Testing Fundamentals

CSE 4495- Lecture 3 - 28/06/2022

Instructor : Md. Mohaiminul Islam

Verification

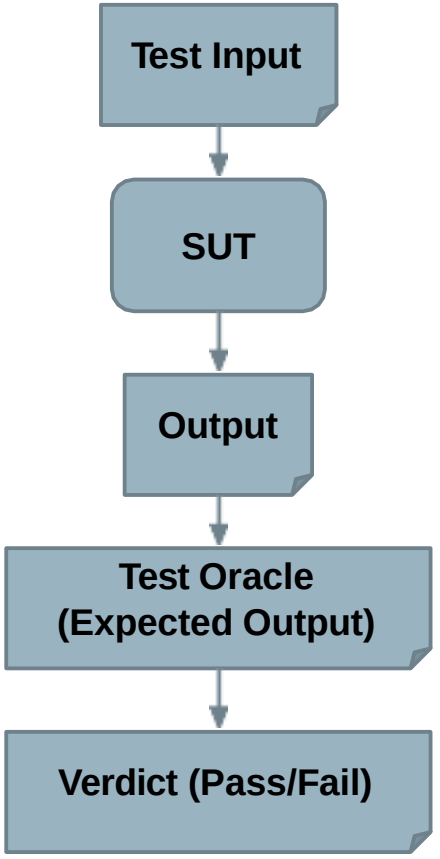
- Ensuring that an implementation conforms to its specification.
  - AKA: Under these conditions, does the software work?
- Proper V&V produces dependable software.
  - **Testing is the primary verification activity.**

We Will Cover

- What is testing?
- Definitions:
  - What are the components of a test case?
- Testing stages:
  - Unit, System (Integration and Exploratory), and Acceptance Testing
- Test planning considerations

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*.**

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

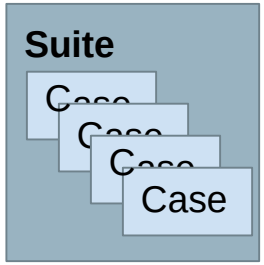
“Program testing can be used to show the presence of bugs, but **never their absence**.”

- Dijkstra

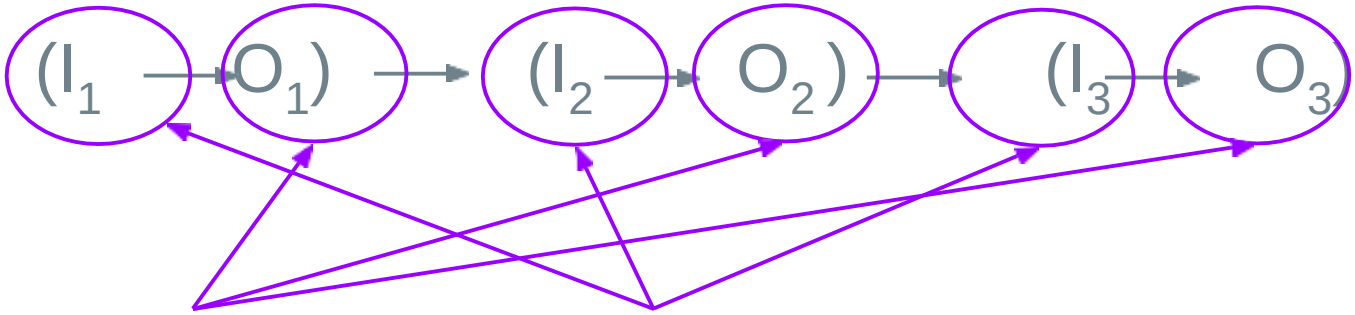
What Goes in a Test Case?

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



if O_n = Expected(O_n)

then…Pass
else… Fail

Test Inputs

How we “stimulate” the system.

Test Oracle

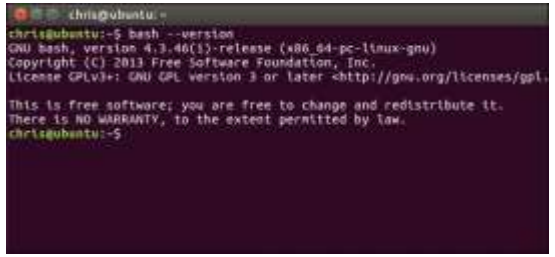
How we check the correctness of the resulting observation.

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, calls a function through an interface.**
  - Method Call
  - API Call
  - CLI Interaction
  - GUI Interaction



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.
    - `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

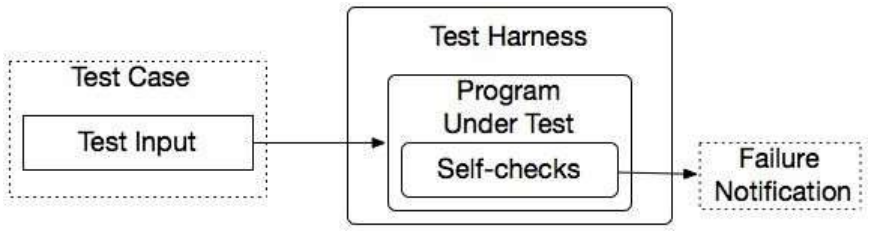
- Simplest oracle - what exactly should happen?

```
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.

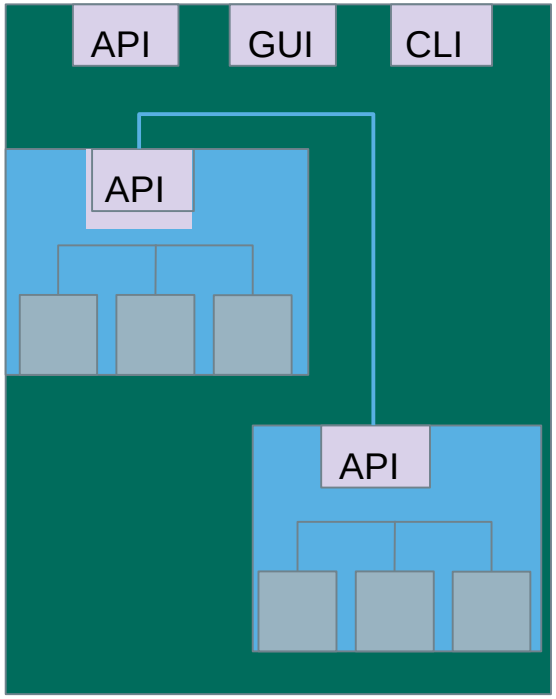
Implicit Oracles

- 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

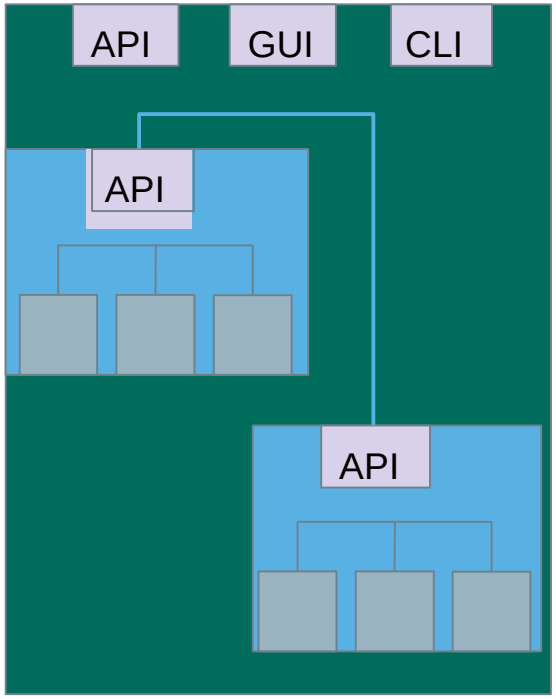
Testing Stages

- 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?

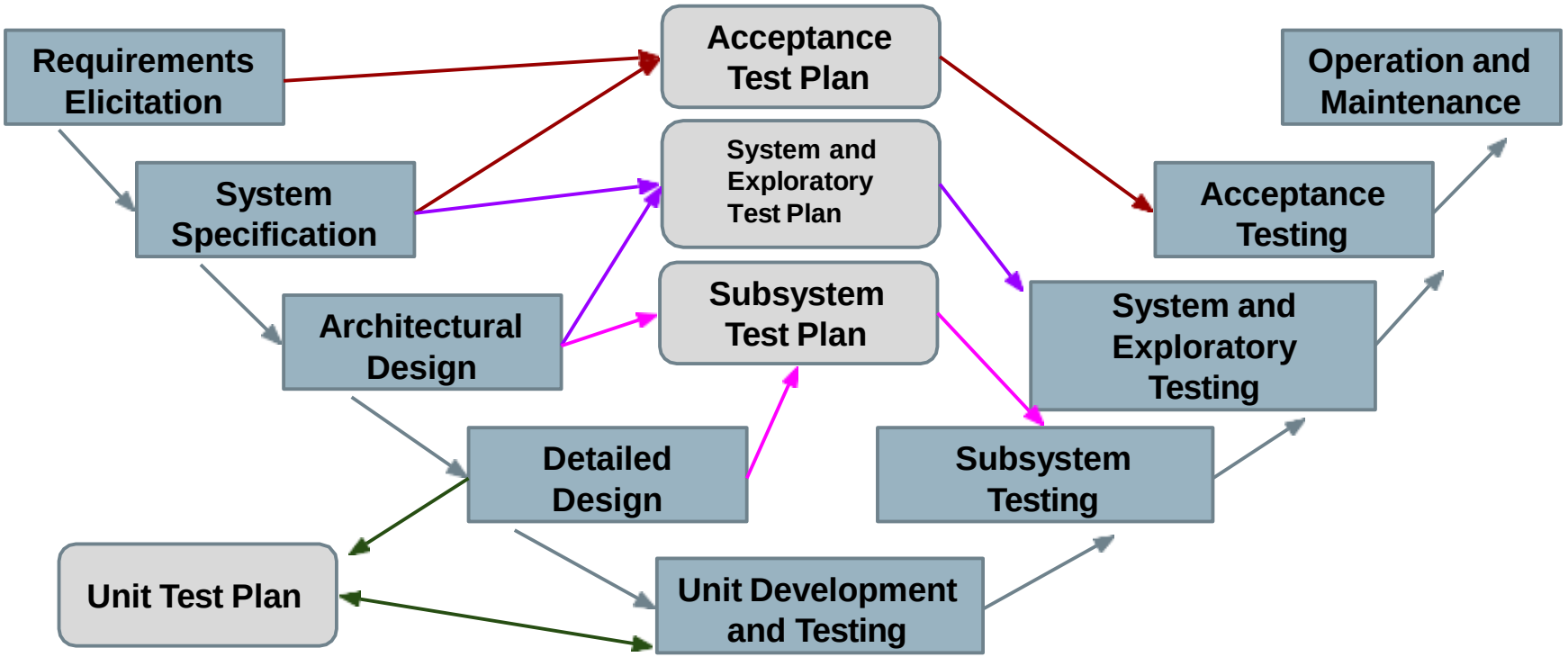


Testing Stages

- **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.

Let’s take a break.

The V-Model of Development

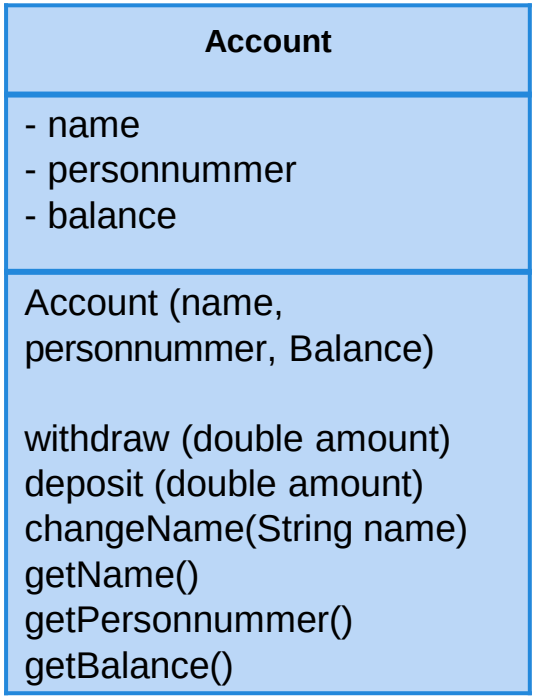


Unit Testing

- 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).



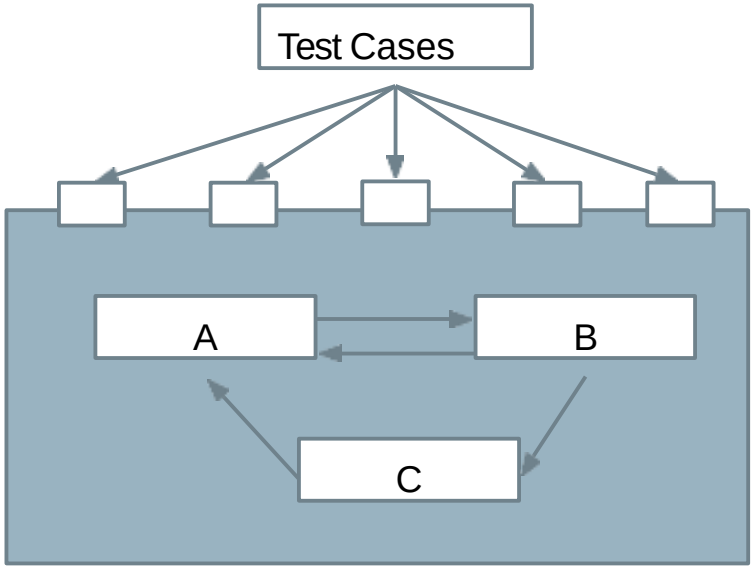
System (Integration) Testing

- 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.



GUI Testing

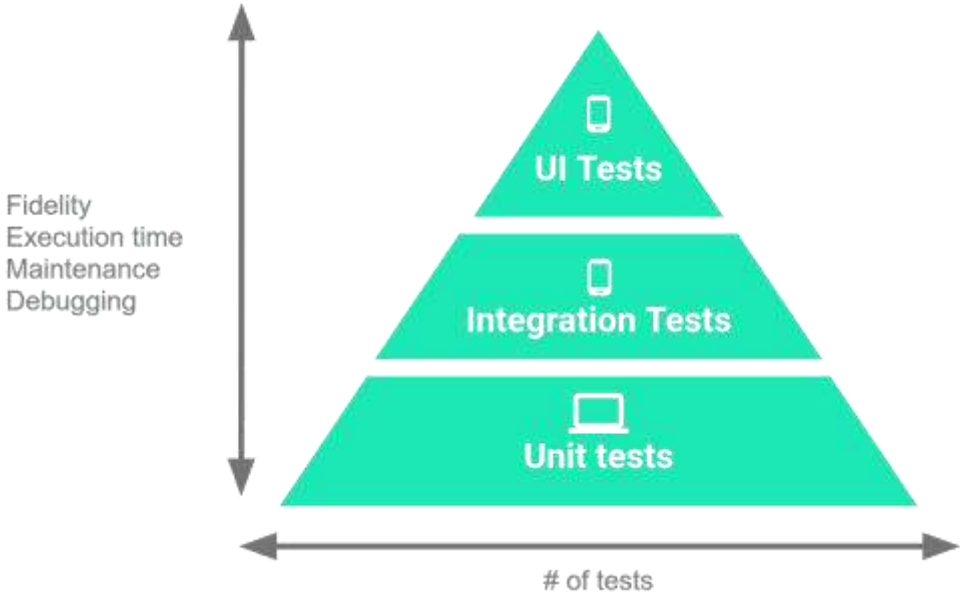
- Tests designed to reflect **end-to-end** user journeys.
  - From opening to closing.
  - Often based on **scenarios**.
- GUI Testing
  - Deliberate tests, specific input.
  - May be automated or human-executed.
- Exploratory Testing
  - Open-ended, human-driven exploration.

Exploratory Testing

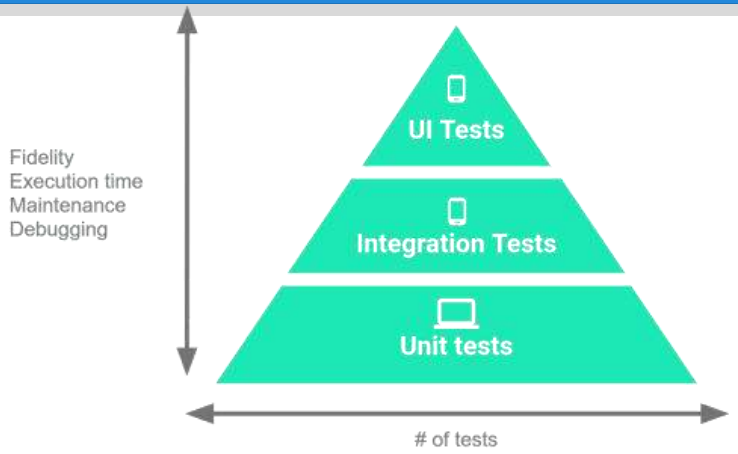
- Tests are not created in advance.
- Testers check the system on-the-fly.
  - Guided by scenarios.
  - Often based on ideas noted before beginning.
- Testing as a thinking idea.
  - About discovery, investigation, and role-playing.
  - Tests end-to-end journeys through app.
  - Test design and execution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

Testing Percentages

- Unit tests verify behavior of a single class.
  - 70% of your tests.
- System tests verify class interactions.
  - 20% of your tests.
- GUI/exploratory 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 only emerge 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.

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.

Next Time

- Next lecture: System Testing
  - Optional reading: Pezze and Young, Ch 10-11

Thank You