Software Engineering

CSDS-393

Luis Jimenez Segovia
Computer and Data Sciences Department

Spring 2023

Agenda

- Software Testing
  - Static vs. Dynamic
  - White Box vs Black Box
  - Functional Testing Levels
    - Unit Testing
    - Integration Testing
    - System Testing
      - Subsystem Testing
    - Acceptance Testing
      - Alpha and Beta Testing
  - Non-Functional Testing Types
    - Load and Performance Testing
- Software Testing Life Cycle
  - Phases
- Specification-based Testing
- Code-based Testing
- Fault-based Testing
- Interaction Testing
- Field Testing
- Regression Testing
- Test-driven development
- Synthetic Testing
- Code Inspection
- Defect Tracking
- Dynamic and Static Analysis Tools
- Statistical Reliability Estimation
- Online Controlled Experiments

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Software Testing

Software Testing

- **Software testing** is intended to **prove** that a program does what it is **intended to do** and to discover **program defects** before it is put into use.
  - We always want to detect defects before the customer does.
- When software is tested, the testing team validates a predefined set of **test cases**.
  - A **test case** is the specification of the input, conditions, testing procedure, and expected results.
  - After the test cases are **executed**, testers validate the results for any indication of errors, anomalies, or deviations from the program's functional and non-functional requirements.

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Software Testing cont.

- **Software tests reveal the presence of errors, not their absence.** <-Key!
  - The application's behavior is evaluated for **conformance to the requirements.**
- Testing is directly related to the **quality** of the software.
- **Quality issues** result in extra costs and risks.
  - Customer incidents, issues that translated to costs.
  - Internal costs required to remediate a defect.
- **Software quality is directly correlated with customer satisfaction.**
- Testing positively supports improvements in the quality of software.

Testing Terminology

- **Test:** A procedure to establish the quality, performance, or reliability of software.
- **Test case**: The specification of the input, conditions, testing procedure, and expected results.
- **Test suite:** A collection of test cases.
- **Failure**: Runtime deviation from required behavior.
- **Defect, fault, or bug**: A flaw in a program that can cause it to fail.
- **Error**: A human action that results in a defect or fault.

Testing Terminology cont.

- **Code coverage :** A software testing metric that determines the number of lines of code successfully validated under a test procedure.
- **Reliability measure**: A numerical measure of the extent to which deployed software conforms to requirements.
  - e.g., **frequency of failures.**
  - Depends on **operational (field) usage.**

Goals of Testing

- **Reliability assessment:**
  - Important for deployment and release decisions.
  - It is often done **subjectively**, based on **in-house** testing.
  - It can be calculated **statistically** based on **field (captured in the field)** testing.
- **Reliability improvement:**
  - Defect identification and repair.

When to Test

- Components should be tested as soon as they become available.
- Upon completing the component, developers have the greatest:
  - **Motivation** to test it.
  - **Focus**, **recall,** and **understanding.**
- The cost of repair is the **lowest** possible throughout the entire SDLC.
- Early testing is facilitated by **tool support and automation.**
- **DevOps and Agile principles** prioritize testing often and testing early.

Limitations of Testing

- Testing **cannot** generally demonstrate program **correctness.**
- Testing **does not** reveal all defects.
  - Even **well-tested** software typically has **latent defects.**
- Testing every scenario and use case of a software product is, in most cases, either **cost-** or **time-prohibitive**.

Testing Costs

- Principal costs associated with traditional testing include:
  - **Analysis of specification and program.**
  - **Test data generation.**
  - **Program execution.**
  - **The evaluation of the program's behaviors (or determining the correct output in advance).**
- Test generation and evaluation often require much **manual effort:**
  - Developers' and testers' time is **expensive.**
- When defining a testing strategy, it is important to validate that the costs of testing do not exceed the cost of the defect.

Testing and DevOps/Agile

- Traditionally, in the **waterfall** approach, testing was a monolithic effort to validate all the functionality of the application.
- DevOps and Agile principles move away from centralized testing, instead using regression (**reusable**) testing **pipelines** (workflows) that can be **executed**, sometimes, thousands of times, to test user functionality as soon as it is available.
- DevOps and Agile prioritize the usage of low-code tools for testing.
- In DevOps and Agile, the principle is to test often and test early.
- **In Agile,** testing is aligned with the **Acceptance Criteria** of the user stories.

Testing Categorization

- There are many approaches, types, levels, and categories of testing. Each test type, approach, and level can be categorized from multiple perspectives.
  - Use the following categorization structures as test dimensions and as a way to organize, plan, and execute test cases.
- Some examples of common testing categories are:
  - **Static vs. Dynamic.**
  - **White-box vs. Black-Box.**
  - **Functional vs. Non-Functional.**

Testing Categorization - Static vs. Dynamic

- The two main categories of software testing techniques are:
  - **Static Testing:** Testing technique used to find defects **without executing the program.**
  - **Dynamic Testing:** Testing technique that is used to test the dynamic behavior of an application **while the code base is being executed.**

Static Testing

- **Static testing** is a manual process performed to identify and avoid errors at the early stages of the **SDLC**, as it is easier and cheaper to identify and solve errors at these early stages.
- Static testing may also be helpful in identifying **errors** that might not be easy to identify using **dynamic testing**.

Static Testing Techniques

- The two primary techniques used for static testing are:
  - **Reviews**: Manual examinations that are basically code reviews. One primary benefits of reviews is that they support knowledge-sharing practices.
  - **Automated Analysis Tools:** Specialized testing tools are used to automate testing through the analysis of the code. Automated Analysis Tools are effective at detecting implicit programming rule omissions.

Dynamic Testing

- The objective of dynamic testing is to ensure the **consistency** of the software.
- In dynamic testing, the software must be **built, executed and tested.**
- During dynamic testing, test cases are executed with a set of **inputs,** and the **output** is compared to the **expected outputs**.
- **Types**, approaches, and levels of dynamic testing includes:
  - Unit Testing.
  - Integration Testing.
  - System Testing.
  - Acceptance Testing.
  - Functional and Non-Functional Testing.

Dynamic Testing Categorization - White-box vs. Black-box

- Dynamic testing can be classified into two categories:
  - **White-box testing:** Testing of the application with access to the source code.
    - Used to validate the source code and business logic of the software application.
    - Used to validate units, paths, and conditions.
    - Access to source code is required.
    - Coding and implementation expertise are required.
    - These tests are executed by the developers or by white-box testers that have programming knowledge.

Dynamic Testing Categorization - White-box vs. Black-box cont.

- **Black-box testing:** Testing of the application without access to the source code.
  - Used to validate functionality and behaviors.
  - Used in system and acceptance testing.
  - No access to source code is required.
  - No coding expertise or knowledge is required.
  - This type of testing is primarily performed by testers.

Testing Categorization - Functional vs. Non-Functional Testing

- **Tests** can be classified into two categories:
  - **Functional testing:** Tests that validate the software's functional specifications. Functional testing executes functional test cases designed and implemented by the testing or QA team.
  - **Non-Functional testing:** Tests that validate the software's non-functional specifications. Non-functional tests focus on the performance, scalability, and availability of the software solution.

Functional Testing

- **The goal of functional testing is to exercise each specified functional requirement at least once.**
- The objective is to identify requirements that have been:
  - Improperly specified or implemented.
  - Neglected (unimplemented).
- **Functional testing is the most common** type of testing in practice.
- Functional testing may be applied to **any component with a specification:**
  - Method, class, subsystem, system
- **Multiple tests** may be created for **each** requirement.

Example: Functional Testing of Blood Analyzer Software Component

Requirements:

29.A. **PROFILE EDITOR**

29.A.1. The user shall be able to define a collection of related blood tests for the purpose of scheduling convenience and grouping results.

29.A.2. The user shall be able to edit existing profile definitions.

29.A.3. The user shall be able to view profile definitions.

29.A.4. The user shall be able to delete profile definitions.

29.A.5. The user shall be able to uniquely name profiles.

29.A.6. The user shall be able to assign profile name aliases

29.A.7. The user shall be able to assign assays.

29.A.8. All result demographic screen presentations shall have user selectable fields.

Basic functional test criteria:

- Create test collections
- Edit profile definitions
  - Exercise each editing operation
- View profile definitions
- Delete profile definitions
- Name profiles and use their names
- Assign and use profile name aliases
- Assign assays
- Select fields of demographic screen presentations

Example: Functional Tests of Automated Teller Machine - Withdraw Cash - Use Case

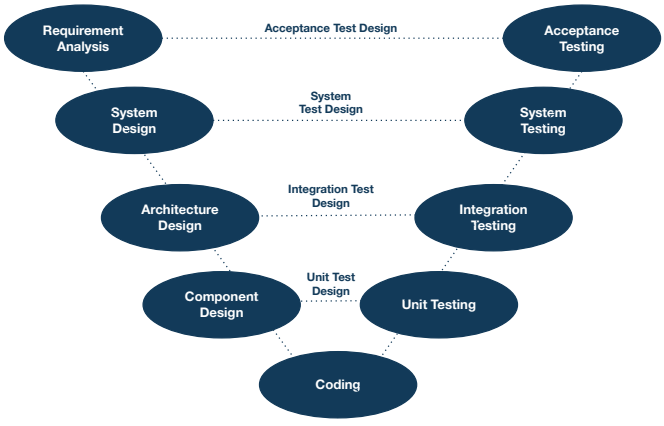
1. Verify the slot for ATM Card insertion is as per the specification
2. Verify that user is presented with options when card is inserted from proper side
3. Verify that no option to continue and enter credentials is displayed to user when card is inserted correctly
4. Verify that font of the text displayed in ATM screen is as per the specifications
5. Verify that touch of the ATM screen is smooth and operational
6. Verify that user is presented with option to choose language for further operations
7. Verify that user asked to enter pin number before displaying any card/bank account detail
8. Verify that there are limited number of attempts up to which user is allowed to enter pin code
9. Verify that if total number of incorrect pin attempts gets surpassed then user is not allowed to continue further- operations like blocking of card etc get initiated
10. Verify that pin is encrypted and when entered
11. Verify that user is presented with different account type options like-saving, current etc
12. Verify that user is allowed to get account details like available balance

Functional Testing Levels

- The four primary levels of functional testing include:
  - Unit Testing
  - Integration Testing
  - System Testing
    - Subsystem Testing
  - Acceptance Testing
    - Alpha and Beta Testing

V-Model in Software Testing

- There are many variants of the V-Model, but the objective is to use different levels of test cases to validate the entire system.



Unit Testing

- Unit testing is the process of **testing individual** components in **isolation**.
  - Unit testing is a defect-testing process.
- Units may be:
  - Individual functions, routines, modules, classes, or methods within an object or class.
  - Object classes with several attributes and methods.
  - Composite components with defined interfaces are used to access functionality.
- There are many different frameworks that facilitate unit testing.
  - A common framework in Java is JUnit.

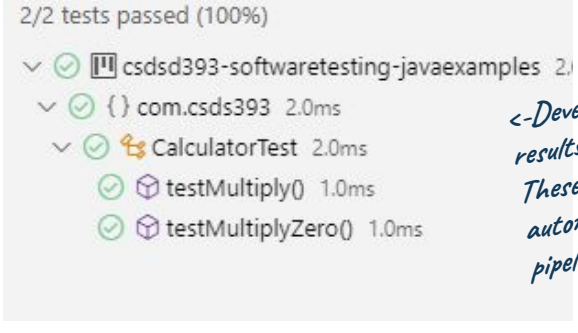
<- Who conducts this unit testing?

Unit Testing - Java - JUnit - Example

```java
public class Calculator {

    public int multiply(int a, int b)
{
        return a * b;
    }
}
```

```java
import static org.junit.Assert.assertEquals;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class CalculatorTest {
    Calculator calculator ;

    @Before
    public void setUp() throws Exception{
        System.out.println("Setting up test.");
        calculator = new Calculator();
    }

    @Test
    public void testMultiply() {
        assertEquals(10, calculator.multiply(2, 5));
    }

    @Test
    public void testMultiplyZero(){
        assertEquals(0, calculator.multiply(0, 0 ));
    }

    @After
    public void tearDown(){
        System.out.println("Test complete.");
    }
}
```


Unit Testing - Java - JUnit - Example cont.



<-Developers can usually see the results of these tests immediately. These tests are also executed automatically as part of the CI/CD pipeline.

Unit Testing - Java - JUnit - Example cont.

Surefire Report

Summary

[Summary] [Package List] [Test Cases]

Tests	Errors	Failures	Skipped	Success Rate	Time
2	0	0	0	100%	0.042

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Package List

[Summary] [Package List] [Test Cases]

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
com.csds393	2	0	0	0	100%	0.042

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

com.csds393

	Class	Tests	Errors	Failures	Skipped	Success Rate	Time
	CalculatorTest	2	0	0	0	100%	0.042

Test Cases

[Summary] [Package List] [Test Cases]

CalculatorTest

	testMultiplyZero	0.005
	testMultiply	0

Copyright © 2022.All rights reserved.

<-Offline reports can be generated by JUnit, and published by the CI/CD pipeline in a radiator.

Integration Testing

- **Integration testing** is a type of testing that involves testing logically **combined** groups of modules or components.
  - The objective of integration testing is to **expose faults in the interaction** between modules or components.
- Integration testing typically happens after **unit tests** are completed and before **system testing**.
- Integration testing can, in most cases, be automated through the use of testing scripts, interfaces, and standardized endpoints.
- Careful integration testing planning is required, in some cases, and depending on the size and type of integration testing, it might be **more expensive than the actual cost of the defects**.

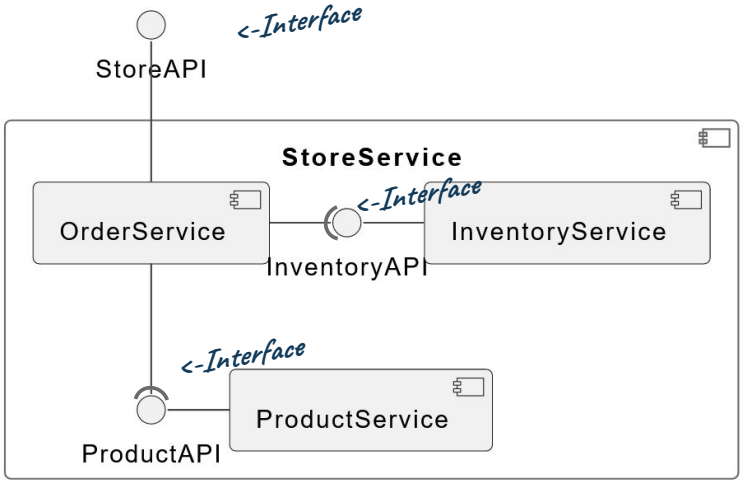
Integration Testing Approaches

- Following are some common approaches for integration testing:
  - **Big Bang Integration Testing:** After unit testing, all the system units are combined and tested at the same time, normally through the use of a common interface. If an issue is detected, an in-depth analysis must be completed to identify the unit with the defect; this activity might be time-consuming depending on the complexity and size of the system.

Integration Testing Approaches

- **Incremental Top-Down Integration Testing:** In this incremental integration testing approach, lower-level integrated modules are tested first, then the process continues to test high-level modules. The disadvantage of this approach is that additional test scripts are needed to test for all the module combinations.
- **Incremental Bottom-Up Integration Testing:** In this incremental integration testing approach, high-level integrated modules are tested first, then the process continues to test low-level modules. The disadvantage of this approach is that additional test scripts are needed to test for all the module combinations.

Integration Testing - Example



Subsystem Testing

- Techniques for **economizing** the cost of the testing efforts.
- It involves dividing the input domain of the program into a finite number of **subdomains or subsystems.**
  - Inputs in each subdomain are treated **similarly** by the specification or program.
  - **One** or a **few** test cases are selected from each subdomain or subsystem.
- **Disjoint** subdomains or subsystems can be viewed as **equivalence classes.**
  - These form a **partition** of the input domain.
- Many testing techniques can be viewed as forms of subdomain or subsystem testing.

System Testing

- The objective of **system testing** is to test the **interactions between the system's components as a whole**.
- System testing validates the **compliance of the system components and interactions** based on the requirements specified in the system design.
  - System testing might include both functional and non-functional testing.
- System tests are end-to-end system tests that evaluate the system, including software and hardware.
- System testing is the sum of a series of different tests that have the purpose of exercising the full system.

System Testing Types

- The following is a non-exhaustive list of tests included in the system testing:
  - **Usability testing:** Tests to validate of the user interface and user accessibility. This testing validates the accessibility requirements.
  - **Functionality testing:** Functionality testing is the process of validating the functionality defined in the software requirements, which is usually associated with features.
  - **Regression Testing:** Tests to ensure that no new defects are introduced as a result of new functionality releases.

*Additional system tests include non-functional system tests.

Acceptance Testing

- The objective of acceptance testing is to **evaluate** the **system's compliance** with the **business requirements and acceptance criteria** defined in the **software requirements specification document (SRS)**.
- An acceptance test is a way for the business and the entire organization to validate that the system is delivering the business value expected and that it complies with the overall expectations of the organization.
- Acceptance testing uses black-box testing, typically performed by business users.

Acceptance Testing cont.

- Acceptance testing can also be performed manually by business users if needed, but in most cases, acceptance testing test cases are automated, and the resulting reports are presented to the business users for evaluation and approval.
  - The objective of automating this type of acceptance test is to provide the business with a way to execute these tests multiple times.
  - Acceptance tests might be performed multiple times, depending on the needs of the organization.
- Acceptance testing is generally performed in an environment that is **as close as possible to production** (typically pre-production or QA).

Acceptance Testing cont.

- Acceptance Testing creates **confidence** that the **functionality** to be delivered **meets the requirements and the business expectations**, but this requires the business to **allocate time** for testing, which might be complicated in certain organizations.
- **Testers** performing acceptance testing need to have a clear idea of what the **business objective** of the platform is, along with other important factors like **operation**, **regulation**, and **contractual** needs.

Acceptance Testing Types

- Following are some common acceptance testing types (variations):
  - **User Acceptance Testing (UAT):** The most common type of acceptance testing, in which business users determine whether a product meets the SRS's business objectives.
  - **Regulation Acceptance Testing:** Acceptance testing that is used to validate that the product meets legal rules and regulations.
  - **Operational Acceptance Testing:** Acceptance testing used to determine that the system meets the non-functional requirements as specified in the SRS.

Alpha and Beta Testing

- **Alpha Testing:** Internal testers use the software simulating the end-user as a way to uncover defects. This testing is performed in the development environment.
- **Beta Testing:** Testing in the field, conducted by the end-user, at the end-user's location.

Non-Functional Testing

- Non-functional testing focuses on validating that **service level agreements** related to non-functional requirements are met.
- Non-functional requirements include quality attributes defined in the software architecture of the solution.
- Non-functional requirements are equally critical for the quality of the solution as functional requirements.

Non-Functional Testing Types

- Non-functional test types include:
  - **Recoverability Test:** Validation to ensure that the system can recover from failure scenarios.
  - **Scalability Testing:** Validation to determine the system's ability to scale.
  - **Security Testing:** Validates the non-functional requirements related to security. Sometimes this testing extends beyond the specified requirements.
  - **Load and performance Testing:** Focuses on the performance and reliability under stress (load).

*This is not an extensive list.

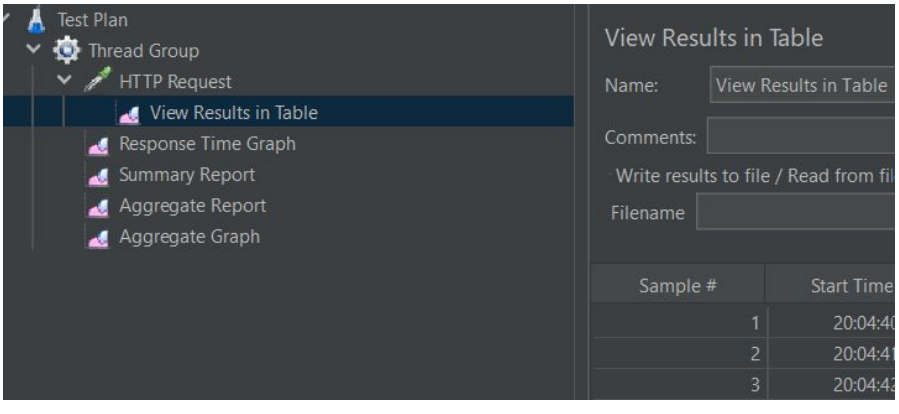
Load and Performance Testing

- Load and performance testing involves “stressing” software with **high loads.** For example:
  - Large input files.
  - Complex inputs.
  - Many concurrent users.
  - Many concurrent service requests.
  - High network traffic.
  - Rapid event-sequences.

Load and Performance Testing cont.

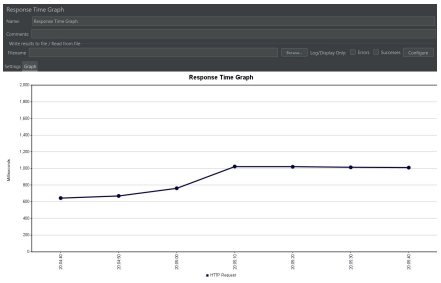
- The load and performance capabilities of a system are directly **correlated** to the **system's hardware characteristics** and **configuration**, but load and performance testing can also uncover important defect in terms of performance, memory leaks, platform configuration, and processing at the component level that can then be subsequently resolved.
- The objective of load and performance testing is to try to push the system to its limits, which, in most cases, **renders it unresponsive.**

Load and Performance Testing - Example - jMeter

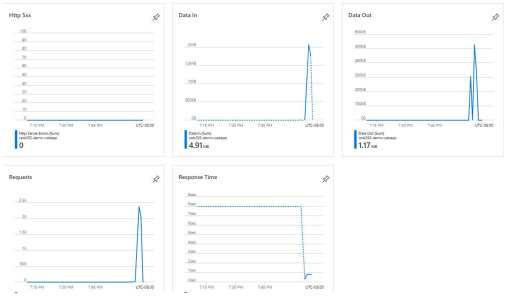


jMeter Test Plan

Load and Performance Testing - Example - jMeter cont.



jMeter Response-time Report



Cloud Web-App Metrics

Aggregate Report

Name: Aggregate Report

Comments:

Write results to file / Read from file

Filename Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/...	Sent KB/sec
HTTP Request	5685	994	1008	1018	1021	1043	21	1270	99.31%	18.1/sec	58.99	0.02
TOTAL	5685	994	1008	1018	1021	1043	21	1270	99.31%	18.1/sec	58.99	0.02

jMeter Aggregate Report

Software Testing Life Cycle

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

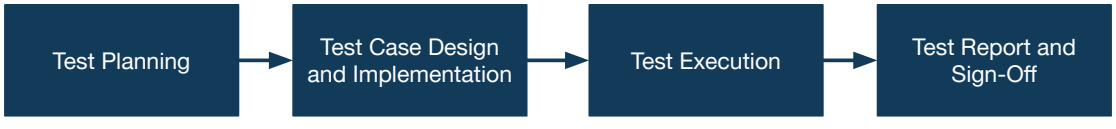
Software Testing Life Cycle

- The **software testing life cycle (STLC)** supports the definition of a test plan for **dynamic testing**.
- The **STLC** includes the following testing phases:
  1. Testing Planning.
  2. Test Case Design and Implementation.
  3. Test Execution.
  4. Test Report and Sign-Off.

<- Iterative, and these can be decomposed further as needed!

Software Testing Life Cycle Phases

**This is a simplified view of the STLC and can be further decomposed.*



<- This process is iterative, and it might happen multiple times in a real-world project.

Test Planning

- During test planning, a Test Plan Document (TPD) is defined. The TPD includes the testing objectives, resources, and processes used to test a specific software product.
- The TPD details the workflow to be followed to successfully test the software product.
- Among the components of this document, we can find:
  - **Scope and Strategy**: A definition of what is going to be tested. <-Key!
  - **Resources**: A definition of what is required to complete the testing (including the environment and human talent).
  - **Risks and Mitigation Plan**: Identified testing risks and expected mitigations.
  - **Traceability Matrix:** Test result/defect tracking mechanism.
  - **Test Schedule:** A plan of when and how long the tests will take.

Test Case Design and Implementation

- The following test artifacts are defined during the test case design and implementation phase:
  - **Test scenarios:** The set of functionality that will be tested.
  - **Test cases:** The set of **tests** to be executed for a particular feature. This requires the definition of a **test type** and **test case** documentation. These test cases include all tests at all levels **(from unit to acceptance tests).**
  - **Test datasets:** The data required to execute all the test cases.
  - **Test scripts:** The set of instructions required to test the application.

Test Case Design and Implementation

- Test case design is a **creative process.**
- The benefits are highly dependent on the **expertise of the testing team.**
- Ideally, testing criteria should be identified and specified:
  - **During requirements analysis and specification.**
  - **During architectural design.**
  - **As each component is designed.**
  - **During design and code reviews.**
- After each stage of the SDLC, relevant **tests should also be reviewed.**
- **A good testing strategy includes tests at all levels, and strives to achieve a high-level of code-coverage.**

Testing Execution

- During this stage, test suites are executed according to the plan previously defined, using the scope, resources, schedule, and traceability matrix defined in the testing planning stage.
- Test suites can be executed either:
  - **Manual testing:** The execution of the test cases is performed manually.
  - **Automated testing:** The execution of the test cases is performed through the use of automation, usually as part of the CI/CD pipeline.

Test Report and Sign-Off

- During this stage, execution results are codified into the tracking matrix. This might result in test cases with a pass state or in the creation of new defects to resolve the failed test cases.
- During this stage, the project's stakeholders meet to discuss and analyze the test results.
- During this stage, the test report is finalized and might include information related to test coverage, test costs, critical defects found, outstanding risks, quality, and other testing metrics.

Specification-Based Testing

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Specification-Based Testing

<-Are there any limitations?

- **Specification-based testing** is a **testing strategy** used to derive and define functional and non-functional test cases.
- As part of the specification-based testing techniques, test cases are **defined by testers** who evaluate the defined requirements, inputs, and outputs of the system.
  - Some of these **input** and **output** specifications **might not** be specifically **stated** in the requirements, so testers must be able to explore and determine what the program does to identify the test cases.
- **Test coverage** of all the requirements **does not indicate a complete test**, but it does indicate that **all specified requirements have been addressed**.

Specification-Based Testing Techniques

- The following are a few examples of specification-based testing techniques; this is by no means an exhaustive list:
  - **User Story Testing:** Test cases are generated based on the user stories' acceptance criteria. Integration test cases can be included to address additional test case scenarios.
  - **Use Case Testing:** Test cases are generated based on the steps outlined in the use case. Alternate paths may also result in additional test cases. At minimum, one use case with one path will require one use case.

Specification-Based Testing Techniques cont.

- **Equivalence Partition Testing:** Test cases are generated by dividing the input data of a software unit into **partitions** of equivalent data. A program with an input integer A, for example, has a valid partition of 1-100 with valid output; all other values result in invalid output. Based on equivalence partition testing, a single test case (input = 4) could result in full test coverage. This technique reduces overtesting.
- **Boundary Value Testing:** Test cases are generated by using the equivalent partitions, e.g., a system with input integer A has a valid partition of 1-100 with valid output, and all other values result in an invalid output. Based on boundary value testing, four test cases with inputs 0, 1, 100, and 101 are required to achieve full coverage.

<-Why would we go from one to four test cases?

Specification-Based Testing Techniques cont.

- **Decision Tables:** Test cases are generated based on conditions that might be the result of a combination of multiple inputs. Decision tables create a **map of combinations** that will result in individual test cases. A decision table contains all the conditions and possible combinations of the conditions that result in a specific output or action, as specified in the requirements.

Code-Based Testing

Code-Based Testing

- Code-based testing is a testing strategy where the objective is to test **each line of code** of a program, as opposed to testing each specification of a program as in specification-based testing.
  - Code-based testing uses a **white-box testing approach**.
  - Code-based testing takes advantage of having **access to language constructs, algorithms, and data structures that are not defined in the requirement specification**.
  - **Implementation-specific** details can also be executed and validated, which increases the likelihood of ensuring the program's correctness.
- Code-based testing can be broken into two categories:
  - **Static Testing.**
  - **Structural Testing.**

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Static Testing

- As previously discussed, static testing does not require the execution of the program.
- **Static testing** can be classified into **four types**:
  - **Informal reviews:** An informal event where developers present the contents to an audience and request feedback. The objective of this event is to identify defects in the early stages of implementation.
  - **Walkthroughs:** A subject-matter expert performs a check in search of defects.
  - **Technical Reviews:** Technical or peer reviews involve validating code in a coordinated way to identify defects.
  - **Code Inspections**: This event involves the verification of the software requirements specification (SRS).

Structural Testing

- Exercises or **covers** each program element of a certain type, e.g.,
  - Statements or basic blocks.
  - Conditional branches.
  - Control flow paths (in control the flow graph).
  - Definition-use (data flow) chains.
- Motivation:
  - **Any such element may be defective.**
  - **Simply executing it may trigger failure.**
- **Percentage coverage** achieved by the test set is one measure of testing **completeness.**
  - **This percentage coverage is inadequate** by itself.

Structural Testing Categories

- Structural testing can be broken into four categories:
  - **Control Flow Testing:** Test cases are defined based on the way statements and instructions are executed.
    - Most common type of structural testing.
    - Control flow testing uses **control flow graphs** (CFGs)
  - **Data Flow Testing (Analysis):** Set of testing strategies used to identify the sequence of events related to the usage of variables and data objects.
    - Variables that are declared but not used.
    - Variables used but not declared.
    - Variables are defined several times before they are used.
    - Deallocation of variables before first use.

Structural Testing Categories

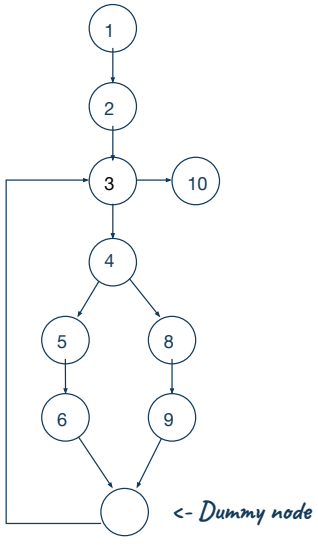
  - **Slice-based Testing:** The objective of this strategy is to slice the program into small sections of particular interest that can be evaluated individually.
    - This strategy is also helpful for debugging.

Control Flow Testing

- The objective of this technique is to determine the execution order of a statement or set of instructions through the definition of a control structure. The control structure is used to develop the test cases for the program.
- Control Flow Graphs(CFG) depict the potential control flow between the program statement and instructions.
- Each simple statement and decision point is represented by a vertex.
- Each potential branch is represented by a direct edge.
  - A decision vertex has two or more edges.
- **Control flow coverage** criteria may be defined in terms of a CFG.

Example: Control Flow Graph

```
1   input(x, y)
2   z ← 1
3   while y > 0 do
4       if even(y)
5           y ← y div 2
6           z ← x * x
7       else
8           y ← y - 1
9           z ← z * x
10  output(z)
```



CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Control Flow Coverage Criteria

- **Statement coverage:** has each statement been executed by the test set?
- **Branch coverage:** has each conditional branch been executed?
- **Condition coverage:** has each boolean subexpression of a branch condition been evaluated as both true and false?
- **Function coverage:** has each function in the program been called?
- **Entry/exit coverage:** has every possible call and return of each function been executed?
- **Loop coverage:** has every possible loop been executed zero times, once, and more than once?
- **Path coverage:** has every complete control flow path been executed?

Measuring Coverage Achieved By Functional Tests

- Test coverage is a common industrial practice, but sometimes it is misused.
- If the test coverage is not adequate, it is best **not** to just create ad-hoc tests to boost it.
  - These ad-hoc tests are generally designed and tailored to pass and increase the test coverage, voiding the actual objective of a test.
- Instead, the **requirements specification** should be examined:
  - **Omissions** in the specification should be identified.
  - **New functional tests** to be exercised should be constructed.
  - This process should be iterated until reaching the test coverage goal.

Fault-Based Testing

Fault-Based Testing

- Fault-based testing is a **paradigm shift** from previous strategies where testers aspired to prevent faults in products.
- Instead, fault-based testing aspires to obtain **information** from **intentionally created faults.**
- In fault-based testing, **mutated** versions of the program are created, and each one of the mutations is **seeded** with a small fault.
  - Tests are designed to distinguish the real program from the mutated program.
  - The paradigm, as a result, is that tests will evolve to detect mutated programs, which subsequently will improve the overall quality of the tests.

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Fault-Based Testing

- **Seeded faults** involve adding faults to the original program that align with a specific kind of programming fault, e.g.,
  - **Incorrect choices of arithmetic, relational, or logical operators.**
  - **Erroneous variable substitutions**
  - **Incorrect constants.**
- Test data is selected to distinguish a supposed fault from a supposed correct code:
  - E.g., (a = 1, b = 0) distinguishes (a & & b) from (a | | b).
- These small changes called **mutations** are automatically seeded into a program, one at a time, creating mutant versions.

Mutation Testing

- A mutant **is said to be "killed"** if a test case produces a **different result** than the one from the respective original program. A different test case result, compared to the test case result using the original program, means that the test case effectively detected the mutation.
  - Otherwise, the **mutant "survived."**
- The **quality** of the tests can be estimated from the **percentage of mutations "killed."** The higher the percentage of mutations "killed" vs. "survived", the higher the quality of the test case.

Fault-Based Example Process (Activities)

1. A program without defects is created.
2. 100 different program mutations are created; each mutation is seeded with a small modification, which is a program defect.
3. The assumption is that those defects are real types of defects present in other programs.
4. The test case is executed on all 100 mutated programs. The test case results reveal only 20 defective mutations; these mutated programs are said to have been "killed." The other 80 mutations do not fail the test case; these mutated programs are said to have "survived." *<-What is the expectation here?*
5. Based on this information, we can determine the quality of our test case by judging the results of "killed" vs. "survived" mutations.
6. If many mutations are "killed," we can infer that the test suite is effective at finding real defects.

Criticisms of Mutation Testing

- There may be an enormous number of possible mutants.
- It is questionable whether “killing” simple mutants helps reveal complex faults.
- There is no generally effective way to automatically generate test cases that “kill” mutants.

Interaction Testing

Interaction Testing

- The **interaction testing strategy** involves the **definition of test cases** for testing **interactions between variables, objects, events, statements, or components**.
  - Interaction testing, therefore, is testing how objects send and receive data from and to other objects.
- Interaction testing may reveal defects that simple test coverage doesn't reveal.
  - Defects related to interactions are difficult to detect with testing.
  - Interaction issues are gradually discovered once the system has been delivered. *<-Key!*
  - Interaction issues are difficult to reproduce.
- **Problem:** The number of possible interactions (combinations) grows rapidly with the number of elements involved.

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Mocks

- Mocks are utility objects that are used to decide whether a unit test has passed or failed. This verification is completed by verifying whether the object under test interacts with other objects as expected through the use of mock objects.
  - The mock object defined a boundary between interacting objects, which simplifies testing.
  - The mock object also saves information regarding the interactions between objects, which can then be used later to validate a test case.

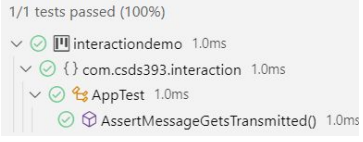
Mock Example

```
public interface WebService {
    void submitMessage(String message);
}

public class MockWebService implements WebService {
    private String lastMessage;

    public void submitMessage(String message)  {
        lastMessage = message;
    }

    public String getLastMessage(){
        return lastMessage;
    }
}

public class Logger {

    private WebService service;

    public Logger(WebService service){
        this.service = service;
    }

    public void logToService(String message){
        service.submitMessage("Message:" + message);
    }

}
```

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Mock Example cont,

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class AppTest
{

    @Test
    public void AssertMessageGetsTransmitted()
    {

        MockWebService mockWebService = new MockWebService();
        Logger log = new Logger(mockWebService);
        String message = "Hello,World!";
        log.logToService(message);
        assertEquals("Message:"+message, mockWebService.getLastMessage());
    }
}
```



When to Use Mock Objects

- Mock objects can be used when:
  - Interactions use defined interfaces.
  - Interactions are difficult to test using other strategies.
  - Specific interaction behaviors are difficult to execute.

Shortcomings of Mocks

- Mocks may hide integration problems.
- Mocks add additional test code.
- Tests using mocks may be fragile if interfaces do not stay consistent.
  - Changes to the interface implementation may break the tests using mocks.

Choosing Among Testing Strategies <-Key!

- The use of a combination of diverse testing techniques highlights the **variety** of possible faults.
  - Multiple strategies can be combined.
- In choosing techniques, developers must often rely on **judgment and experience**.
- The best **combination** of test technique choice may be **context-specific.**
- Companies should **collect and analyze data** about the **effectiveness** of different techniques for revealing faults.

Field Testing

Field Testing

- In field testing, software is **deployed** in the field to one or more sites, multiple users.
- Field testing is also called **operational testing** or **beta testing.**
- In field testing, the end-user uses and evaluates the product for a period of time, that ranges from weeks to months.
  - Users employ the software product as **they see fit and** report failures as feedback.
- Users involved in field testing are aware of the **service-level agreement limitations** (SLAs) that imply using the service in a non generally-available (GA or beta) context. In most cases, the software product providers (development organizations) do not offer the same level of SLAs compared to GA functionality.

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Capture/Replay of Field Executions

- Field-testing allows testers to capture and replace the customer's activities on the platform.
  - **Inputs** and other execution data are captured in the field can be **replayed offline.**
- **This capability supplements the** end-user evaluation.
- Capture and replay capabilities require specialized software and architecture configurations.
- Important behavior and metrics are reviewed by the system provider.
- Captured inputs may be **reused** for **regression testing.**

Site and User Selection

- Field testing is analogous to a **survey.**
- Ideally, **probability sampling** is used for site and user selection.
- Lack of cooperation (**nonresponse**) can **bias** results:
  - **Persistence and incentives** may increase end-user cooperation.
  - **A statistical adjustment** may help compensate for bias or the lack of homogeneity among the users.

Advantages of Field Testing

- Field testing allows development organizations to exercise the software product under **realistic operating conditions.**
- In field testing, the program's inputs and outputs are **not** influenced by **developer biases.**
- In field testing, end-users provide valuable **feedback:**
  - **Frequent**, **impactful** failure types are most likely to be reported immediately.
    - This feedback facilitates reliability improvements.
- Field testing supports the **measurement** of operational **reliability.**

Disadvantages of Field Testing

- Field testing can be **difficult** to arrange and manage with customer organizations.
  - Field testing is easier to coordinate with individual users via the Internet.
- The plan to release a software must include the field testing phase, which might delay the software's release to production depending on the field testing results.
- Field testing creates a reliance on end users:
  - End-users are not trained testers.
  - End-users have other priorities.
  - End-users may **overlook** or **inaccurately report** failures.
    - This creates a problematic situation for reliability estimation.

Disadvantages of Field Testing cont.

- There might be situations where clients do not wish to use or access non-GA or beta functionality because this involves a double time-investment:
  - Time investment to validate the non-GA functionality.
  - Time investment to validate the released functionality.
    - This functionality might not be released or might be different from the non-GA or beta functionality.

Newer Forms of Field Testing

- **"Dogfooding" :** The provider use its the new service/feature before it is released to the customer.
- **Canary releases:** Functionality is rolled out to only a small subset of users.
- **Gradual rollouts:** Release functionality gradually to users.
- **Online Controlled Experiments or A/B testing:** Variant of experiments are released to different subset of users to proof an hypothesis.

Regression Testing

Regression Testing

- **Regression testing** is a type of system testing to **validate** that **changes introduced** as part of a **new release increment** have not broken previous working code.
- Normally, regression testing uses **automation** to rerun every single test every time there is a change implemented in the product.
- The entire **test suite** of the application must **pass** before a change can be **committed** to the main branch. This is achieved using automation.

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Regression Testing cont.

- It is standard practice to **reuse** test cases when possible:
  - **Reuse reduces costs** for analysis, test generation, and test evaluation.
  - Regression test suites may grow **very large.**
- Some tests may be intentionally **omitted**, but with automation, typically all tests are executed.
- **Additional tests** may be needed to cover **new features or code.**
- **Field replays** can be used as part of regression testing to address the lack of variation in the test cases.

Test-Driven Development

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Test-Driven Development

- Test-Driven Development (TDD) is a development style in which automated tests for new functionality are written **before** the functionality is implemented.
- The goal of TDD is "clean code that works."
- TDD is closely linked to **Agile** development.

TDD Steps in Detail

1. **Examine the specifications for the requirements.**
2. **Create a test case for the requirement without coding.**
   - Create the new **code stub**. A simple program so that the code compiles.
   - Run the test case. This should result in a failed test case. This test is to ensure that the test is using the correct stub.
3. **Make the test pass** by implementing the code required.
   - The objective is to make the test pass with the simplest possible code.
   - When the test passes, run a regression test.
4. Refactor the code as necessary to simplify the design and remove duplicated code.
   - After every modification, run a regression test.
5. Repeat the cycle as needed.

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible

Two Strategies for Getting to Pass State

- **Constant Returns**: Return a constant and gradually replace constants with variables until you have the real code.
- **Obvious Implementation**: Type in the real implementation.

Benefits of TDD

- Executing a test suite linked to TDD assures that each component is still working, after changes have been made.
- Test suites can act as documentation.
- TDD forces critical analysis because the developer cannot create the production code without truly understanding the desired outcome.
- It is easier to detect flaws because they are usually the cause of a regression test failing.

Disadvantages of TDD

- TDD might create tunnel vision towards only making tests pass without really understanding the full set of requirements.
- Incomplete requirements also affect the implementation.
- TDD is difficult with end-to-end tests or tests involving integrated components and services.
- A passing test suite may create a false sense of security.
- Automated test suites require significant maintenance.

Synthetic Testing

CASE WESTERN RESERVE UNIVERSITY EST. 1826
think beyond the possible