing, low-code/no-code automation tools
  - As AI and ML advance, test automation becomes more sophisticated and efficient

Automation Frameworks

Strategies and Tools for Efficient Software Testing

What is an Automation Framework?

- A **testing framework** is a **set of guidelines** or **rules** used for creating and designing test cases
- Comprised of a combination of **practices and tools** that are designed to help QA professionals to **test more efficiently**
- Framework **Components**:
  - May include specific **coding standards** for test scripts
  - **Strategies** for managing test data effectively
  - **Utilization** of **object repositories** to maintain test elements
  - **Systems** for **logging** and **retrieving** test outcomes
  - **Guidelines** for **interacting** with **external resources** and systems

Why the Need of a Testing Framework?

- If each project requires a different testing strategy, it can **prolong the time** for testers to become efficient
- Using a **single, application-independent** testing framework avoids the need to modify the automation setup for new applications
- Organized testing frameworks **prevent the duplication** of test cases
- Test frameworks help in systematically arranging test suites, thereby **improving testing efficiency**

Types of Testing Frameworks

- Linear Automation Framework
- Modular Based Testing Framework
- Library Architecture Testing Framework
- Data-Driven Framework
- Keyword-Driven Framework
- Hybrid Testing Framework

Linear Automation Framework
Hybrid Testing Framework
Modular Driven Framework
Top Test Automation Frameworks
Keyword-Driven Framework
Library Architecture Framework
Data-Driven Framework

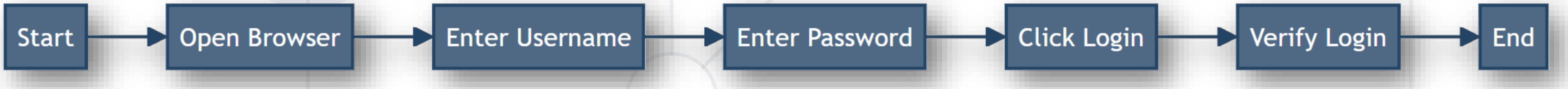
Linear Automation Framework

- Characterized by a **sequential approach** to test script execution without modular division
- Series of test scripts that are executed in a **specific order**
- No abstraction layer; each test script interacts directly with the application, performing **actions step by step** as a user would
- If application flow changes, the test script might need changes too
- **Straightforward** and doesn't require the complexity of understanding modules
- Suitable for **simple test cases** or **smaller applications**

Linear Automation Framework Example

- **Example:** A **tester records actions** in a test environment which is then replayed by the script

```
public void TestLogin()
{
    OpenBrowser("http://example.com/login");
    EnterText("usernameField", "testUser");
    EnterText("passwordField", "testPass");
    ClickButton(“loginButton");
    Assert.IsTrue(CheckIfLoginSucceeded());
}
```



Modular Testing Framework

- Divides the application into **separate units** or **modules** for test script creation
- Allows for **test script reuse** and **hierarchical script building**
- Enhances test maintenance and **reduces duplication** by isolating changes to individual modules
- Promotes reusability and scalability with **organized and focused scripts**
- Suitable for applications with clearly defined modular structures

Modular Testing Framework Example

- **Example:** Creating **separate test scripts** for the login page, dashboard, and user settings in a web application

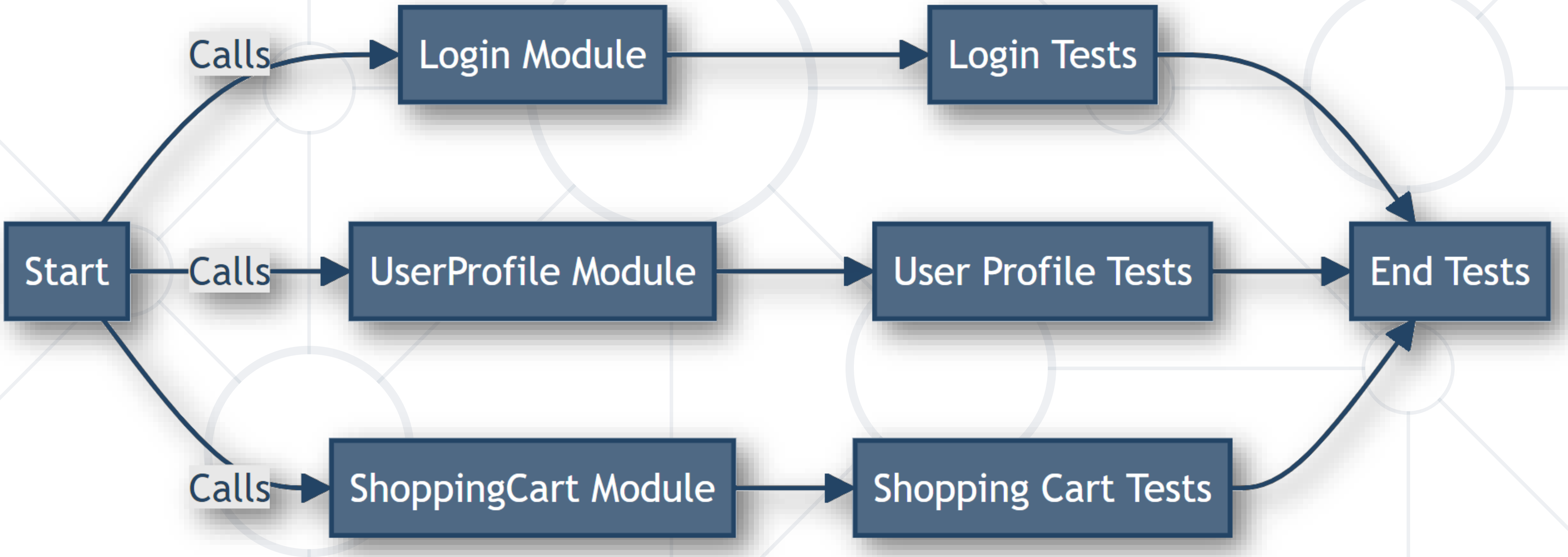
```
// Reusable login function
public void Login(string username, string password)
{
    OpenBrowser("http://example.com/login");
    EnterText("usernameField", username);
    EnterText("passwordField", password);
    ClickButton("loginButton");
}
```

Modular Testing Framework Example

```
public void TestLoginModule()
{
    Login("testUser", "testPass");
    Assert.IsTrue(CheckIfLoginSucceeded());
}

public void TestUserProfileModule()
{
    Login("testUser", "testPass");
    NavigateToUserProfile();
    Assert.IsTrue(CheckUserProfileDetails());
}
```

Modular Testing Framework Diagram



Library Architecture Testing Framework

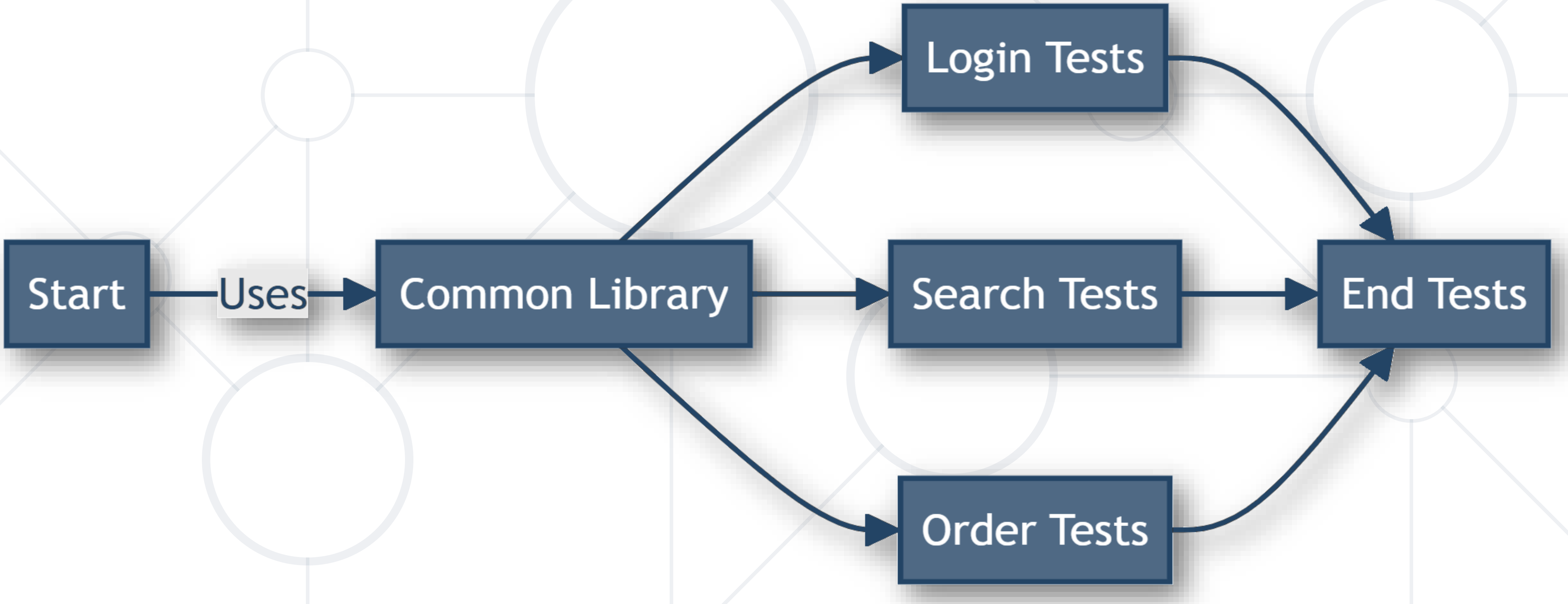
- Extends the modular framework by creating common functions in a **shared library**
- **Reduces script redundancy** by utilizing library functions across multiple test scripts
- Encourages **better organization** and **maintenance** of common tasks within the tests
- Facilitates **consistency** in test execution and validation methods
- Ideal for **applications with many common operations** or functions
- **Example:** A common method for **file uploading** used by different modules

Library Architecture Testing Framework Example

```
// Common library function for entering text into fields
public void EnterTextIntoField(string fieldId, string text)
{
    FindFieldById(fieldId).EnterText(text);
}

public void TestLogin()
{
    OpenBrowser("http://example.com/login");
    EnterTextIntoField("usernameField", "testUser");
    EnterTextIntoField("passwordField", "testPass");
    ClickButton("loginButton");
    Assert.IsTrue(CheckIfLoginSucceeded());
}
```

Library Architecture Testing Framework Diagram



Data-Driven Framework

- **Separates test scripts** from data, **storing data externally**, typically in databases or files
- Enables running the **same test script with multiple** sets of **data** inputs
- Useful for **validation** of the **same functionality** under different input conditions
- Enhances the **flexibility** and extendibility of test scripts
- Ideal for scenarios requiring testing with **varied data sets**

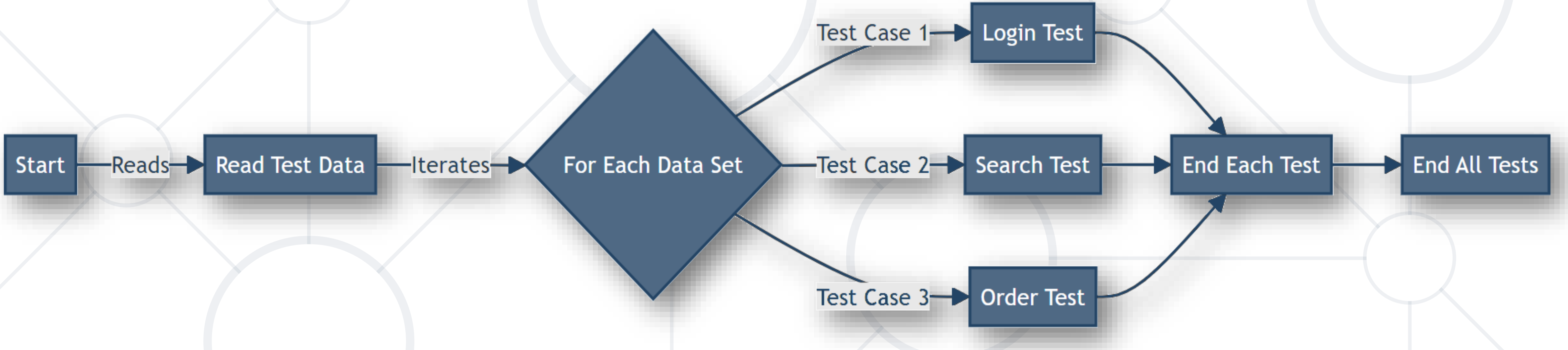
Data-Driven Framework Example

- **Example:** An Excel file stores login credentials to run the same login test script for different users

```
public void TestLoginWithData(string username, string password)
{
    OpenBrowser("http://example.com/login");
    EnterText("usernameField", username);
    EnterText("passwordField", password);
    ClickButton("loginButton");
    Assert.IsTrue(CheckIfLoginSucceeded());
}

// This method would be called with different sets of data
TestLoginWithData("user1", "pass1");
TestLoginWithData("user2", "pass2");
```

Data-Driven Framework Diagram



Keyword-Driven Framework

- Uses **keywords** to **represent user actions** in the application under test
- Allows testers to **write test cases** without deep knowledge of scripting languages
- Facilitates **easier readability** and understanding of test cases
- Enhances test maintenance by **separating** the **technical implementation** from the **test case design**
- Suitable for **teams with varied skill levels** and for applications with a large number of user interactions

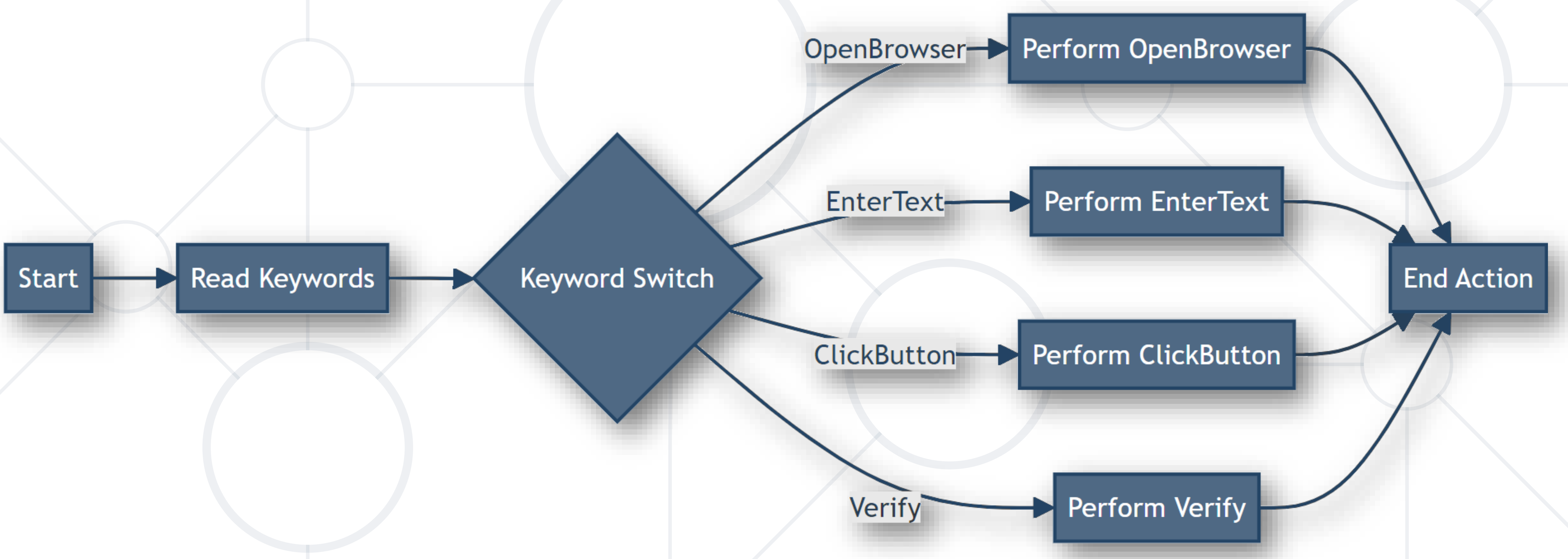
Keyword-Driven Framework Example

- **Example:** A spreadsheet **uses keywords** like "ClickButton" with associated data to direct the framework's actions

```
public void ExecuteAction(string keyword, string param)
{
    if (keyword == "OpenBrowser") OpenBrowser(param);
    else if (keyword == "EnterText") EnterText("inputField", param);
    else if (keyword == "ClickButton") ClickButton(param);
    // ...and so on for other keywords
}

// These actions would be read from an external source, like a spreadsheet
ExecuteAction("OpenBrowser", "http://example.com/login");
ExecuteAction("EnterText", "testUser");
ExecuteAction("EnterText", "testPass");
ExecuteAction("ClickButton", "submitButton");
```

Keyword-Driven Framework Diagram



Hybrid Testing Framework

- **Combines features** of two or more of the aforementioned frameworks to utilize their benefits
- **Customizable** to fit the project's specific needs and complexities
- Offers **flexibility** in test design, execution, and maintenance
- Can **adapt to changes** in the application with minimal impact on the existing tests
- Suitable for **complex applications** and environments where no single framework suffices

Hybrid Testing Framework Example

- **Example:** A combination of modular-based structure, data-driven elements for input, and keyword-driven actions

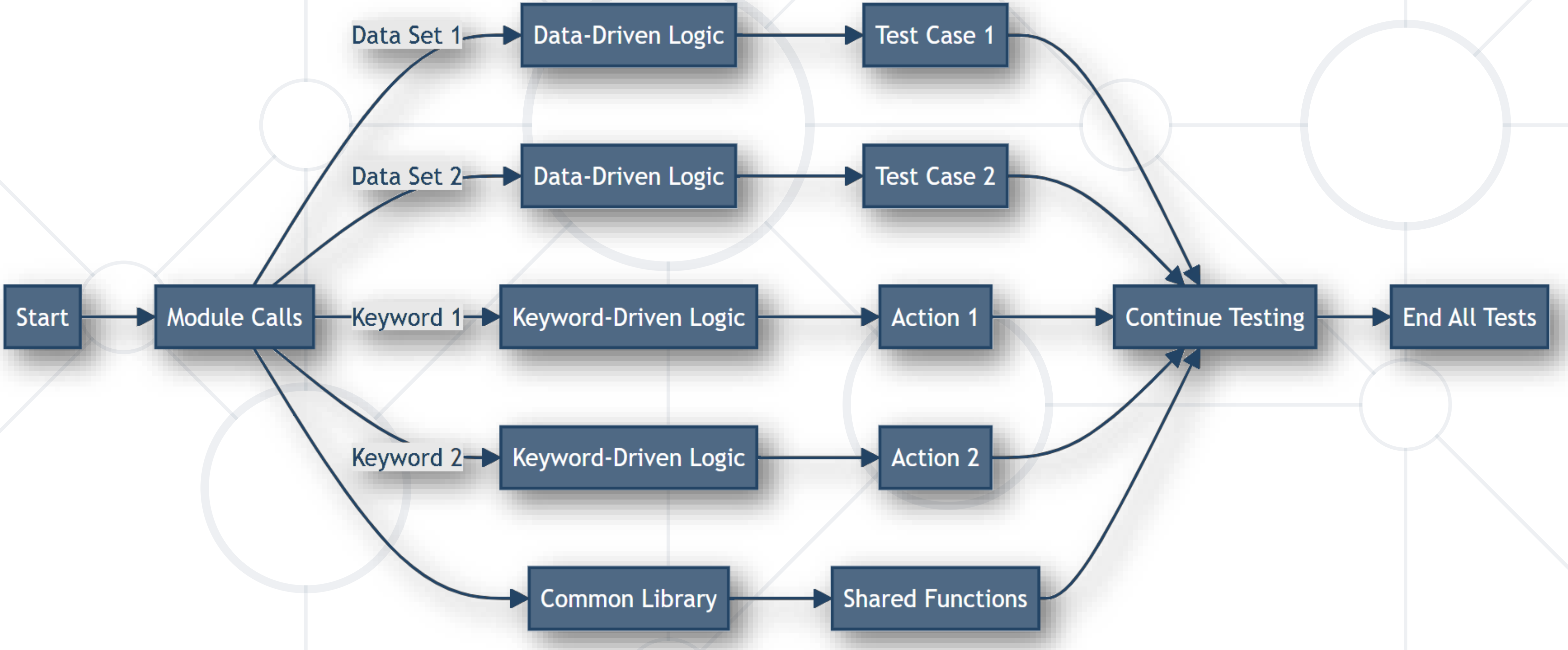
```
// A hybrid approach that combines modular, data-driven,
and keyword-driven concepts
public void ExecuteTest(string module, Dictionary<string, string>
testData)
{
    foreach (var action in testData)
    {
        ExecuteAction(action.Key, action.Value);
    }
    Assert.IsTrue(CheckModuleOutcome(module));
}
```

Hybrid Testing Framework Example

```
// Data and module-specific actions would be loaded
from external sources
Dictionary<string, string> loginData =
                             new Dictionary<string, string>
{
    { "OpenBrowser", "http://example.com/login" },
    { "EnterText", "testUser" },
    { "EnterText", "testPass" },
    { "ClickButton", "submitButton" }
};

ExecuteTest("LoginModule", loginData);
```

Hybrid Testing Framework Diagram



Key Concepts for Framework Selection

- **Complexity of the Application:** Complex applications might benefit from a Hybrid or Data-Driven Framework
- **Frequency of Tests:** For tests that run often, an Keyword or Data-Driven Framework is ideal
- **Team Expertise:** If the team excels in coding, a Library Architecture or Modular Based Framework may be preferred
- **Project Timeline:** For projects with tight deadlines, a Linear Framework might be quickest to implement
- **Application Updates:** Frequent updates may require a Keyword-Driven or Data-Driven Framework for agility

Practical Considerations

- **Code Level:** Does the framework require writing code (e.g., Selenium) or is it codeless (e.g., Leapwork)
- **Platform Specificity:** Is the framework specialized for web, iOS, Android, or cross-platform (e.g., Appium for mobile, Cypress for web)
- **Budget:** Does the project budget allow for commercial tools (e.g., Ranorex) or should it rely on open-source (e.g., Robot Framework)
- **Tester Skill Set:** Preference or skill set for scripting tests or using record-and-playback features (e.g., Katalon Studio)?

Framework Selection in Reality

- **E-commerce Website:** For managing the diverse functionalities of an e-commerce platform, a Modular Based Framework like Selenium with a structured approach can be utilized
- **Gaming App:** A Hybrid Framework that combines Appium for Data-Driven tests covering user scenarios across different devices and operating systems, with Cucumber for Keyword-Driven tests to define UI elements behavior in a readable format
- **Banking Software:** For ensuring the security and reliability of banking software, a Library Architecture Framework like Robot Framework can be used

Back-End Testing and Tools

Testing the Server-Side (Back-End) Components

Back-End Testing

- The process of testing the server-side (back-end) components, including **databases, server logic, and APIs**
- **Integration testing** to ensure components work together, **API testing** for external interfaces, **performance testing** for speed and stability
- When **selecting tools**, consider compatibility with tech stack, community support, ease of integration, and scalability to match project needs

API Testing

- **Tools for API Testing:**
  - **RestSharp** is a .NET library that simplifies HTTP requests and is used for testing APIs
  - Using RestSharp with NUnit in C# allows to simulate API calls and evaluate the responses, ensuring components interact correctly
  - **Postman** is a versatile tool for testing APIs interactively, which can be automated with **Newman** for command-line execution

Performance Testing

- **Performance Testing Tools**
  - **JMeter:** A versatile open-source tool designed for performance and load testing. Enables simulation of heavy loads and analysis of server performance under different conditions
  - **K6:** Designed to test the performance of backend services. Scriptable tests, complex user scenarios, integrates well with CI/CD pipelines

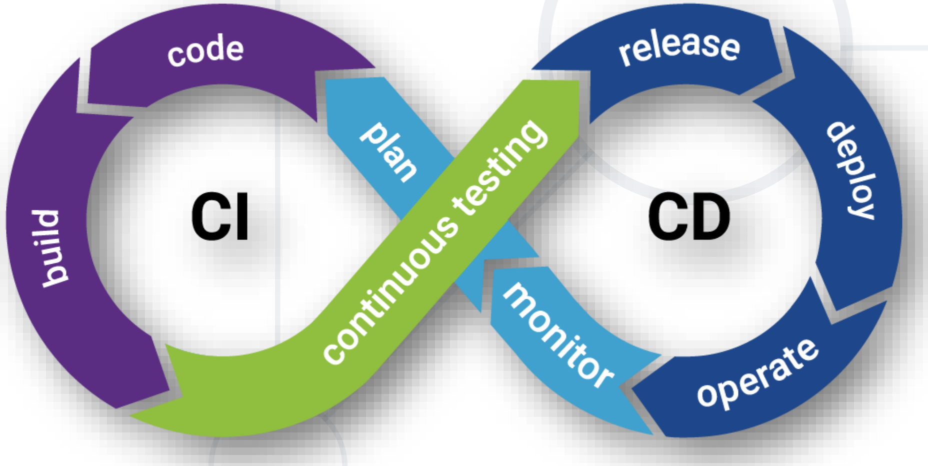
Code Quality

- **Code Quality Standards and Tools:**
  - **SonarQube:** Detailed reports on bugs, vulnerabilities, code smells, and code complexity
  - **Plato:** An analysis tool for JavaScript offering insights into code complexity and structure. It generates reports on maintainability and potential vulnerabilities
  - **ESLint:** A configurable tool for ECMAScript/JavaScript, identifying and reporting code patterns

CI/CD Pipeline

CI/CD Pipeline

- **Continuous Integration (CI):** Developers merge code changes into a central repository frequently, where automated builds and tests are run
- **Continuous Delivery (CD):** Code changes are automatically built, tested, and prepared for a release to production



CI/CD Pipeline Example

- A **web application development team** uses Jenkins, a popular CI/CD tool, to automate their development process
- Every time a developer **pushes code** to the **version control** system (e.g., GitHub), Jenkins automatically **triggers a build** and **runs a series of tests**
- If the **build and tests pass**, Jenkins then **deploys the changes** to a staging environment for further automated or manual tests
- Once the **code passes all staging tests**, it's **automatically deployed to production**, making the latest features or fixes available to users

Selection Process for Testing and CI/CD Tools

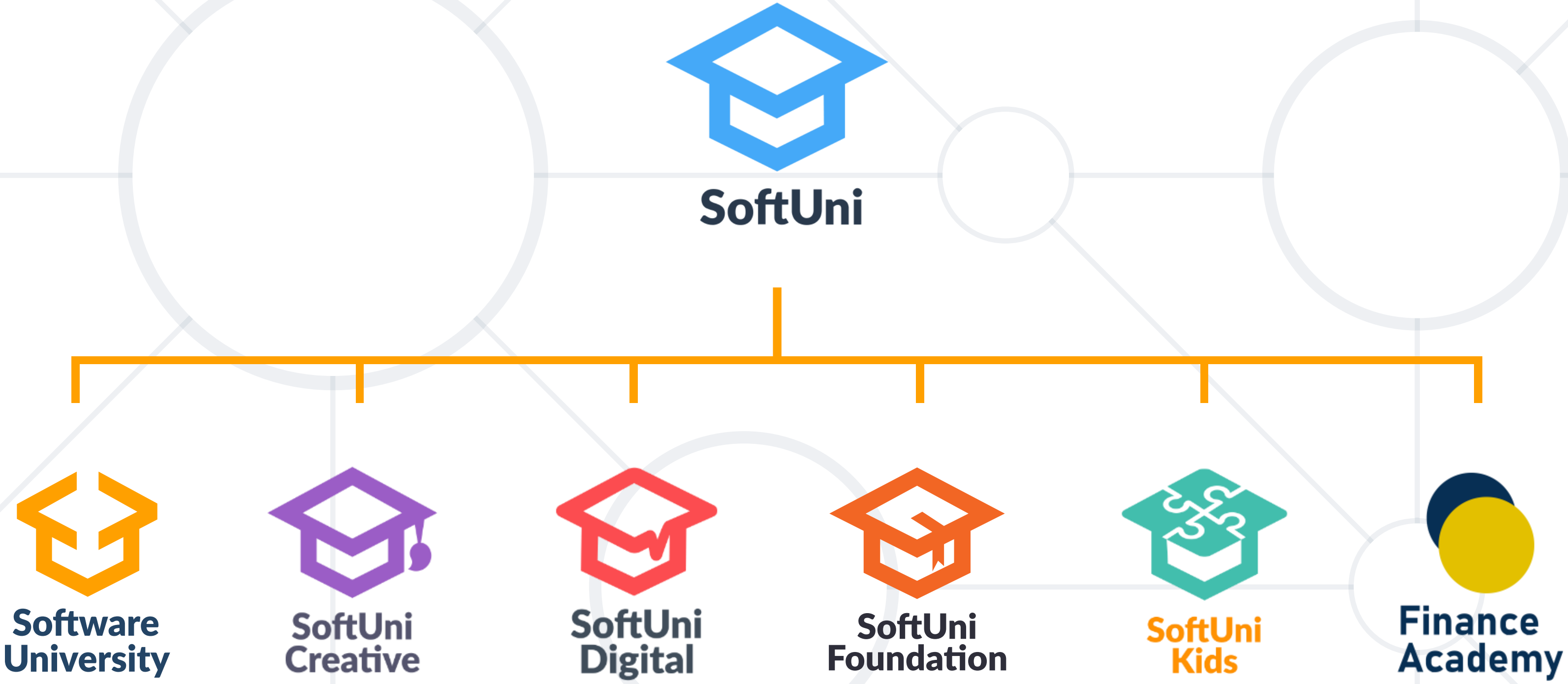
- **Define Requirements:** Identify what you need from a tool (e.g., API testing, performance testing)
- **Research and Shortlist:** Look for tools that fit those needs
- **Evaluate Against Criteria:** Assess tools based on compatibility, community support, ease of integration, and scalability
- **Trial and Pilot:** Test tools on a small scale to see how well they integrate with your systems
- **Gather Feedback:** Include team members in the evaluation process
- **Make a Decision:** Choose the tool that best fits your project's needs

Summary

- Automated software testing process;
- Progression from manual to automated;
- When to Automate? Repetitive, regression, large-scale tests
- Test Automation Frameworks: Structured testing method collections;
- Back-end Testing and Tools: Server-side testing; Tool-dependent;
- CI/CD Pipeline: Automate build, test, deploy;

Software University

Questions?



SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

Software
University

License