

# Qualidade de Software (14450)

## Testing Fundamentals

(adapted from lecture notes of the “DIT 635 - Software Quality and Testing” unit,  
delivered by Professor Gregory Gay, at the Chalmers and the University of Gothenburg, 2022)

# Today's Goals

---

## ✧ Some definitions and concepts:

- Let's get the language right.
- What are the components of a test case?

## ✧ Testing stages:

- Unit, System (Integration and Exploratory), and Acceptance Testing

## ✧ Test planning considerations

# Verification

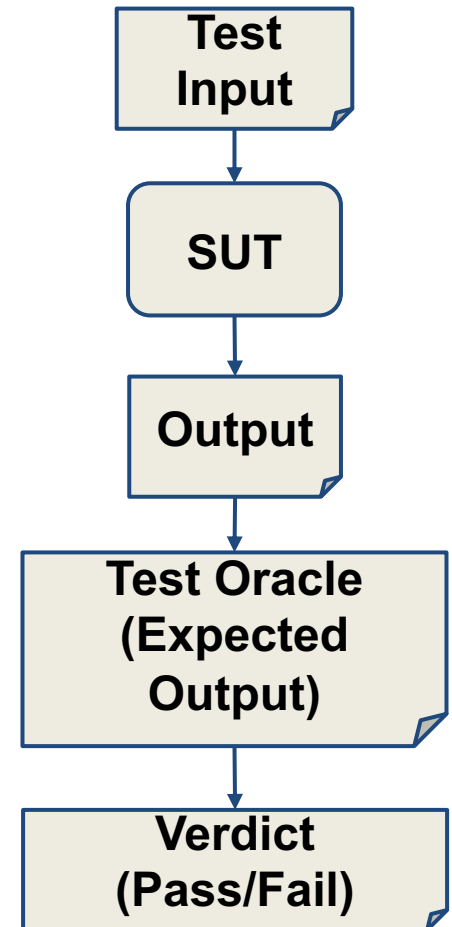
---

- ✧ Ensuring that an implementation conforms to its specification.
  - AKA: Under these conditions, does the software work?
- ✧ Proper V&V produces dependable software.
  - **Testing is the primary verification activity.**

# Software Testing

---

- ✧ An investigation into system quality.
- ✧ Based on sequences of **stimuli** and **observations**.
  - **Stimuli** that the system must react to.
  - **Observations** of system reactions.
  - **Verdicts** on correctness.



# Bugs? What are Those?

---

✧ **Bug is an overloaded term.**

- Does it refer to the bad behavior observed?
- Is it the source code mistake that led to that behavior?
- Is it both or either?

# Faults and Failures

## ✧ Failure

- An execution that yields an incorrect result.

## ✧ Fault

- The problem that caused a failure.
- Mistake in the code, omission from the code, misuse.

✧ **When we *observe a failure*, we try to *find the fault*.**



```
12 if(ArrivalTime != current.ArrivalTime) {
13     current.ArrivalTime = ArrivalTime;
14     current.WIPs = [];
15     current.Cantidades = [];
16 }
17
18 current.labels.assert("WIP", []);
19 current.labels.assert("Cantidades", []);
20 int indice = current.WIPs.indexOf(WIP);
21
22 if(indice == -1) {
23     current.WIP.push(WIP);
24     current.Cantidades.push(1);
25     indice = current.Cantidades.length-1;
26 }
27 else
28     current.Cantidades[indice] += 1;
29
30 string Cantidad = string.fromSum(current.Cantidades[indice]);
31 string query;
32
33 if(Cantidad == "1") {
34     query = concat("insert into 'flexsim'.JamonesOutput' (ArrivalTime, WIP, Cantidad, PT, WIP) values ('", Arriv
35 )
36 }
37 else {
38     query = concat("update 'flexsim'.JamonesOutput' set Cantidad = '", Cantidad, "' where WIP = '", WIP, "' and Arriv
39 )
40 dbopen("flexsim", "select * from JamonesOutput", 0);
41 dbqlquery(query);
42 dbclose();
```

# Software Testing

---

✧ **The main purpose of testing is to find faults:**

“Testing is the process of trying to discover every conceivable fault or weakness in a work product”  
- Glenford Myers

✧ Tests must reflect normal system usage and extreme boundary events.

# Testing Scenarios

---

## ✧ **Verification:**

- Demonstrate that software meets the specification.
- Tests tend to reflect “normal” usage.
- Any lack of conformance is a fault.

## ✧ **Resilience:**

- Show that software can handle rare/extreme situations.
- Tests tend to reflect extreme usage.
  - Large volume of data, null data, malformed data, attacks.



## Axiom of Testing

---

“Program testing can be used to show the presence of bugs, but **never their absence.**”

- Dijkstra

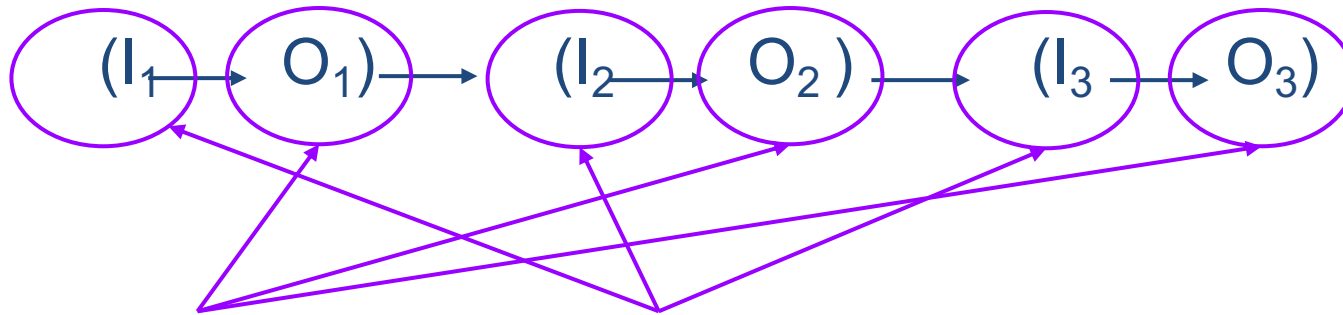
# Test Suite and Test Case

---

- ✧ A **test suite** is a collection of **test cases**.
  - Executed together.
  - Each test case should be independent.
- ✧ May have multiple suites in one project.
  - Different types of tests, different resource/time needs.
- ✧ A test case consists of:
  - Initialization, Test Steps, Inputs, Oracles, Tear Down

# Anatomy of a Test Case

---



## Test Inputs

How we “stimulate” the system.

if  $O_n = \text{Expected}(O_n)$   
then... Pass  
else... Fail

## Test Oracle

How we check the correctness of the resulting observation.

# Anatomy of a Test Case

---

## ✧ Initialization

- Any steps that must be taken before test execution.

## ✧ Test Steps

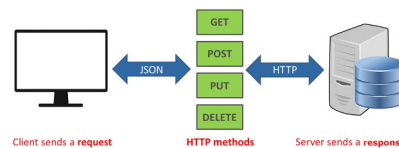
- Interactions with the system, and comparisons between oracle and actual values.

## ✧ Tear Down

- Any steps that must be taken after test execution.

# Test Input

- ✧ Any deliberate interactions with a software feature.
  - Generally, call a function through an interface.
  - Method Call
  - API Call
  - CLI Interaction
  - GUI Interaction



The screenshot shows an IDE with two panes. The left pane displays a project tree with the following structure:

- StringEqualContract.java (Selected)
- StringEqualContractTest.java
- StringEqualContractTest2.java
- StringEqualContractTest3.java
- StringEqualContractTest4.java
- StringEqualContractTest5.java
- StringEqualContractTest6.java
- StringEqualContractTest7.java
- StringEqualContractTest8.java
- StringEqualContractTest9.java
- StringEqualContractTest10.java
- StringEqualContractTest11.java
- StringEqualContractTest12.java
- StringEqualContractTest13.java
- StringEqualContractTest14.java
- StringEqualContractTest15.java
- StringEqualContractTest16.java
- StringEqualContractTest17.java
- StringEqualContractTest18.java
- StringEqualContractTest19.java
- StringEqualContractTest20.java
- StringEqualContractTest21.java
- StringEqualContractTest22.java
- StringEqualContractTest23.java
- StringEqualContractTest24.java
- StringEqualContractTest25.java
- StringEqualContractTest26.java
- StringEqualContractTest27.java
- StringEqualContractTest28.java
- StringEqualContractTest29.java
- StringEqualContractTest30.java
- StringEqualContractTest31.java
- StringEqualContractTest32.java
- StringEqualContractTest33.java
- StringEqualContractTest34.java
- StringEqualContractTest35.java
- StringEqualContractTest36.java
- StringEqualContractTest37.java
- StringEqualContractTest38.java
- StringEqualContractTest39.java
- StringEqualContractTest40.java
- StringEqualContractTest41.java
- StringEqualContractTest42.java
- StringEqualContractTest43.java
- StringEqualContractTest44.java
- StringEqualContractTest45.java
- StringEqualContractTest46.java
- StringEqualContractTest47.java
- StringEqualContractTest48.java
- StringEqualContractTest49.java
- StringEqualContractTest50.java
- StringEqualContractTest51.java
- StringEqualContractTest52.java
- StringEqualContractTest53.java
- StringEqualContractTest54.java
- StringEqualContractTest55.java
- StringEqualContractTest56.java
- StringEqualContractTest57.java
- StringEqualContractTest58.java
- StringEqualContractTest59.java
- StringEqualContractTest60.java
- StringEqualContractTest61.java
- StringEqualContractTest62.java
- StringEqualContractTest63.java
- StringEqualContractTest64.java
- StringEqualContractTest65.java
- StringEqualContractTest66.java
- StringEqualContractTest67.java
- StringEqualContractTest68.java
- StringEqualContractTest69.java
- StringEqualContractTest70.java
- StringEqualContractTest71.java
- StringEqualContractTest72.java
- StringEqualContractTest73.java
- StringEqualContractTest74.java
- StringEqualContractTest75.java
- StringEqualContractTest76.java
- StringEqualContractTest77.java
- StringEqualContractTest78.java
- StringEqualContractTest79.java
- StringEqualContractTest80.java
- StringEqualContractTest81.java
- StringEqualContractTest82.java
- StringEqualContractTest83.java
- StringEqualContractTest84.java
- StringEqualContractTest85.java
- StringEqualContractTest86.java
- StringEqualContractTest87.java
- StringEqualContractTest88.java
- StringEqualContractTest89.java
- StringEqualContractTest90.java
- StringEqualContractTest91.java
- StringEqualContractTest92.java
- StringEqualContractTest93.java
- StringEqualContractTest94.java
- StringEqualContractTest95.java
- StringEqualContractTest96.java
- StringEqualContractTest97.java
- StringEqualContractTest98.java
- StringEqualContractTest99.java
- StringEqualContractTest100.java

The right pane displays the content of 'StringEqualContract.java':

```

package org;

import static org.junit.Assert.*;

import org.junit.Assert.*;

public interface EqualContract {

    T createEqualValue();

    @NotNull
    default void equalsContract(T value, T value2) {
        assertEquals(value, value2);
    }
}

```

```
chris@ubuntu: ~  
chris@ubuntu:~$ bash --version  
GNU bash, version 4.3.46(1)-release (x86_64-pc-linux-gnu)  
Copyright (C) 2013 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.  
  
This is free software; you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
chris@ubuntu:~$
```

# Test Input

---

## ✧ Environment manipulation

- Set up a database with particular records
- Set up simulated network environment
- Create/delete files
- Control available CPU/memory/disc space

## ✧ Timing

- Before/at/after deadline
- Varying frequency/volume of input

# Test Creation and Execution

---

## ✧ Can be **human-driven**

- Exploratory testing, alpha/beta testing

## ✧ or **automated**

- Tests written as code
  - Testing frameworks (JUnit)
  - Frameworks for manipulating interfaces (Selenium)
- Capture/replay tools can re-execute UI-based tests (SWTBot for Java)
- Automated input generation (AFL, EvoSuite)

# Sources of Input

---

## ✧ **Black Box (Functional) Test Design**

- ✧ Use knowledge about how the system should act to design test cases.
  - Requirements, comments, user manuals, intuition.
- ✧ Tests can be designed before code is written.
  - (test-driven development)



# Sources of Input

---

## ✧ **White Box (Structural) Test Design**

✧ Input chosen to exercise part of the code.

✧ Usually based on **adequacy criteria**:

- Checklists based on program elements.
- **Branch Coverage** - Make all conditional statements evaluate to all outcomes (if-statements, switches, loops)

✧ Fill in the gaps in black-box test design.

# Test Oracle - Definition

---

- ✧ A predicate that determines whether a program is correct or not.
  - Based on observations of the program.
  - Output, timing, speed, energy use, ...
- ✧ Will respond with a **pass** or a **fail** verdict.
- ✧ Can be specific to one test or more general.

# Test Oracle Components

---

## ✧ Oracle Information

- Embedded information used to judge the correctness of the implementation, given the inputs.

## ✧ Oracle Procedure

- Code that uses that information and relevant observations to arrive at a verdict.
- Often as simple as...

```
if (actual value != expected value) { fail (...); }  
assertEquals(actual value, expected value);
```

# Oracles are Code

---

## ✧ Oracles must be developed.

- Like the project, an oracle is built from the requirements.
  - ... and is subject to interpretation by the developer
  - ... and may contain faults

## ✧ A faulty oracle can be trouble.

- May result in false positives - “pass” when there was a fault in the system.
- May result in false negatives - “fail” when there was not a fault in the system.

## Expected-Value Oracles

---

✧ Simplest oracle - what exactly should happen?

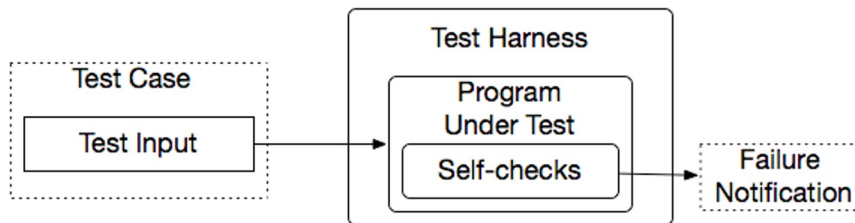
```
int expected = 7;  
int actual = max(3, 7);  
assertEquals(expected, actual);
```

✧ Oracle written for a single test case, not reusable.

# Property-based Oracles

---

Rather than comparing actual values, use properties about results to judge sequences.



```
@Test
```

```
public void propertiesOfSort (String[] input) {
```

```
// Tests
```

```
String[] sorted = quickSort(input);
```

```
assert(sorted.size >= 1, "This array can't be empty.")
```

```
for (int item = 1; item < sorted.length; item++)
```

```
    assert(sorted[item] > sorted[item - 1], "Items  
        should be sorted in ascending order");
```

```
}
```

Uses assertions, contracts, and other logical properties.

# Properties

---

- ✧ Usually written at “function” level.
  - For a method or high-level API/UI function.
  - Properties based on behavior of that function.
- ✧ Work for any input to that function.
- ✧ Trade-off: limited by number of properties.
  - Faults missed even if specified properties are obeyed.
  - More properties = more expensive to write.

# Implicit Oracles

---

- ✧ Check properties expected of any program.
  - Crashes and exceptions.
  - Buffer overruns.
  - Deadlock.
  - Memory leaks.
  - Excessive energy usage or downloads.
- ✧ Faults that do not require expected output to detect.



# Testing Stages

---

- ✧ We interact with **systems** through **interfaces**.
  - APIs, GUIs, CLIs
- ✧ Systems built from **subsystems**.
  - With their own interfaces.
- ✧ Subsystems built from **units**.
  - Communication via method calls.
  - Set of methods is an interface.

# Testing Stages

---

## ✧ Unit Testing

- Do the methods of a class work?

## ✧ System-level Testing

- **System (Integration) Testing**
  - (Subsystem-level) Do the collected units work?
  - (System-level) Does high-level interaction through APIs/UIs work?
- **Exploratory Testing**
  - Does interaction through GUIs work?

# Testing Stages

---

## ✧ Acceptance Testing / AB Testing

- Give product to a set of users to check whether it meets their needs.
  - Alpha/beta Testing - controlled pools of users, generally on their own machine.
  - Acceptance Testing - controlled pool of customers, in a controlled environment, formal acceptance criteria
- Can expose many faults.
- Can be planned during requirements elicitation.

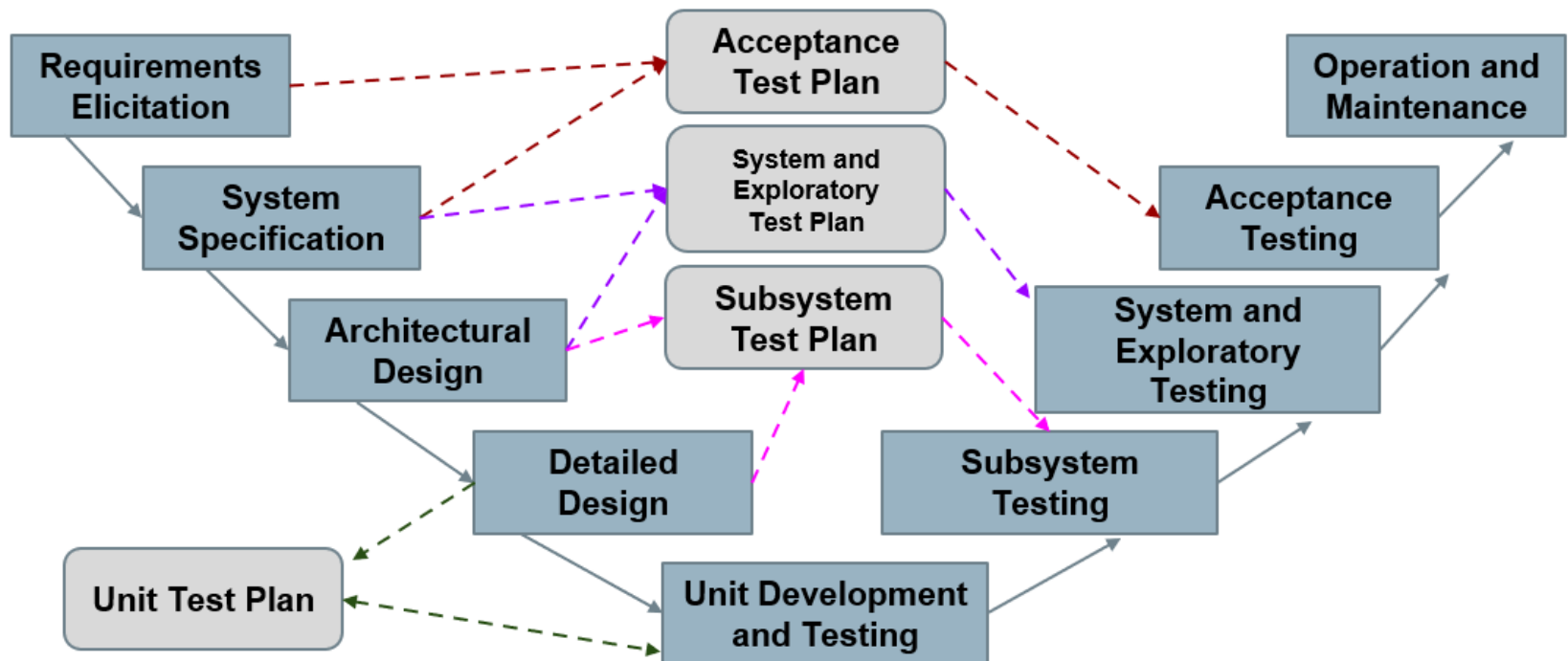
# Automation vs Human-Driven

---

- ✧ Unit/System Testing heavily use automation.
  - Tests written as code.
  - Executed repeatedly, often on check-in.
- ✧ Exploratory/Acceptance Testing often human-driven
  - Humans interact with app.
  - Based on scenarios, without pre-planned input.
  - Some tool support, but not often repeated exactly.

# The V-Model of Development

---



# Unit Testing

---

- ✧ Testing the smallest “unit” that can be tested.
  - Often, a class and its methods.
- ✧ Tested in **isolation** from all other units.
  - **Mock** the results from other classes.
- ✧ Test input = method calls.
- ✧ Test oracle = assertions on output/class variables.

# Unit Testing

---

✧ For a unit, tests should:

- Test all “jobs” associated with the unit.
  - Individual methods belonging to a class.
  - Sequences of methods that can interact.
- Set and check class variables.
  - Examine how variables change after method calls.
  - Put the variables into all possible states (types of values).

WeatherStation
identifier temperature pressure
checkLink() reportWeather() reportInstrumentStatus() restart(instrumentName) shutdown(instrumentName) reconfigure(instrumentName, commands)

# Unit Testing

---

## ✧ Unit tests should cover:

- Set and check class variables
  - Can any methods change identifier, temperature, pressure?
- Each “job” performed by the class.
  - Single methods or method sequences.
  - Vary the order methods are called.
  - Each outcome of each “job” (error handling, return conditions).

WeatherStation
identifier temperature pressure
checkLink() reportWeather() reportInstrumentStatus() restart(instrumentName) shutdown(instrumentName) reconfigure(instrumentName, commands)



# System (Integration) Testing

---

✧ After testing units, test their **integration**.

- Integrate units in one subsystem.
- Then integrate the subsystems.

✧ Test input through a defined interface.

- Focus on showing that functionality accessed through interfaces is correct.
- Subsystems: “Top-Level” Class, API
- System: API, GUI, CLI, ...

# System Testing

---

Subsystem made up classes of A, B, and C. We have performed unit testing...

- ✧ Classes work together to perform subsystem functions.
- ✧ Tests applied to the interface of the subsystem they form.
- ✧ Errors in combined behavior not caught by unit testing.

# Interface Errors

---

## ✧ Interface Misuse

- Malformed data, order, number of parameters.

## ✧ Interface Misunderstanding

- Incorrect assumptions made about called component.
- A binary search called with an unordered array.

## ✧ Timing Errors

- Producer of data and consumer of data access data in the wrong order.

# GUI Testing

---

- ✧ Tests designed to reflect **end-to-end** user journeys.
  - From opening to closing.
  - Often based on **scenarios**.
- ✧ GUI Testing
  - Deliberate tests, specific input.
  - May be automated or human-executed.
- ✧ Exploratory Testing
  - Open-ended, human-driven exploration.

# Exploratory Testing

---

- ✧ Tests are not created in advance.
- ✧ Testers check the system on-the-fly.
  - Guided by scenarios.
  - Often based on ideas noted before beginning.
- ✧ Testing as a thinking idea.
  - About discovery, investigation, and role-playing.
  - Tests end-to-end journeys through app.
  - Test design and execution done concurrently.

# Exploratory Testing

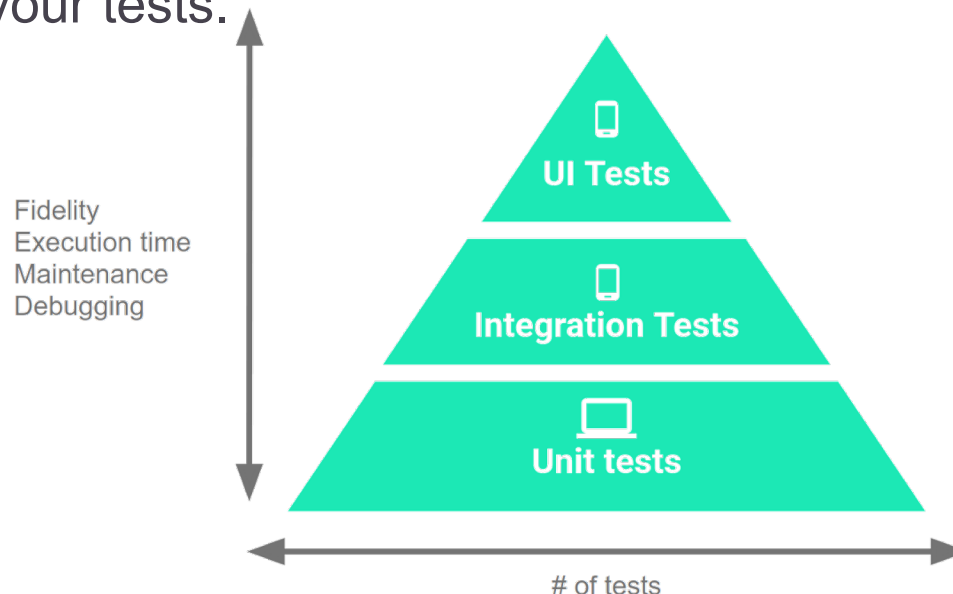
---

- ✧ Tester write down ideas to give direction, then create critical, practical, and useful tests.
  - Requires minimal planning. Tester chooses next action based on result of current action.
- ✧ Can find subtle faults missed by formal testing.
  - Allows tester to better learn system functionality, and identify new ways of using features.

# Testing Percentages

---

- ✧ Unit tests verify behavior of a single class.
  - 70% of your tests.
- ✧ System tests verify class interactions.
  - 20% of your tests.
- ✧ GUI tests verify end-to-end journeys.
  - 10% of your tests.



# Testing

---

- ✧ 70/20/10 recommended.
- ✧ Unit tests execute quickly, relatively simple.
- ✧ System tests more complex, require more setup, slower to execute.
- ✧ UI tests very slow, may require humans.
- ✧ Well-tested units reduce likelihood of integration issues, making high levels of testing easier.



# Acceptance Testing

---

Once the system is internally tested, it should be placed in the hands of users for feedback.

- ✧ Users must ultimately approve the system.
- ✧ Many faults do not emerge until the system is used in the wild.
  - Alternative operating environments.
  - More eyes on the system.
  - Wide variety of usage types.

# Acceptance Testing Types

---

## ✧ Alpha Testing

- A small group of users work closely with development team to test the software.

## ✧ Beta Testing

- A release of the software is made available to a larger group of interested users.

## ✧ Formal Acceptance Testing

- Customers decide whether or not the system is ready to be released.

# Acceptance Testing Stages

---

## ✧ Define acceptance criteria

- Work with customers to define how validation will be conducted, and the conditions that will determine acceptance.

## ✧ Plan acceptance testing

- Decide resources, time, and budget for acceptance testing. Establish a schedule. Define order that features should be tested. Define risks to testing process.

# Acceptance Testing Stages

---

## ✧ Derive acceptance tests.

- Design tests to check whether or not the system is acceptable. Test both functional and non-functional characteristics of the system.

## ✧ Run acceptance tests

- Users complete the set of tests. Should take place in the same environment that they will use the software. Some training may be required.

# Acceptance Testing Stages

---

## ✧ Negotiate test results

- It is unlikely that all of the tests will pass the first time. Developer and customer negotiate to decide if the system is good enough or if it needs more work.

## ✧ Reject or accept the system

- Developers and customer must meet to decide whether the system is ready to be released.

# Test Plans

---

✧ Plan for how we will test the system.

- **What** is being tested (units, subsystems, features).
- **When** it will be tested (required stage of completion).
- **How** it will be tested (what scenarios do we run?).
- **Where** we are testing it (types of environments).
- **Why** we are testing it (what purpose do tests serve?).
- **Who** will be responsible for writing test cases (assign responsibility to team members).

# Why Make a Test Plan?

---

- ✧ Guides development team.
  - Rulebook for planning test cases.
- ✧ Helps people outside the team understand the testing process.
- ✧ Documents rationale for scope of testing, how we judge results, why we chose a strategy.
  - Can be reused when making decisions in future projects.

# Analyze the Product

---

- ✧ Must understand the product before you can test it.
  - What are the needs of the users?
  - Who will use the product?
  - What will it be used for?
  - What are the dependencies of the product?
- ✧ Review requirements and documentation.
- ✧ Interview stakeholders and developers.
- ✧ Perform a product walkthrough (if code is running).

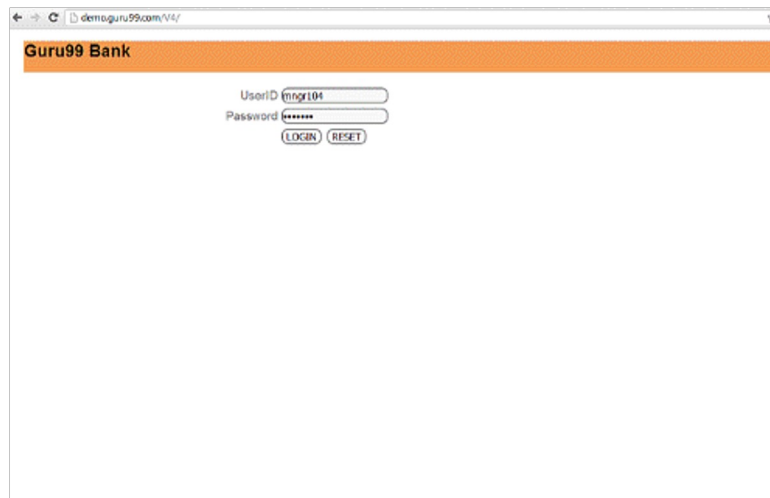


# Analyze the Product

---

## ✧ Banking Website

- What features do we want to see?
- Account creation, deletion, manipulation.
- Fund transfers
- Fund withdrawal
- Check deposit
- ...?



The screenshot shows a web browser window with the address bar displaying 'demo.guru99.com/V4/'. The page has an orange header bar with the text 'Guru99 Bank'. Below the header, there is a login form with two input fields: 'UserID' containing the text 'mg9104' and 'Password' containing masked characters '\*\*\*\*\*'. Below the password field are two buttons labeled 'LOGIN' and 'RESET'.

# Develop the Test Strategy

---

## ✧ Document defining:

- Test Objectives (and how to achieve them)
- Testing Effort and Cost



# Testing Scope

---

## ✧ What are you planning to test?

- Software, hardware, middleware, ...

## ✧ ... AND... What are you **NOT** going to test?

- Gives project members a clear understanding about what you are responsible for.

## ✧ Must take into account:

- Requirements, budget, skills of your testing team

# Testing Scope

---

## ✧ Banking website

- Requirements specified only for functionality and the external interface.
  - **These are in-scope.**
- No requirements for database or client hardware.
- No quality requirements (performance, availability).
  - **These are out-of-scope.**

# Identify Testing Types

---

✧ Which should we apply?

- Consider the project domain.

✧ Which can we skip or limit to save money?

## **For the banking site:**

- System Testing

- Focus on verifying access points and interfaces.
- Functionality likely spread over multiple classes, many features interact

- Exploratory Testing

### **Could limit:**

- Unit Testing (focus on integration/interfaces over individual classes)

### **Can skip:**

- Acceptance Testing

# Create Test Logistics

---

## ✧ Who will write and execute test cases?

- What types of testers do you need?
  - Skills needed for the targeted domain
- What is the budget for testing?
  - How many people can you hire to test?

## ✧ When will each testing activity occur?

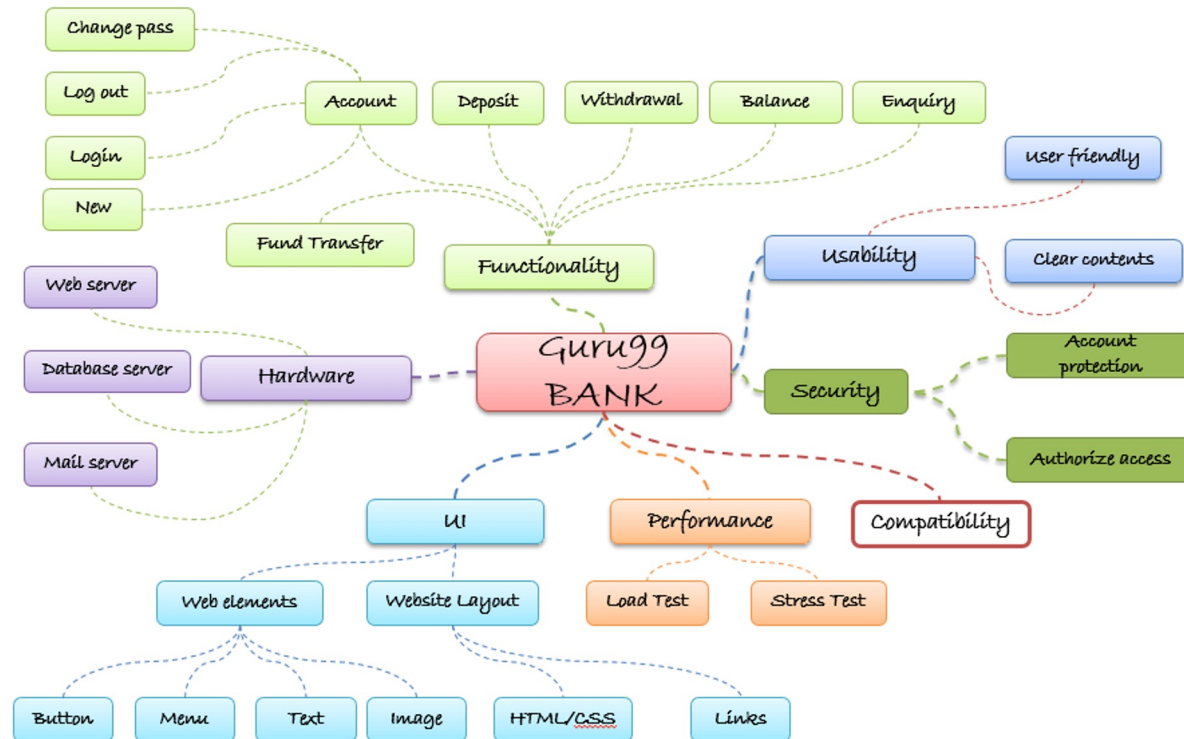
- When to design and when to execute tests.
- Pair with appropriate stage of development.
  - Unit development -> unit testing -> system testing -> ...

# Define Test Objectives

---

- ✧ What are the goals of the testing process?
  - What features, system elements need to be tested?
  - What quality attributes do we need to demonstrate?
  - For each feature or quality, what scenarios do we want to walk through?
- ✧ Does not include a list of specific tests
  - But, at a high level, should detail scenarios we plan to examine by writing one or more test cases.

# Define Test Objectives





# Define Test Criteria

---

✧ When have we completed our testing objectives?

- For qualities, set appropriate thresholds.
  - Availability, ROCOF, throughput, etc.
- For functionality, commonly defined using:
  - **Run Rate: Number of Tests Executed / Number Specified**
  - **Pass Rate: Number of Passing Tests / Number Executed**
  - Often aim for 100% run rate and a high pass rate (> 95%)

# Resource Planning

---

- ✧ Summarize resources that you have.
  - Allows estimation and adjustment of testing scope, objectives, and exit criteria.
- ✧ Human Resources: Managers, testers, developers who assist in testing, system administration.
- ✧ System Resources: Servers, testing tools, network resources, physical hardware.

# Plan Test Environment

---

## ✧ Where will you execute test cases?

- Software and hardware execution environment
- Often defined as part of continuous integration.

## ✧ Need to account for:

- Requirements on both server and client-side.
- Different networking conditions (bandwidth, load).
- Different client or server-side hardware.
- Different numbers of concurrent users.

# Schedule Estimation

---

- ✧ Break testing plans into individual tasks, each with an effort estimation (in person-hours)
  - Create test specification, 170 person-hours
  - Write unit tests, 80 person-hours
  - Write API tests, 50 person-hours
  - Perform test execution, 1 person-hour (per suite execution)
  - Write test report, 10 person-hours
  - ...

# We Have Learned

---

- ✧ What is testing?
- ✧ Testing terminology and definitions.
  - Input, oracles
  - Faults, failures
- ✧ Testing stages include unit testing, system testing, exploratory/GUI testing, and acceptance testing.
- ✧ Test planning needs to consider resources, time, scope, environment.

