TEKsystems Global Services | WORKFORCE DEVELOPMENT

Configuration Management and DevOps: Automate Infrastructure Deployments

PARTICIPANT GUIDE

Content Usage Parameters

Content refers to material including instructor guides, student guides, lab guides, lab or hands-on activities, computer programs, etc. designed for use in a training program

1

Content is subject to copyright protection

2

Content may only be leveraged by students enrolled in the training program

3

Students agree not to reproduce, make derivative works of, distribute, publicly perform and publicly display content in any form or medium outside of the training program

4

Content is intended as reference material only to supplement the instructor-led training

Develop a passion for learning

Microservices Application Architecture

Concepts, Principles and Practices

Logistics

- Class Hours:
- Instructor will provide class start and end times.
- Breaks throughout class

- Telecommunication:
- Turn off or set electronic devices to vibrate
- Reading or attending to devices can be distracting to other students

- Lunch:
- 1 hour 15 minutes

- Miscellaneous
- Courseware
- Bathroom

Develop a passion for learning

Day 4

Testing and Quality

Testing

More than the act of testing, the act of designing tests is one of the best bug preventers known.

The thinking that must be done to create a useful test can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest.

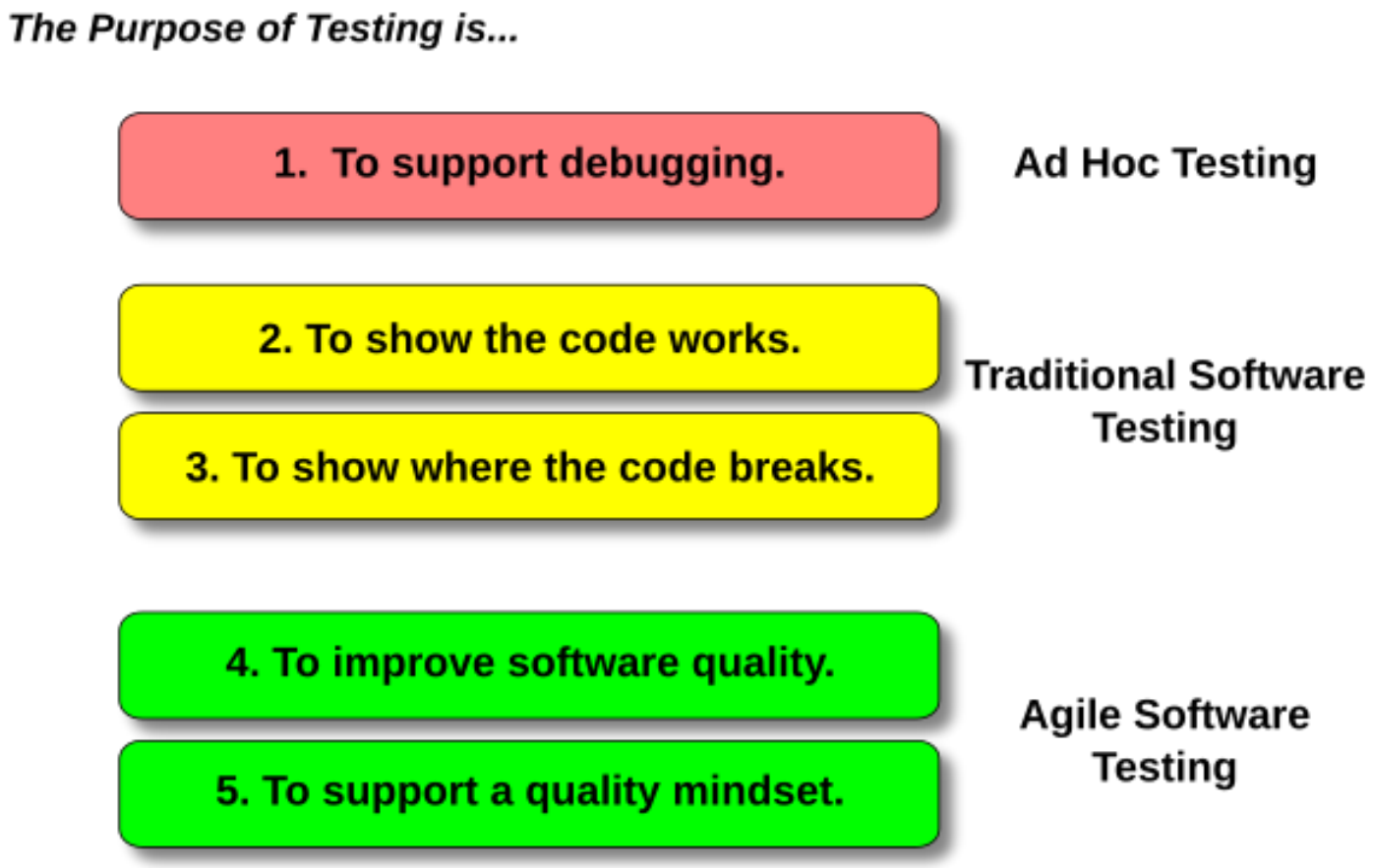
If you can't test it, don't build it.
If you don't test it, rip it out.

Boris Beizer

The Testing Community

- Testing and Quality Management is a discipline
  - Like accounting, it has a core of theory, practices and applications
  - These can be applied to any number of areas
  - Engineering, software, social science, physical sciences, etc.
  - The theory and practices can be tailored to accommodate the requirements of each area they apply to
- Testing measures and evaluates
  - But measurements are meaningless without a context to interpret what the test results mean
  - Quality control and quality management provide that context

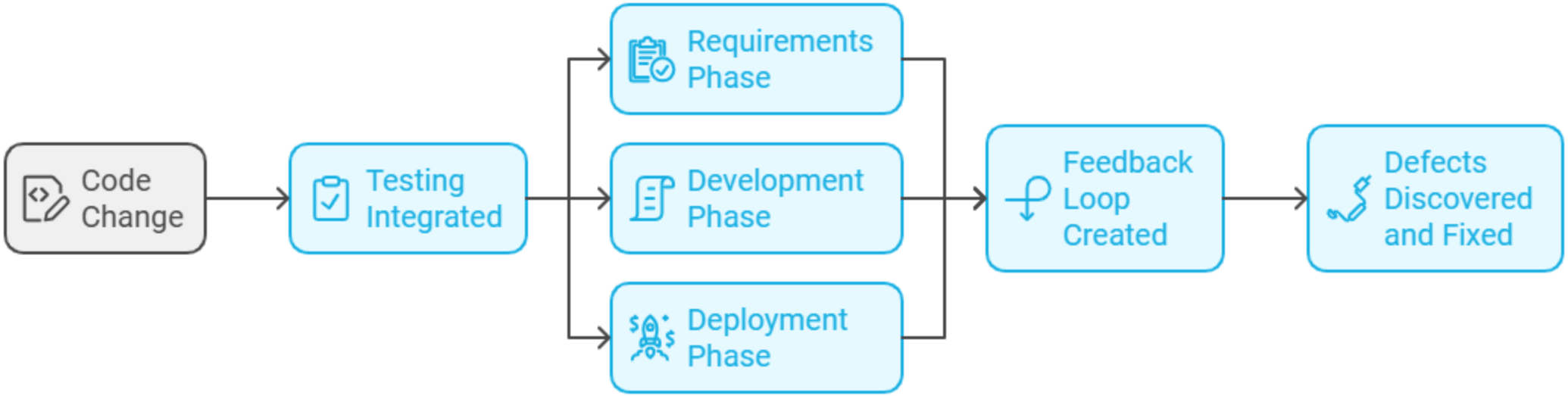
Beizer's Phases in a Tester's Mental Life



Quality is Everyone's Responsibility

- **Shared Responsibility**: Quality is not only the responsibility of testers, but all members of the team, from product owners, to developers, to testers.
- **Team Collaboration:** Teamwork, communication, and collaboration ensure a better understanding of the quality requirements.
- **Quality mindset:** The team needs to embrace a culture of quality where they all strive to achieve and maintain a high quality product.

Testing in Agile



- **Continuous Integration**: In agile development, we must embrace continuous integration where testing must be done at every code change.
- **Frequent Testing:** Testing is integrated into all phases of development from requirements to deployment.
- **Feedback Loops**: A strong testing methodology creates a fast feedback loop, to discover and fix defects as soon as possible.

Mission Critical Software and Risk Mitigation

- Grady Booch defines mission critical software as:
  - Software that is so critical to the functioning of the organization that if it fails, then the organization cannot function
  - Because it is mission critical, it tends to be so complex that it exceeds the intellectual comprehension of a single individual
  - This complex software is often referred to as “industrial strength”
  - The results of software failures and errors of mission critical often tend to have catastrophic results
  - One of the main goals of software testing of mission critical systems is risk mitigation
- Microservices are often used to implement mission critical software

Test Plan Overview

- **Test Scope**: Define what is included and what isn't.
- **Test Environment**: Describe the test environment, which can be production, or a staging environment.
- **Metrics**: List what you will measure as well as how the results will be evaluated.
- **Roles and Responsibilities**: Define who will do what in the testing phases.
- **Acceptance Criteria**: Create a list of quantifiable acceptance criteria to be met before the test can be considered successful.

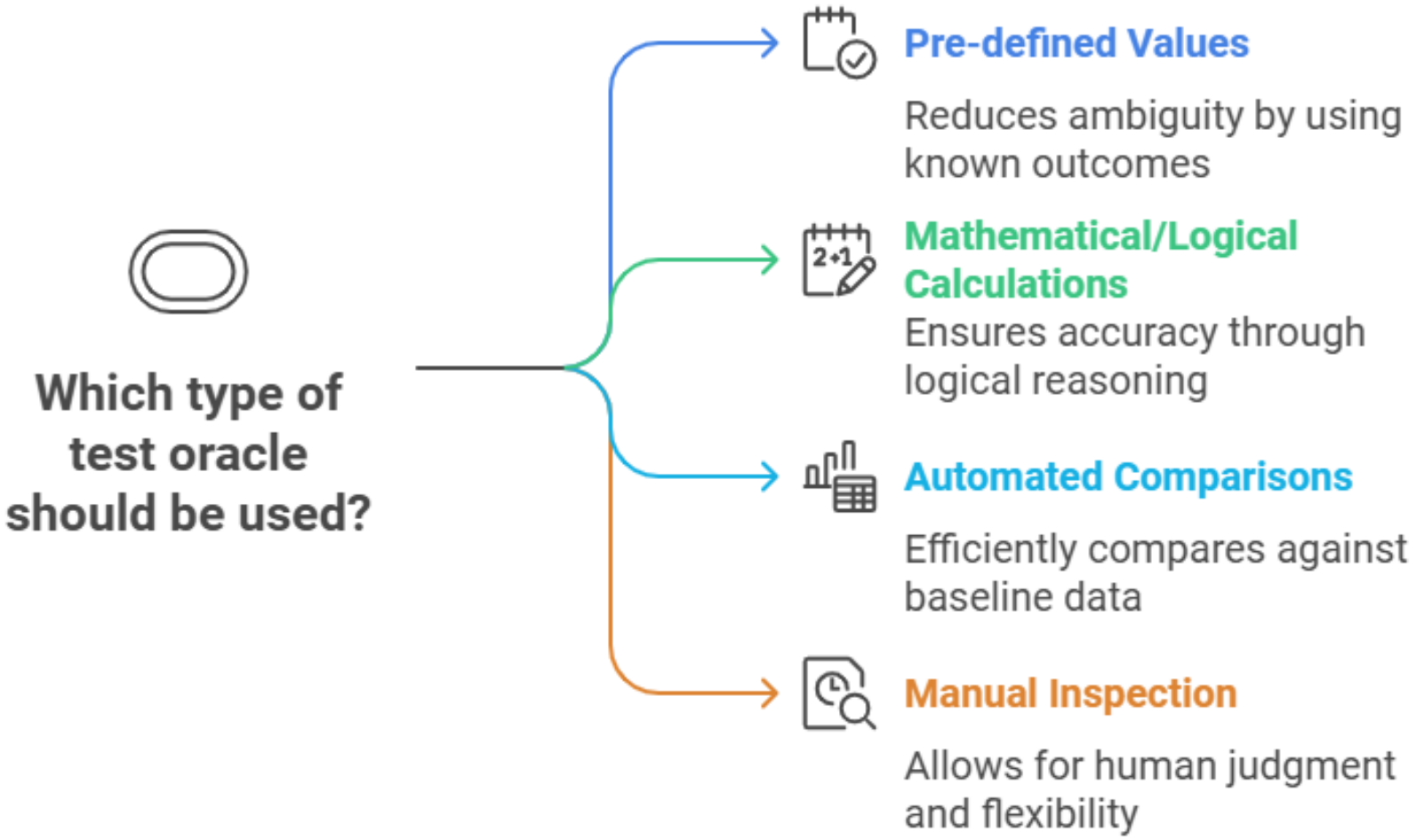
Understanding Test Data

- **Real Data:** Test data should reflect real world scenarios of users using the application.
- **Representative Data**: It must capture the various situations and edge cases to ensure that the application is production-ready.
- **Good Test Data:** Well designed test data will provide good coverage of all features and functionality.

Test Case Characteristics

- **Unique Identity:** Each test case should have its own identifier that you can use to track and reference.
- **Clear Description:** A proper description of the test that outlines its purpose, and what is being tested.
- **Steps to Execute:** The test case should outline the exact steps that are required for a person or automation to complete the test.
- **Expected Results:** A properly defined test case should clearly list out the expected outcome, which can be used to verify success or failure of a test case.

Test Case Oracles



Good Testing

- Testing that is done badly produces results that are worthless
  - This is true in all areas testing is done
- There are four primary criteria for good testing
  - Testing is ***valid*** when we are actually testing what we think or claim we are testing
  - Testing is ***accurate*** then it has low rates of false positives and false negatives
  - Testing is ***reliable*** when the results of test depend only on the tests and not some other factor such as how the tests are being run
  - Testing is ***comprehensive*** when nothing has “slipped through the cracks”
- If we are using tests to guide development, bad test results give us bad guidance

Sources of Testing Errors

- Test Design Errors
  - Errors made in designing tests, choosing test data, selection of pass-fail criteria, composition of test suits and other methodological mistakes
- Test Procedural Errors
  - Errors resulting from running tests and analysis results incorrectly, even if the test themselves have no design errors
- Human Errors
  - Errors that testers make because we are people
  - Includes social influences, perceptual errors and cognitive quirks
  - These are controlled by following rigorous test protocols and procedures

Quality Control

- How do we know if a test passes?
  - There must be an evaluation criteria to decide if a given outcome is a pass of fail
- The criteria needed for a test pass or fail definition
  - Accuracy needs to be quantified (there is no “sorta pass-ish”)
  - Removes any element of subjectivity so that any two people using the same criteria will make the same decision
  - Needs to quantify precision to establish acceptable margins of error
- The quality control function is to establish these criteria for all testing activities
  - This includes functional and non-functional requirements testing

What is Quality?

- Quality attributes
  - Those properties of a product which will be used by some stakeholder to assess the product's “quality”
  - Quality is always perceived by stakeholders in terms of what is important to them and is not an inherent property of the product.

I text all the time
I visit family in Europe often
My son needs a phone

Good Quality

Attributes

Unlimited Texting
Free International Roaming
Add a free family member

Cell Phone Plan

Inever text
I never travel outside the country
I have no family

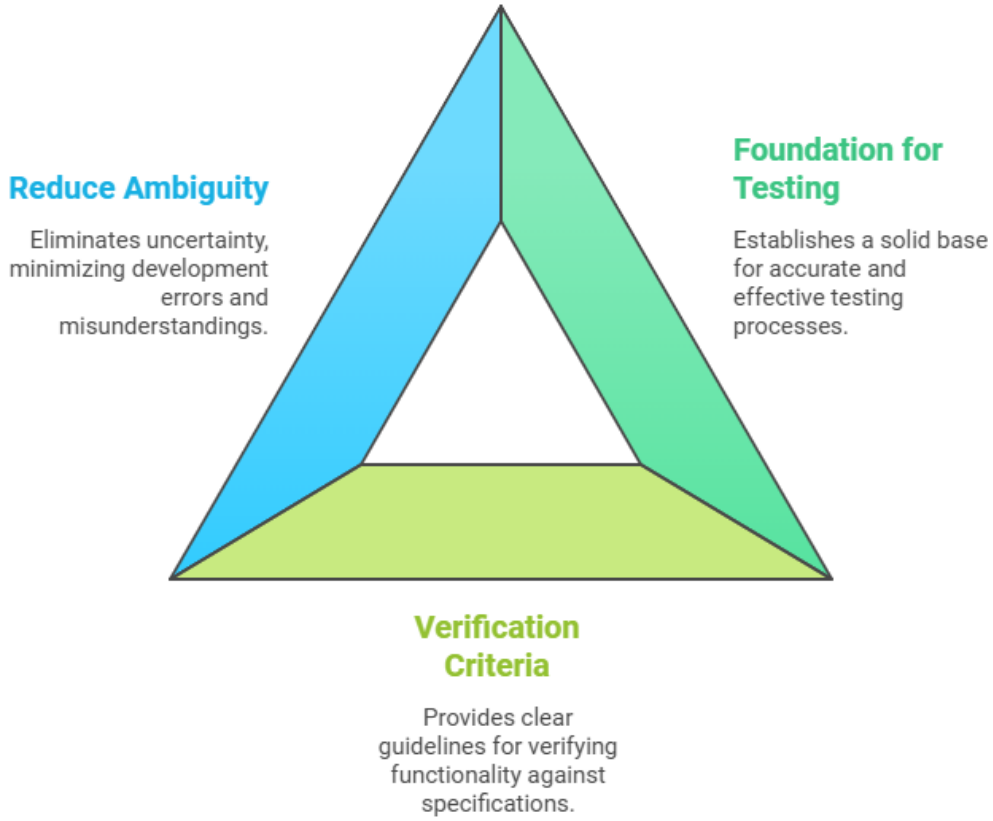
Poor Quality

The Importance of the Specification

- The specification describes how what is being build
  - Will function and perform
  - Will interact with users and other systems
  - Will interact with the organizational environment
- The specification can take many forms
  - A common part of Agile specs is a complete set of acceptance tests
- Role of the spec
  - Developers must define exactly how their code will work
  - Enables testers to develop the quality control criteria for testing

"If there is not enough information to write a test to a spec item, you don't have enough information to write code to implement it."

Good Specifications Provide:



IEEE Spec Properties

Complete	*The specfication covers all possible inputs and conditions, both valid and expected as well as invalid and unexpected.*
Consistent	*No two specification items require the system to behave differently under the same conditions, inputs and system state.*
Correct	*The described functionality for a spec item is consistent with the appropriate business logic, processes and documentation.*
Testable	*Each specification item is quantified and measurable so that a pass/fail test can be written for it.*
Verifiable	*There is a finite cost effective process for ensuring each specification item is in the final product and can be evaluated.*
Unambiguous	*There is only possible interpretation of each specification item.*
Valid	*All stakeholders can read and understand each specification item so they can formally approve the item.*
Modifiable	*The specification items are organized in a way so that they are easy to use, modify and update.*
Ranked	*The specification items are in a priority order that both the business and technical sides agree on.*
Traceable	*Every specification item can be traced back to the original requirements criteria that it is intended to satisfy.*

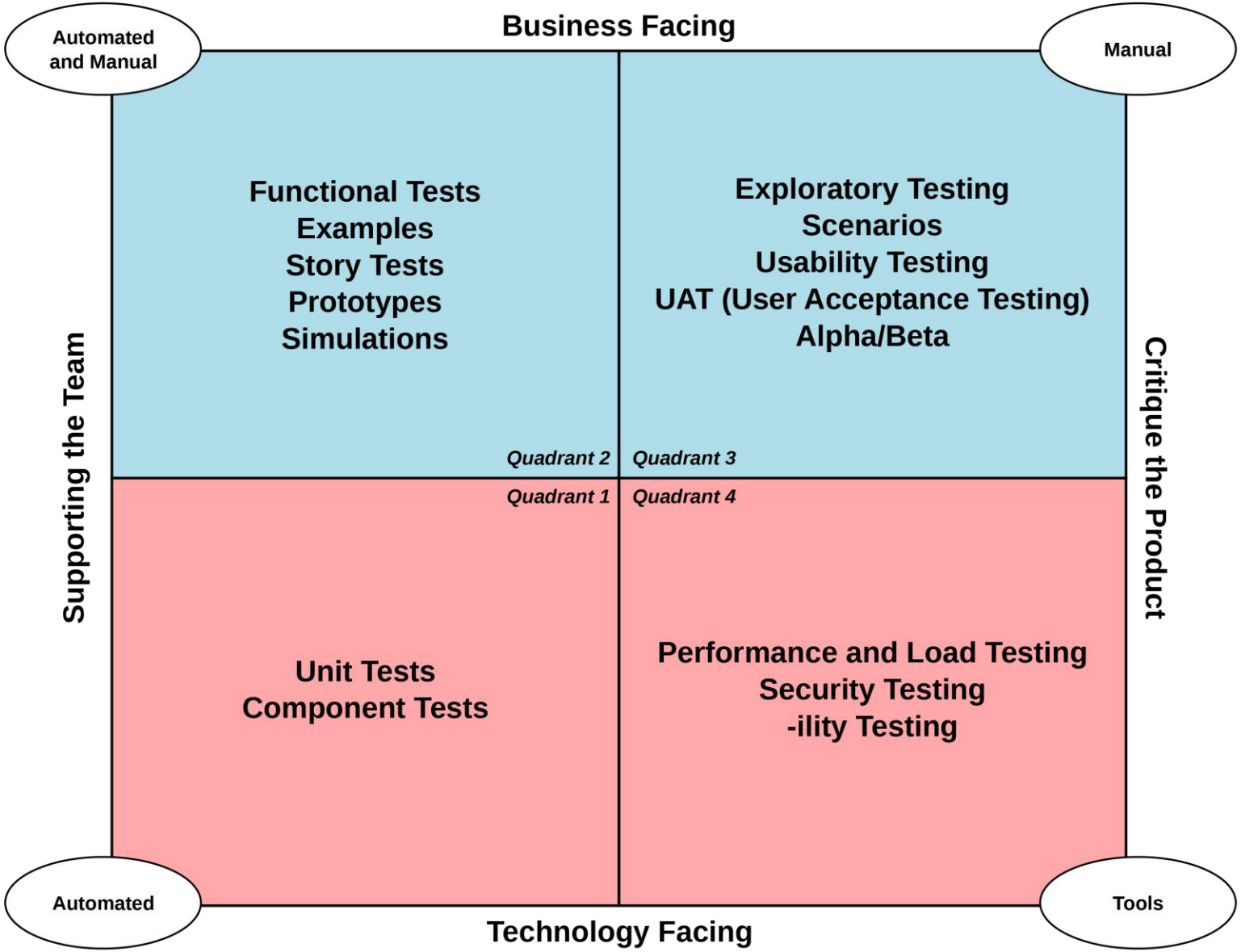
IEEE Acceptance Tests

Complete	*The acceptance tests cover all possible inputs and conditions, both valid and expected as well as invalid and unexpected.*
Consistent	*No two acceptance tests require to system to behave differently when the state and input of two tests are the same.*
Correct	*The expected result of each test meets the acceptance criteria.*
Testable	*All test inputs, states and outputs are quantified and measurable.*
Verifiable	*There is a finite cost effective process for executing each test.*
Unambiguous	*There is only possible way to interpret or understand each test and test result.*
Valid	*Everyone can read, understand, and analyze the tests well enough to formally approve the tests.*
Modifiable	*The test cases are organized in a way so that they are easy to use, modify and update.*
Ranked	*The test cases are in a priority order that everyone on both the business and technical sides agree on.*
Traceable	*Every test case can be traced back to the example and acceptance criteria that motivated it and then back to the original requirement.*

Agile Testing Principles

- **Continuous Feedback:** Agile testing emphasizes frequent feedback loops, where test results will inform the development process and guide improvements.
- **Collaborative Approach:** Testers work closely with developers and product owners.
- **Iterative Testing**: In an agile process, testing is done on every sprint or sprint like cycle. This ensures all new features are being tested and that existing features are not breaking.

Agile Testing Quadrants



Automation

- Automation is critical to being able to manage the testing activities, especially in quadrants 1 and 4 and to a lesser extent in quadrant 2
- The challenge is to ensure the automated tests themselves do not introduce errors
- The test automation code needs to be tested usually against a sample set of test data and cases that should always pass – test failures indicate errors in the test automation code
- Allows any team member to run tests in three of the quadrants
- Avoids systemic errors by removing opportunity for human errors

Automation Pitfalls

- Test code errors
  - Errors in the code that runs the tests may give invalid test results
- Unstable Tests:
  - Automating tests too early may require rework if the tests are not stable the requirement is still in flux
- Interface Dependencies
  - Changes to the interface the test is running through may affect the results
- Complacency: "Out of sight, out of mind"
  - Tendency to forget to critically review the automated tests to ensure that are still relevant and correct as the requirements change or new ones are added

Automation Pitfalls

- Configuration Issues:
  - Ensure the test automation does the correct set up before running the tests, and setup requirements may need to be modified as the system evolves
- Test Metrics:
  - Since automated tests can be run without cost, we need to avoid metrics that depend on measures like the number of times tests are run
- "That’s odd..":
  - Automated tests do not have a tester’s intuition
  - Automated tests that are not accompanied by exploratory testing can miss critical issues

Benefits of Automation

Increased Confidence

Assurance that the system functions correctly.

Repeatable Tests

Tests can be run multiple times with minimal effort.

Early Bug Detection

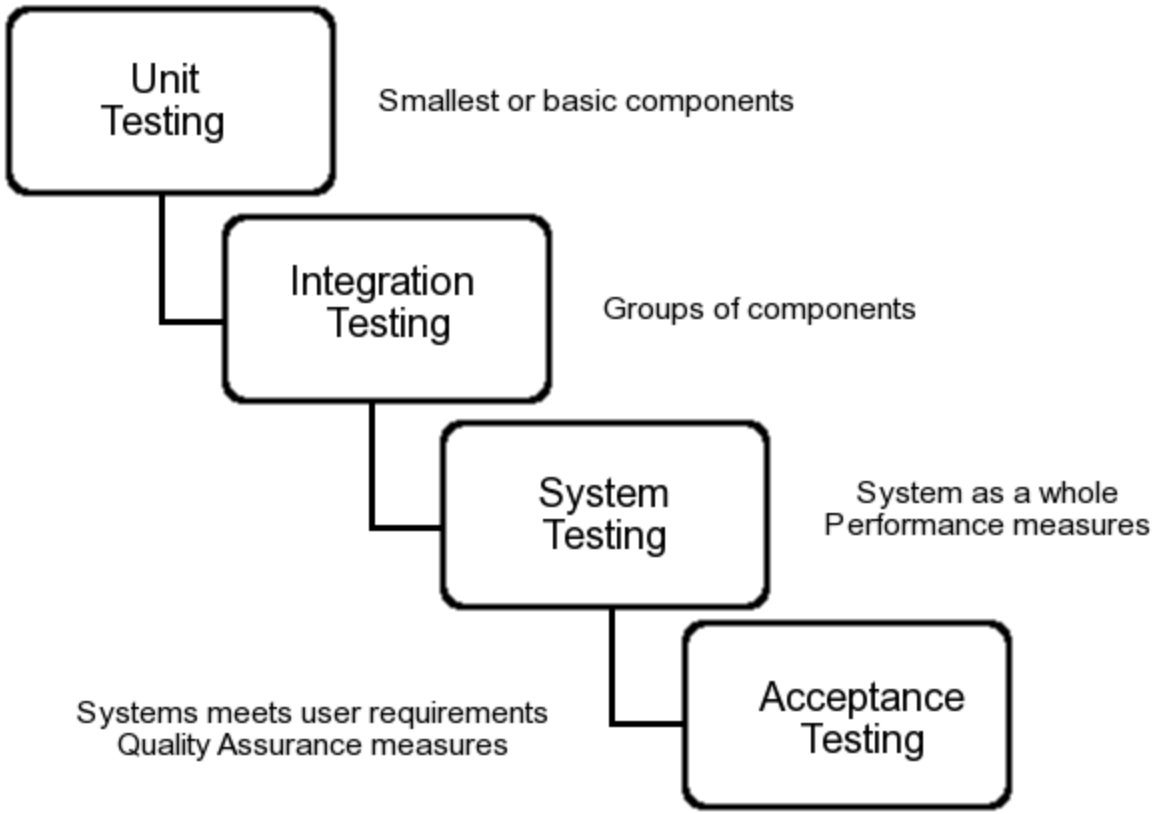
Bugs are identified early in the development cycle.

Increased Efficiency

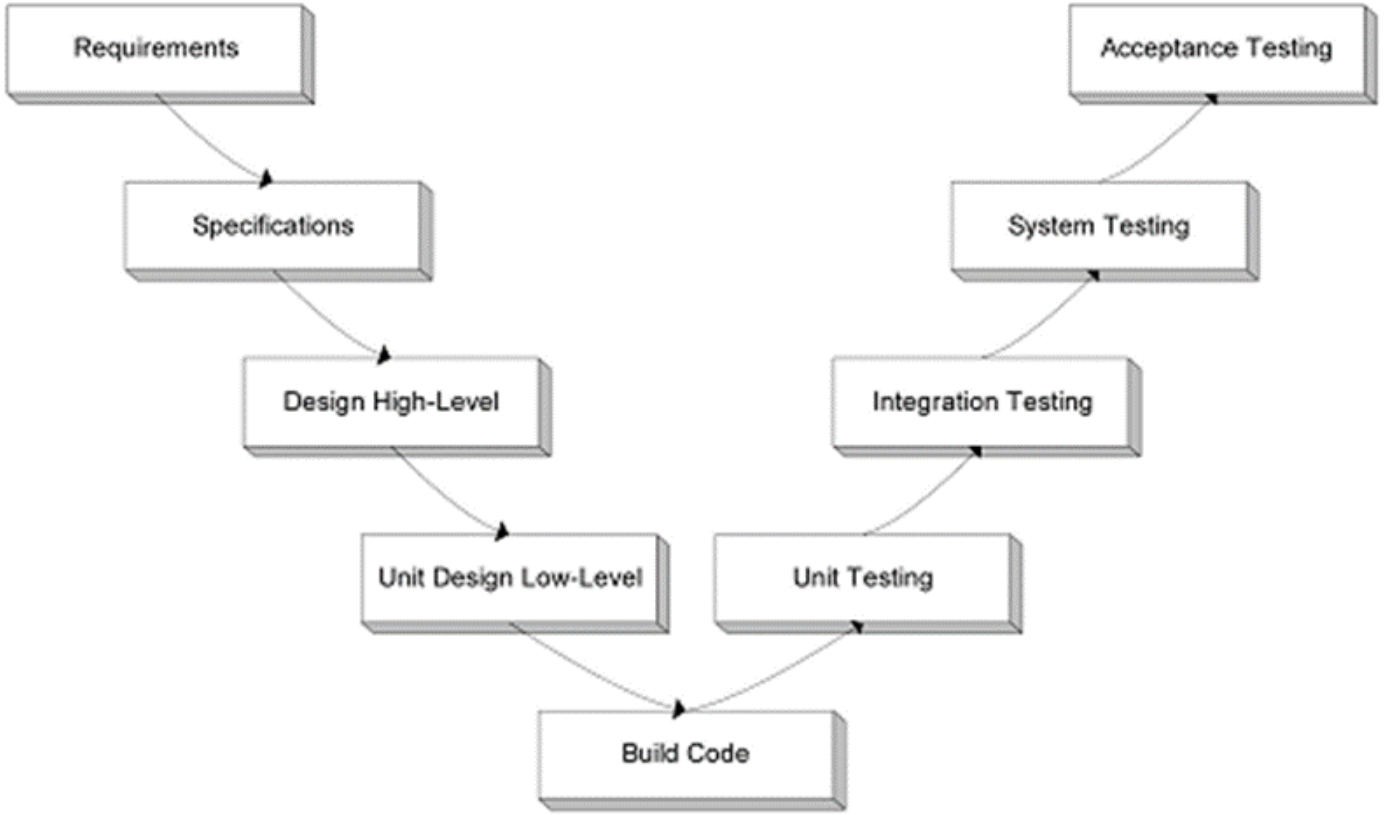
Automation speeds up the testing process significantly.

Automation is key to efficiency and confidence. It speeds up the testing process and can be used in a variety of contexts

Levels of Testing



The “V” Model



Levels of Testing

- Unit testing
  - Testing the functionality of a single component of system in isolation
  - Uses a testing framework or mock to emulate other components
- Integration testing
  - Testing the functionality of a build or how a collection of components work together
  - The components used in an integration test build have all been previously unit tested
- System testing
  - Tests the non-functional aspects of system
  - Done after the system functionality is fully tested because changes in functionality might change performance characteristics

Levels of Testing

- Scenario or End to End testing (e2e)
  - Runs the scenarios or use cases or examples against the full system
  - Ensures the application logic is correctly supported
- Near Production testing
  - E2E testing in an environment that simulates the production development
  - A more intense and expensive form of e2e
- Acceptance testing
  - Ensuring the system meets the end user requirements and quality standards
  - Beta testing, etc.

Level of Testing: Unit Testing

- **Purpose:** The main goal of unit tests are to verify the logic of a single unit of code.
- **Isolation:** Unit tests should isolate and test code separately from other dependent code.
- **Scope:** These tests are very focused on verifying the internal workings of a single component.
- **Tools**: Many frameworks are available for use when implementing unit tests, such as JUnit, pytest, and Mocha.

Level of Testing: Integration Testing

- **Purpose:** Integration testing ensures that the system components interact and function as intended.
- **Scope:** Integration tests test a system at the level of how the different modules interact with each other, as opposed to the smaller units of code found in unit tests.
- **Build Verification:** They help to catch integration related errors as you combine different components.

Level of Testing: System Testing

- **Purpose:** System tests are designed to test the whole system end-to-end and verify non-functional requirements.
- **Performance**: System testing will include performance tests, load tests, and stress tests.
- **Security:** This can also include security related tests and verifying that the system can prevent unauthorized access.
- **User Expectations**: System tests determine if the system meets user expectations and can handle normal and unusual situations.

Level of Testing: Acceptance Testing

- **End-User Validation:** Acceptance tests confirm that the system meets the end user requirements and also verifies the overall quality of the user experience.
- **Beta and UAT Testing**: This typically involves testing with real end users as with beta testing or user acceptance testing (UAT).
- **User Goals:** Acceptance tests will make sure that all user workflows have been implemented correctly.

Develop a passion for learning

Testing and Quality – TDD

Test Driven Development

- Developed as Kent Beck as part of the XP Agile methodology
- Write tests before writing the code
  - Forces developers to think through a solution before writing code
  - Identifies where developers do not enough information
  - Once the code works, then developers can then refactor the code
- As code is written, the tests are run
  - Any errors are caught and corrected
  - Code is debugged as it is written

Empirical Evidence

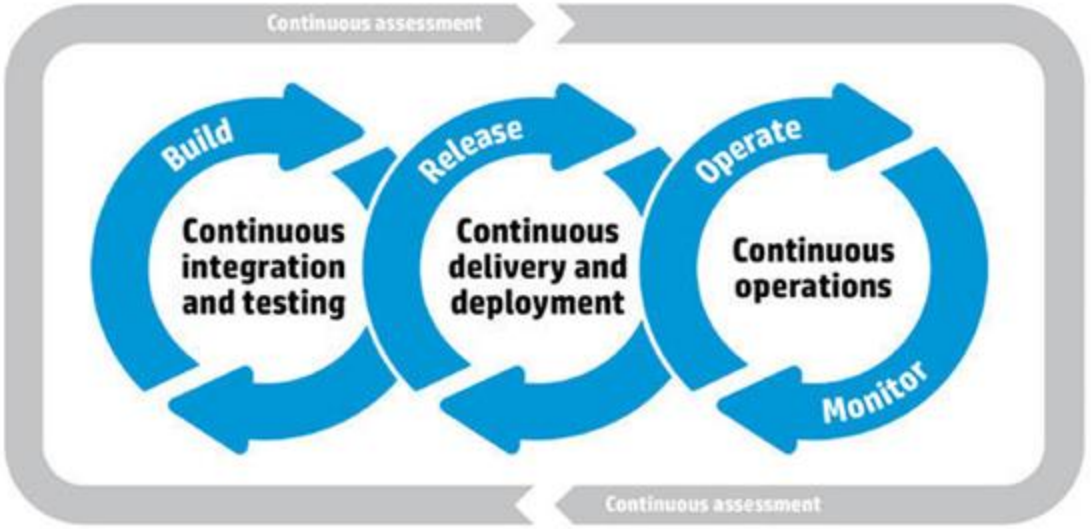
- On average, code produced with TDD:
  - Has a lower defect density
  - Has better coverage of possible cases
  - Shows mixed results as to increases in programmer productivity and decreased development time
  - Shows initial increase in development time but off-set by “down the road” savings in development and application support
  - Is more loosely coupled
  - Shows dramatic improvements in programmer productivity in many cases, some show no change, and a few show a decrease
  - Has higher client satisfaction measures on average

The Critical Success Factor

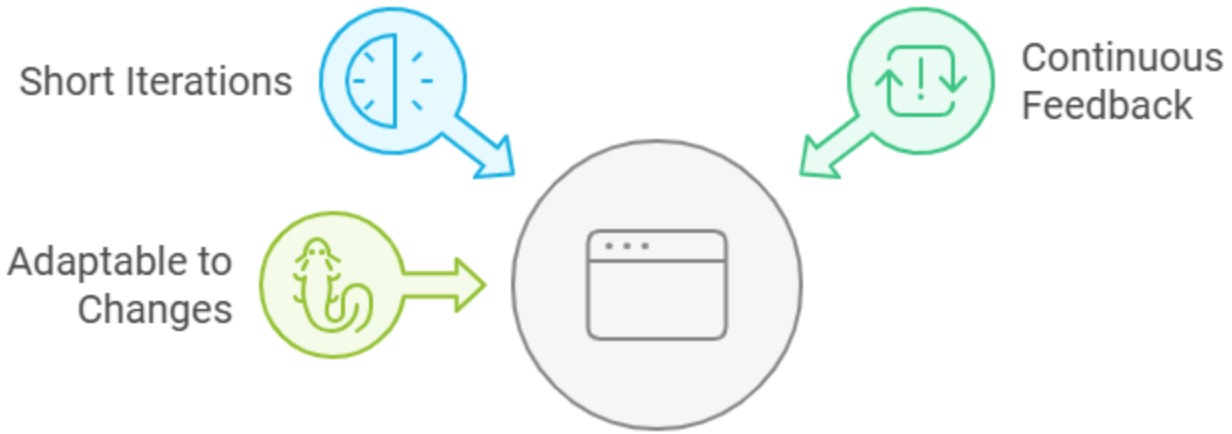
- One factor that explains the mixed results is how rigorously TDD is followed
- Best results come from engineering and industrial environments where TDD is applied rigorously and consistently
  - Engineering culture is focused on quality of code, risk and productivity
- Neutral or negatives results tend to come from academic or non-industrial contexts where the culture is not usually highly quality oriented
  - Non-industrial contexts are more like to practice lazy TDD

DevOps and TDD

- DevOps, for example, is built around the goal of continuous delivery of software
  - In order to meet the requirement for continuous testing as part of continuous delivery, software must be tested as it is under development
  - DevOps relies on TDD to meet this requirement



Agile Testing and TDD



- **Short Iterations:** Agile teams work with short iterations, and the implementation of tests is equally incremental.
- **Continuous Feedback**: The TDD approach ensures that the team gets continuous feedback. If tests fail, the feedback is immediate and problems can be addressed immediately.
- **Adaptable to Changes:** With iterative changes, test can be modified alongside the changes to requirements and design.

Why Agile Teams should embrace automation

- **Scalable Testing:** Automation allows for more test coverage in less time.
- **Faster Feedback:** Faster test execution means faster feedback on code changes.
- **Faster Release Cycle:** Ultimately, test automation speeds up the release cycle so that teams can be more agile in releasing features more rapidly.

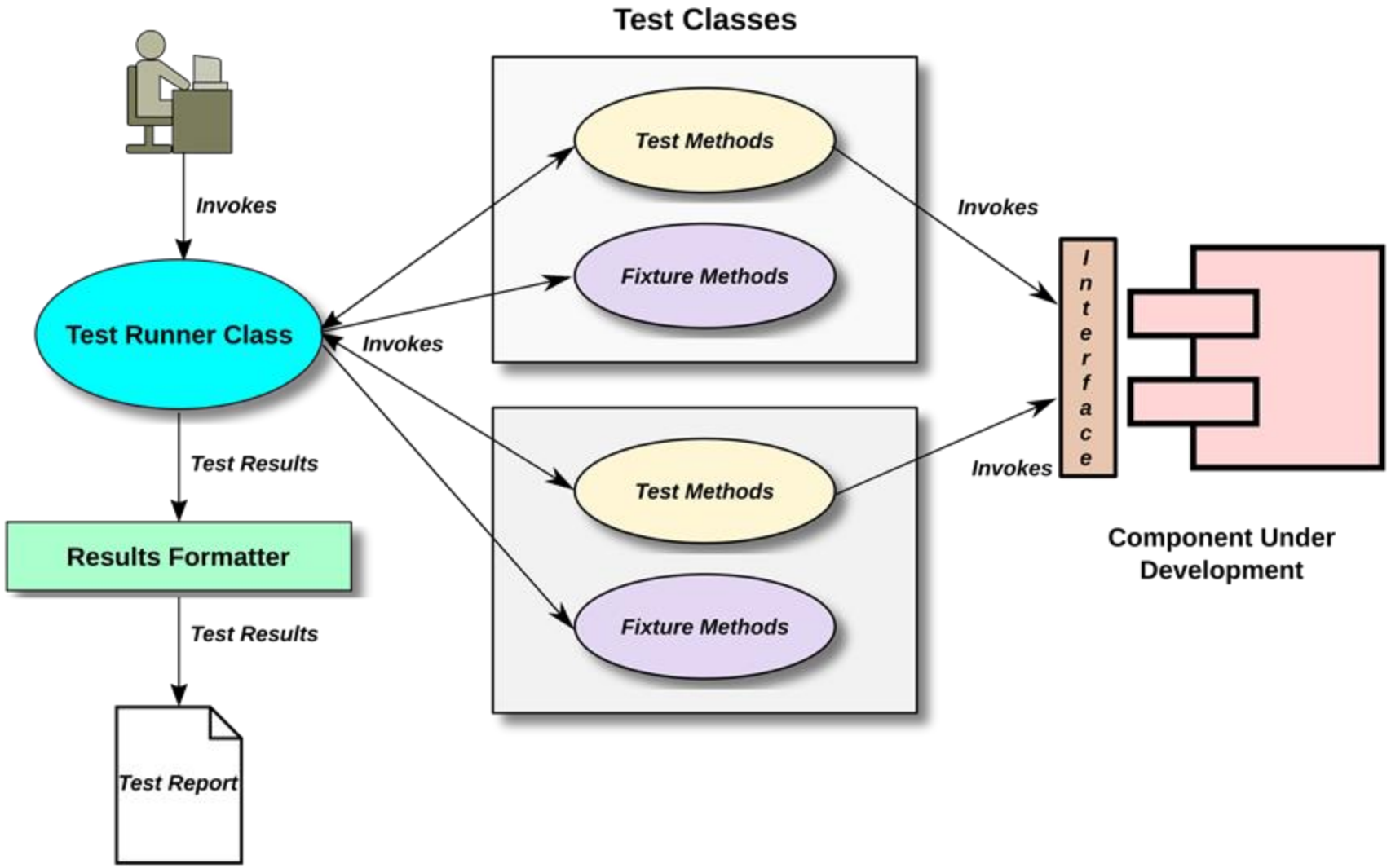
Programmer Comments on TDD

- “I’m writing code faster with less rework and less profanity”
- “It’s like the code is writing itself, I’m not doodling in code any more trying to figure out why I wrote that piece of code a month ago”
- “My code is simpler, cleaner and easier to understand”
- “Amount of rework is reduced since bugs are caught early, which means I go home on time”
- “Continuous regression testing means that adding new code doesn’t break my existing code”
- “Programming is fun again. I don’t spend all my time on bug hunts, and I get to think instead about optimizing my design and implementation”

Why Does TDD Improve Programming?

- Writing tests first
  - Forces programmers to immerse themselves in the problem before writing any code
  - Forces a critical analysis as to whether the developers know what the software is supposed to do
  - “If you can’t figure out what the right result of a test should be, then you don’t know what your code should do in that situation”
  - Requires a global view of the problem to be solved before crafting a solution
  - Is consistent with the cognitive mechanisms used to solve problems

Automation JUnit



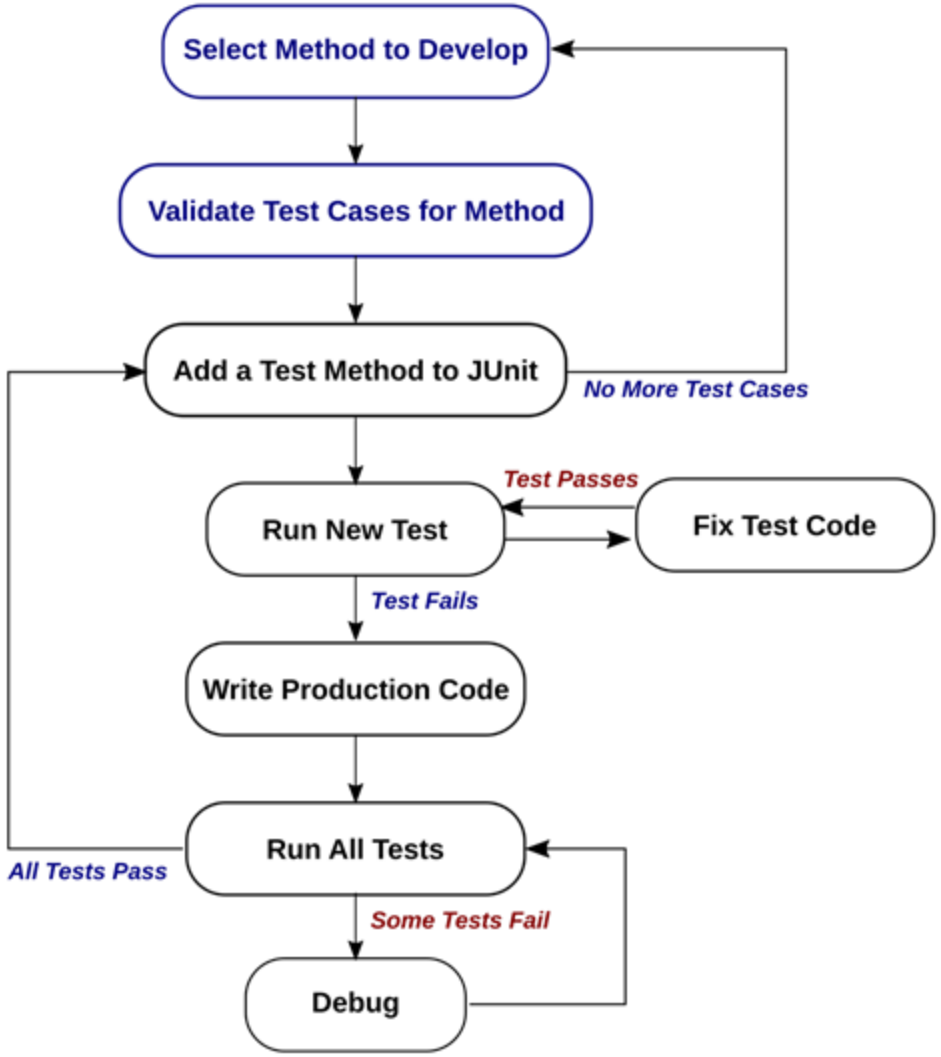
Adding Test Code

Calculator.java

```
1
2  public interface Calculator {
3      public int add(int x, int y);
4      public int subtract(int x, int y);
5      public int multiply(int x, int y);
6      public int divide(int x, int y);
7
8  }
9
```

```
11
12     @Test
13     public void AddTest1() {
14         // adding 3 + 7 expect 10
15          Calculator c = new CalcImp();
16          assertEquals("Test failed",c.add(7, 3),10);
17          // or we could write either of the following
18          assertTrue("Test failed",c.add(3,7)==10);
19          assertFalse("Test failed",c.add(3,7)!=10);
20     }
21
```

The TDD Workflow



Refactoring

Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small behavior-preserving transformations, each of which "too small to be worth doing". However, the cumulative effect of each of these transformations is quite significant.

By doing them in small steps you reduce the risk of introducing errors. You also avoid having the system broken while you are carrying out the restructuring - which allows you to gradually refactor a system over an extended period of time.

Martin Fowler

Refactoring

- Refactoring
  - Changes the structure of existing code
  - Does not change the functionality of the code
  - Uses TDD to ensure code changes do not cause bugs
- Refactoring is usually done:
  - After the functionality of the code changes
  - As part of a design change
  - After a bug has been fixed
  - After a code review

The Red-Green-Refactor Cycle

- **Red (Test Fail):** Start by creating a test and make sure the test fails, because you have not implemented the functionality yet.
- **Green (Test Pass):** Now implement the functionality and make the test pass. The test must only have the bare minimum of code required to get to the passing state.
- **Refactor:** Once the test is passing, then you should refactor the code to eliminate duplication or other code smells, and improve the overall design of the code.

Code Refactoring Strategies

- **Start Small:** Break down large code changes into smaller, manageable parts.
- **Test before changing:** Make sure to create a comprehensive test suite before making any code changes.
- **Continuous Integration:** Try and integrate all your changes frequently to catch any issues.
- **Review your code**: Have other developers review your code.

Well-Structured Code

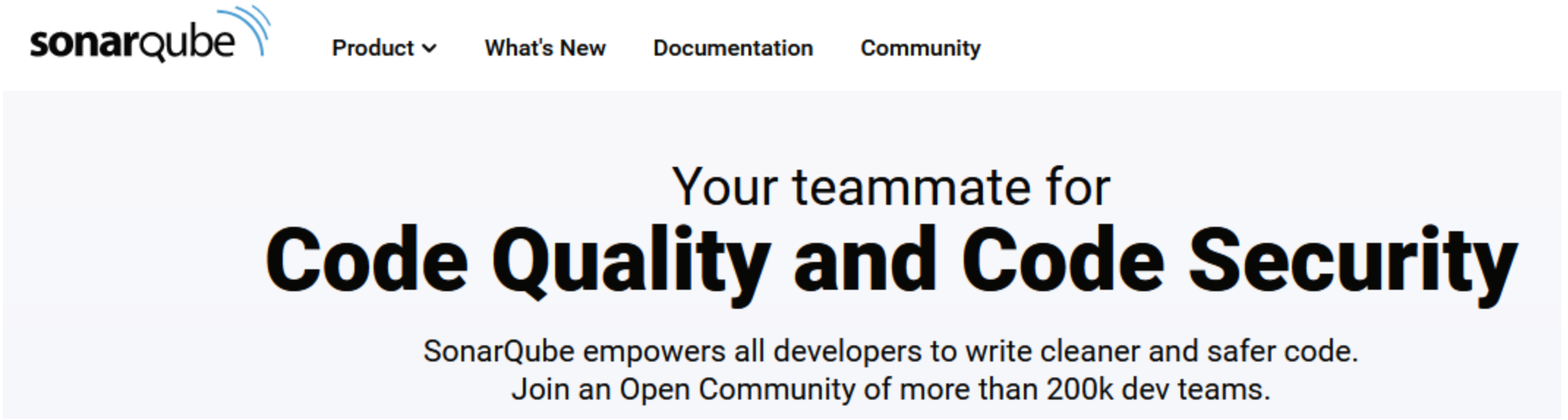
- Code that is well designed and well structured:
  - Is easier for programmers to understand
  - Is easier to modify
  - Is easier to maintain
  - Is more elegant and efficient
- When changes to code take place:
  - The structure of the code degrades
  - The degradation to the code is cumulative
  - Eventually the code devolves into an unstructured mess

Code Smell

- A code smell is:
  - "... a surface indication that usually corresponds to a deeper problem in the system"
  - Where the code does not support good design practices
  - An indication a refactoring needs to be done
- This is closely related to the idea of an anti-pattern
  - Anti-patterns are recurring patterns of how developers go from a problem statement to a bad solution
  - Refactorings also exist for anti-patterns

Code Quality

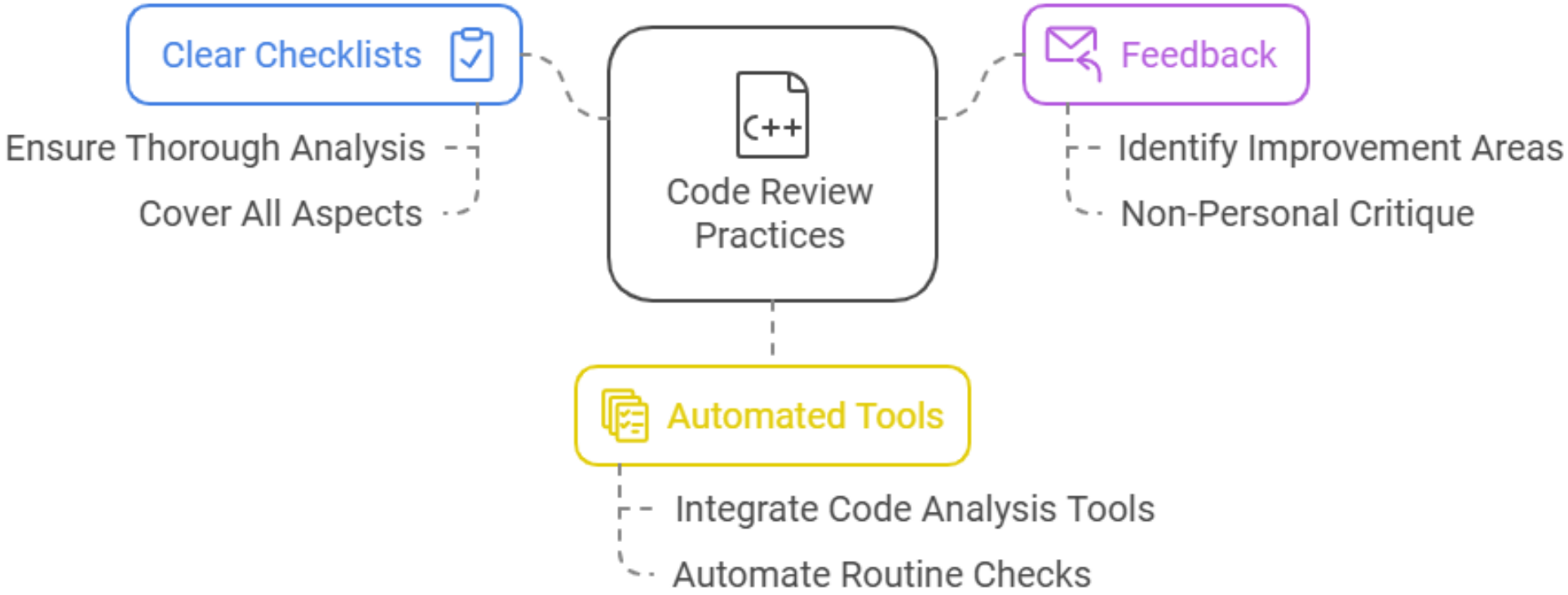
- Measures the elegance of the code
  - Often called software craftsmanship
  - Or Clean Code
  - General coding best practices plus language specific best practices
- Code smells are usually indicators of poorly written code
- Most effective way to clean up code is:
  - A code review
  - Pair programming – reviewing code as it is being written
- Modern code analysis tools can also measure code quality
  - As well as doing security analysis


sonarqube
Product
What's New
Documentation
Community
Your teammate for
Code Quality and Code Security
SonarQube empowers all developers to write cleaner and safer code.
Join an Open Community of more than 200k dev teams.

Importance of Code Analysis Tools

- **Automated Analysis:** Code Analysis tools can perform static analysis of your code to verify that it matches standards.
- **Code Security:** These tools can also identify security vulnerabilities in your codebase.
- **Improve Code Quality**: By using code analysis tools, you can improve the readability, maintainability and the overall quality of the code.

Code Review Practices



Lab Exercise 1

In this lab, we'll explore advanced unit testing using Jest.

We'll cover more complex scenarios, edge cases, test different parts of a class, and also use nyc to measure code coverage.

After creating your tests, then you will refactor your code to make it simpler, more modular, and better designed, and finally rerun the tests.

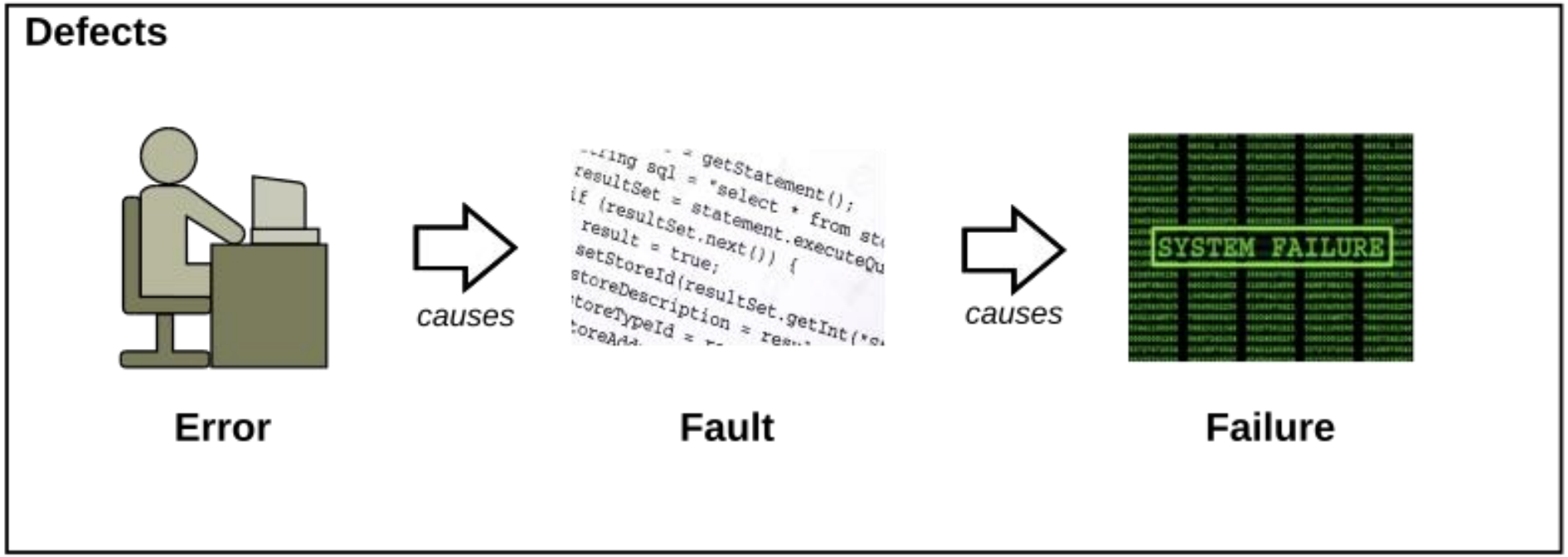
This process will simulate the Test Driven Development workflow and also highlight the importance of code refactoring.

INSTRUCTIONS: Day4 Labs/Lab1.md

Testing and Quality – ATTD/BDD

Terminology

Root Causes of Defects



Error: a human action that eventually leads to a fault

Fault: an incorrect step in building the system at any point that results in failure

Failure: any place the software does not perform as required

Defect: a generic term for any of the above

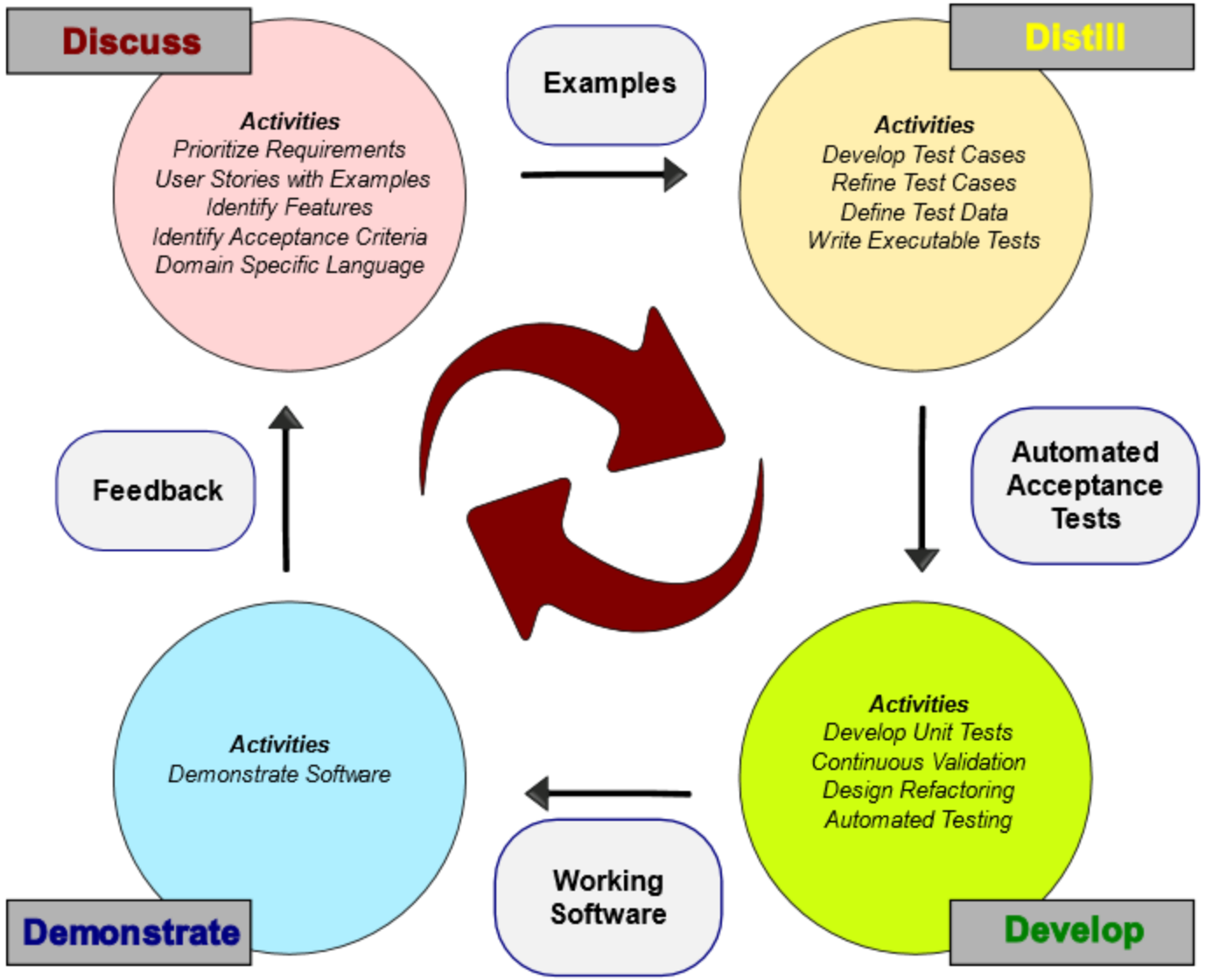
Some Typical Errors

- Communication errors – ambiguity, misunderstanding etc.
- Misunderstood requirements – wrong solutions to the problems
- Missing cases – things that 'fell through the cracks'
- Team members not coordinated – not on the same page
- Disconnect between testing, development and product owner on exactly what should be done and what has been done
- Meandering design because the result is not clearly defined
- Lack of verification process to identify problems

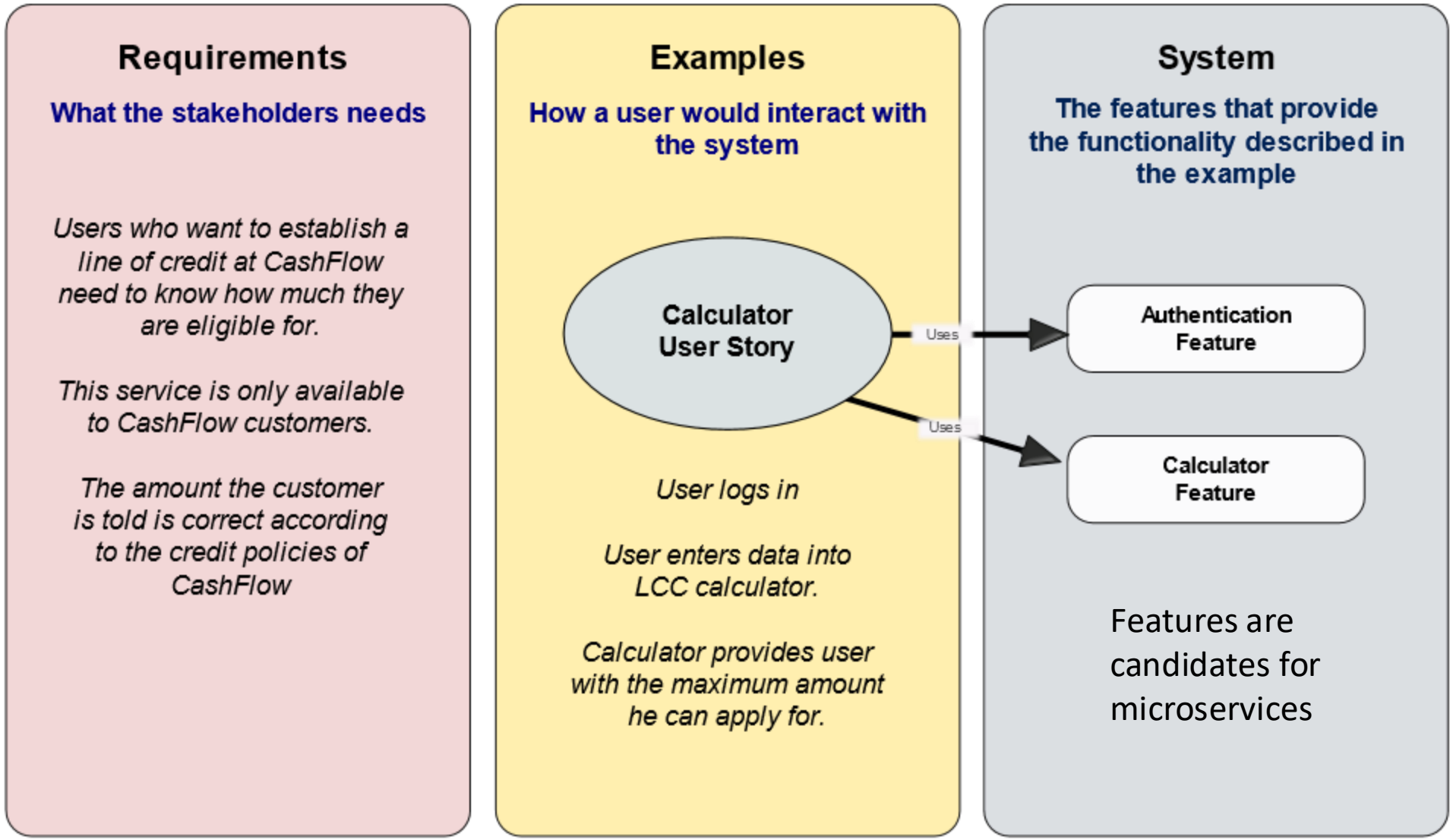
Behavior Driven Development (BDD)

- **Behavior over Implementation**: BDD is a methodology that focuses on the behavior of the system as opposed to the actual implementation of a system.
- **User-Centric Testing:** BDD tests scenarios and use cases through a user perspective.
- **Communication**: Using a domain specific language, the behavior of the system can be described in a human readable way, making it easier for the development team to communicate with business stakeholders.

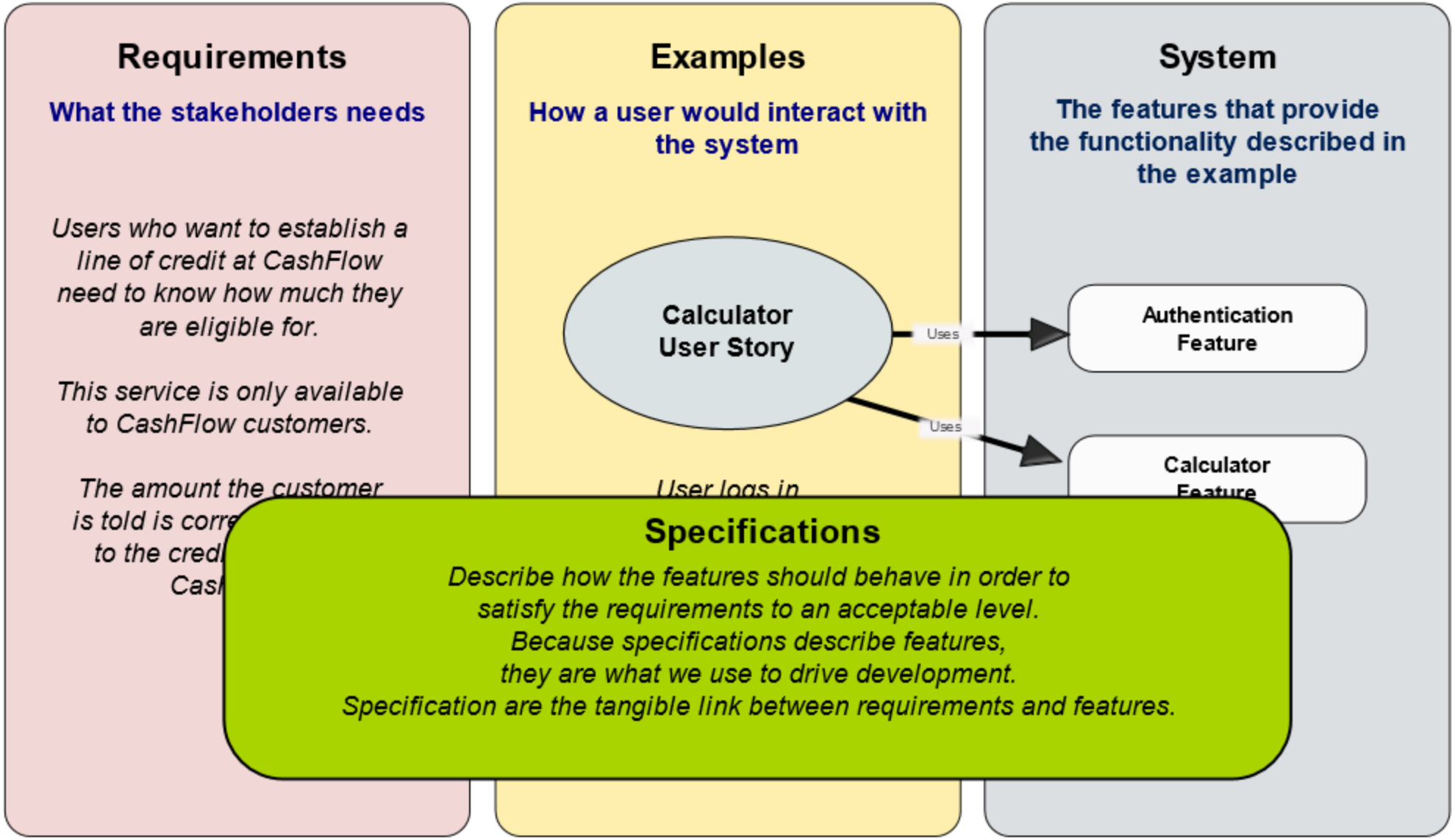
ATTD Process



Feature Discovery



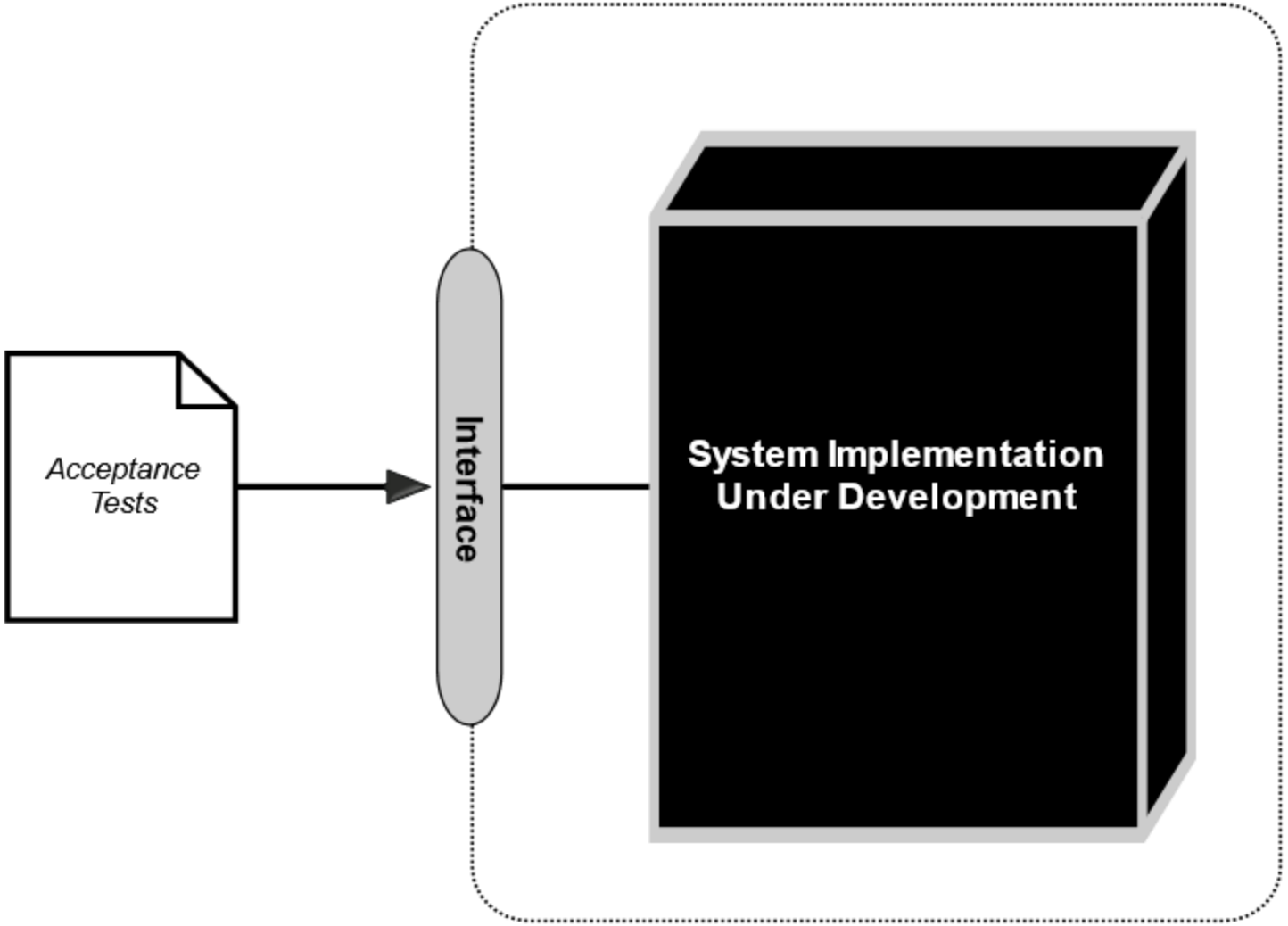
Exploring Behavior



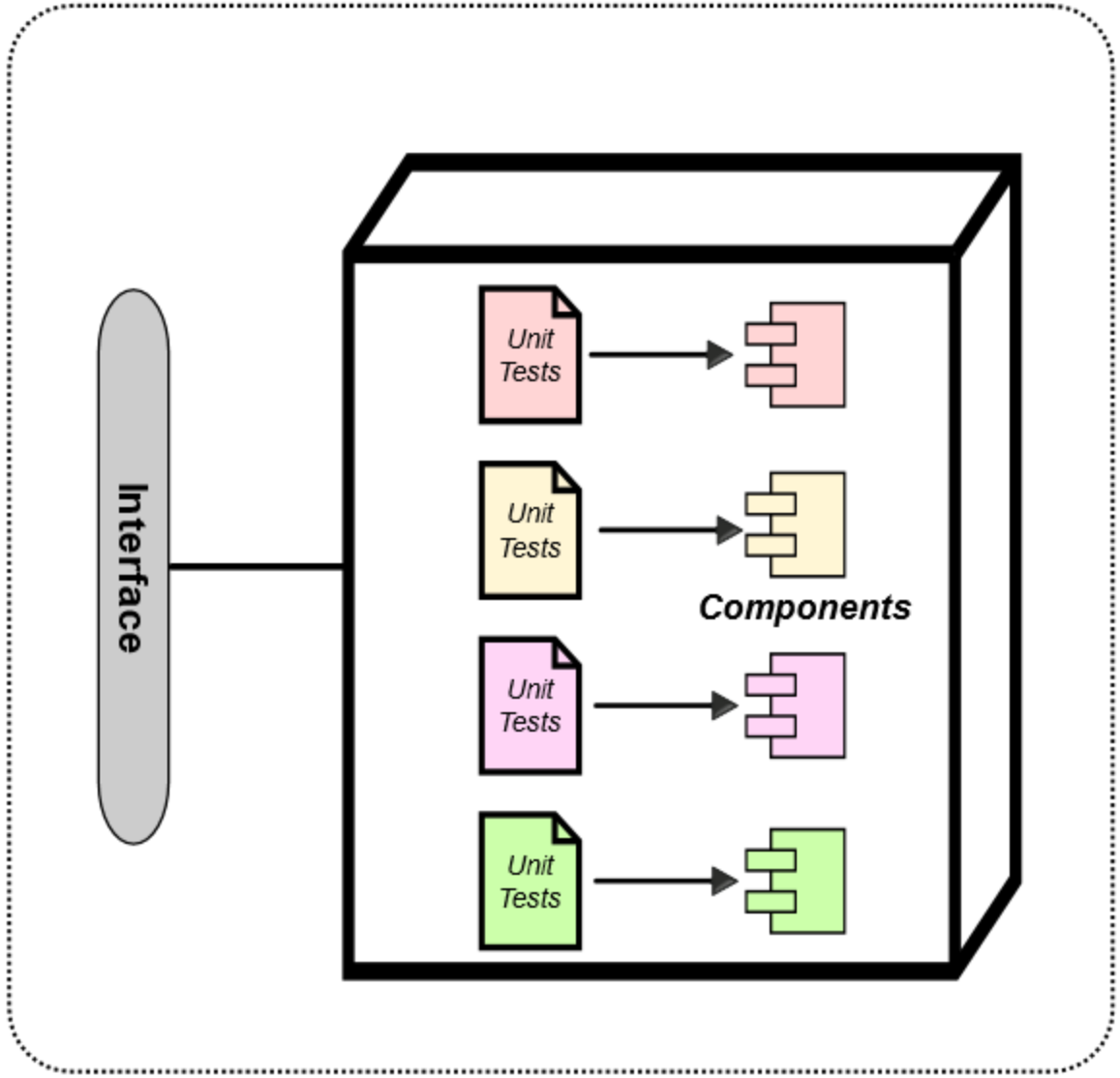
ATTD/BDD – Acceptance Testing

- **Automated Execution of Acceptance Tests**: You can use automated tools like Cucumber or SpecFlow that execute acceptance test criteria.
- **Domain Specific Language**: These tools support domain specific languages like Gherkin which are used to describe acceptance criteria.
- **Business Readable Tests**: The tests written in these languages can also be read and understood by stakeholders that are non-technical.

ATTD/BDD – Integration Testing



TDD – Tests Individual Components



Gherkin

- Language for writing automated tests
- Understood by Cucumber and other BDD tools
- Business Readable, Domain Specific Language
  - It has a simple formal syntax
  - Each Gherkin feature file contains acceptance tests for a single feature
  - A feature may be associated with more than one Gherkin feature file

Gherkin Example

file lcc.feature

```
Feature: Line of Credit Calculator

  Background:
    Given that user has been authenticated
    And the lcc calculator is available

  Scenario: Valid Edge Case
    When the customer enters 50000 for income
    And enters 180 for days of employment
    And enters 750 for the credit rating
    Then the the maximum line of credit is 5000

  Scenario: Income too low
    When the customer enters 49999 for income
    And enters 180 for days of employment
    And enters 750 for the credit rating
    Then the the maximum line of credit is 0

  Scenario: Employment too low
    When the customer enters 50000 for income
    And enters 179 for days of employment
    And enters 750 for the credit rating
    Then the the maximum line of credit is 0
```

Feature: Section

Background: Section

Scenario: Section

Scenario: Section

Scenario: Section

Cucumber Output

```
Feature: Login Feature
  As an authenticated
  I want to login to application
  So that I can access application feature
Open browser
Starting ChromeDriver 79.0.3945.36 (3582db32b33893869b8c1339e8f4d9ed1816f143-refs/branch-heads/3945@{#614}) on p
Only local connections are allowed.
Please protect ports used by ChromeDriver and related test frameworks to prevent access by malicious code.
[1582969229.146][WARNING]: This version of ChromeDriver has not been tested with Chrome version 80.
Feb 29, 2020 3:10:29 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C
Close browser
Open browser
Starting ChromeDriver 79.0.3945.36 (3582db32b33893869b8c1339e8f4d9ed1816f143-refs/branch-heads/3945@{#614}) on p
Only local connections are allowed.
Please protect ports used by ChromeDriver and related test frameworks to prevent access by malicious code.
[1582969239.876][WARNING]: This version of ChromeDriver has not been tested with Chrome version 80.
Feb 29, 2020 3:10:40 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C

  Scenario: Check valid login                 # /Users/mansishah/Selenium_automation/Automation/blog/src/test/java
    Given user is on Login Page               # LoginSteps.user_is_on_Login_Page()
    When user has provided valid credentials  # LoginSteps.user_has_provided_valid_credentials()
    Then user should be able to login         # LoginSteps.user_should_be_able_to_login()
Item searched successfully
Close browser

  Scenario: SearchProduct                     # /Users/mansishah/Selenium_automation/Automation/blog/src/test/java
    Given user is on Login Page               # LoginSteps.user_is_on_Login_Page()
    When user has provided valid credentials  # LoginSteps.user_has_provided_valid_credentials()
    Then user search for the product          # LoginSteps.user_search_for_the_product()

2 Scenarios (2 passed)
6 Steps (6 passed)
0m25.389s
```

Lab Exercise 2

- **Focus:** In this lab, we'll move beyond unit testing and focus on integration tests. You will be creating tests for a REST API.
- **Real-World Interaction:** Test the interaction of a basic REST API with a client (using test code).
- **Edge Cases:** Explore edge cases to see how an application behaves under various conditions and error scenarios.
- **Tools:** Using supertest and jest, this lab will solidify integration tests

INSTRUCTIONS: Day4 Labs/Lab2.md

E2E Testing

Scenario Based Testing

- Basic unit of testing is the use case scenario, or user story example
- User story:

  A customer goes to the CashFlow website. At the login page, they enter their user name and password. Once they are authenticated, then they go to the line of credit calculator, provide the required information and press the submit button. The amount they are eligible for is displayed.
- Example:

  A customer goes to the CashFlow website. At the login page, they enter "BOBK" as their user name and "KittyKat" as their password. Once they are authenticated, then they go to the line of credit calculator, enter $40,000 for income, 759 for credit rating, and 7 years for employment, and press the submit button. The amount they are eligible for is displayed as $6,898.

Scenario Testing

- Integration testing ensures the acceptance tests pass
- Scenario testing ensures that the functionality delivers the correct result from the business or user perspective
- Automated acceptance testing only tests that the various features work correctly
  - Scenario testing ensures they produce the correct overall result
- E2E testing fully exercises the system
  - It also shows what parts of the system are not being exercised

Near Production Testing

- Scenario testing but in a production environment
- Much more expensive
  - Production environments are highly complex
- Shows up problems that might not be observable in a test environment
- Easier to do with cloud-based systems
  - Full copies of a production environment can be spun up, used for testing, then spun down

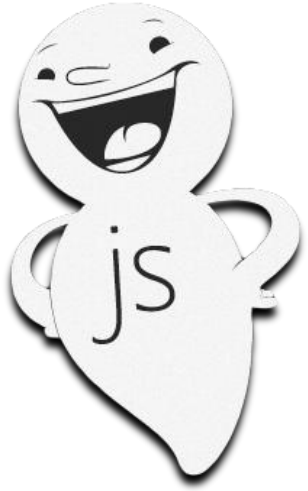
Benefits of End-to-End Testing

- **Holistic Verification:** End-to-end testing ensures that all components of the system function together to meet the business use cases.
- **Detect Integration Issues:** These tests can catch integration issues between different parts of the system.
- **Real-World Scenarios:** E2E testing simulates real user workflows, ensuring the functionality is correctly implemented from start to finish.

Challenges of End-to-End Testing

- **Time Consuming**: Compared to unit or integration tests, end-to-end tests are often more difficult to set up and maintain. The scope and complexity will also lead to longer execution times.
- **Brittle**: Because E2E tests will touch all aspects of the application, it makes them more susceptible to breaking whenever changes are made to the underlying code.
- **High setup cost:** End-to-End testing is much more expensive than other types of testing as it involves setting up an environment that closely mirrors production.

Some e2e Tools



Casper

Protractor

end to end testing for Angular

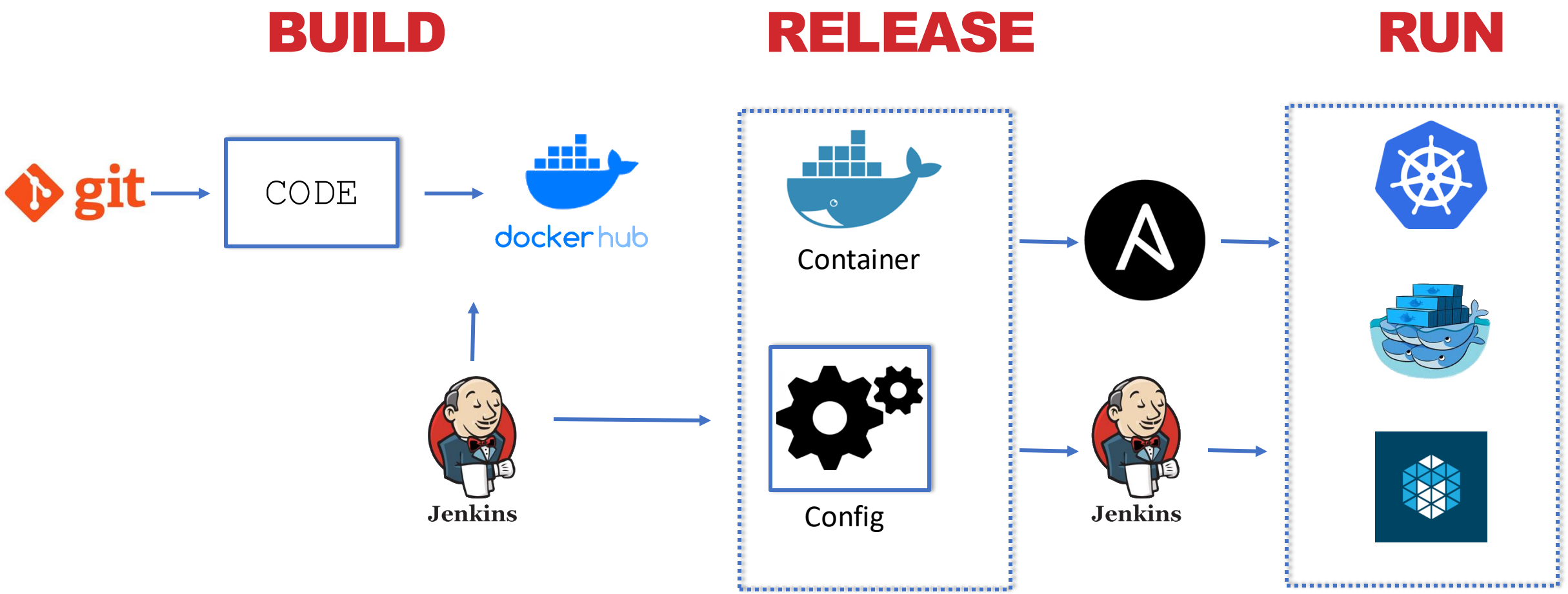
CodeceptJS

Nightwatch.js

End-to-end testing, the easy way.

TestCafé

BUILD PIPELINES: Ci/CD



BUILD PIPELINES: Choosing Environment

Jenkins

Travis CI

TeamCity

Circle CI

CodeShip

GitLab

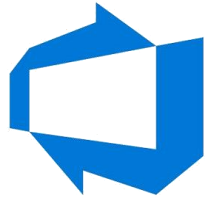
Buddy

Semaphore

AppVeyor

Google Cloud Build

AWS CodePipeline



Azure DevOps

81

BUILD PIPELINES: Choosing Environment

Product	Comment
Jenkins	Jenkins is the number one open-source project for CI/CD automation. Community publishes thousands of plugins to facilitate easier implementation of common tasks.
Travis CI	Used by companies such as Facebook, Mozilla, Twitter, Heroku,
TeamCity	TeamCity takes advantage of cloud computing by dynamically scaling out its build agents farm on Amazon EC2, Microsoft Azure, and VMware vSphere
Circle CI	CircleCI uses language-specific tools like rvm and virtualenv to ensure dependencies are installed into an isolated environment
CodeShip	Codeship is a hosted continuous integration platform that's focused on efficiency, simplicity, and speed. Teams can use Codeship to test, build, and deploy directly from projects on GitHub.
GitLab CI	GitLab CI provides tools for issue management, code views, continuous integration and deployment, all within a single dashboard.
Buddy	Custom support for Grunt. Gulp, MongoDB, and MySQL
Semaphore	Custom tests for dependencies, units, code style, security, and acceptance
AppVeyor	AppVeyor is a Windows-only cloud-based service for testing, building, and deploying Windows applications. SSD drives with dedicated hardware to provide fast speeds.
Google Cloud Build	Google's provider based solution for DevOps and CI/CD management
AWS CodePipeline	Amazon Web Services's provider based solution for DevOps and CI/CD management
Azure DevOps	Microsoft Azure's provider based solution for DevOps and CI/CD management

Spring Cloud

- Spring Cloud provides tools to build some of the common patterns in distributed systems
  - Focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others
- Spring is a lightweight framework for Java development
  - Address the complexity of Java EE
- Spring Boot makes developing microservices with Spring Framework faster by implementing three main features
  - Autoconfiguration
  - An opinionated approach to configuration
  - The ability to create standalone applications

Lab Exercise 3

- **Focus:** In this lab, we'll explore testing from the perspective of the user.
- **Behavior-Driven Development (BDD):** Learn to describe the expected system behavior using Gherkin language.
- **End-to-End Testing:** Implement a realistic scenario that involves multiple components.
- **Tools:** We'll use Python and the Behave framework to implement the tests.

INSTRUCTIONS: Day4 Labs/Lab3.md

Develop a passion for learning

Thank You