To Test or Not To Test

Marian Edu

AcornIT

PUG CHALLENGE

Boston, 2024

Testing

* noun - /ˈtestɪŋ/

the process of using or trying something to see if it works, is suitable, obeys the rules, etc.

* Cambridge Dictionary

Software Testing

Software testing is the process of evaluating and verifying that a software product or application does what it's supposed to do.

Over time several types of software testing emerged setting the stage for a broader view of testing, which encompassed a quality assurance process that became an integrated part of the software development lifecycle.

Types of Software Testing

- Unit testing: Validating that each software unit runs as expected. A unit is the smallest testable component of an application.
- Integration testing: Ensuring that software components or functions operate together.
- Acceptance testing: Verifying whether the whole system works as intended.

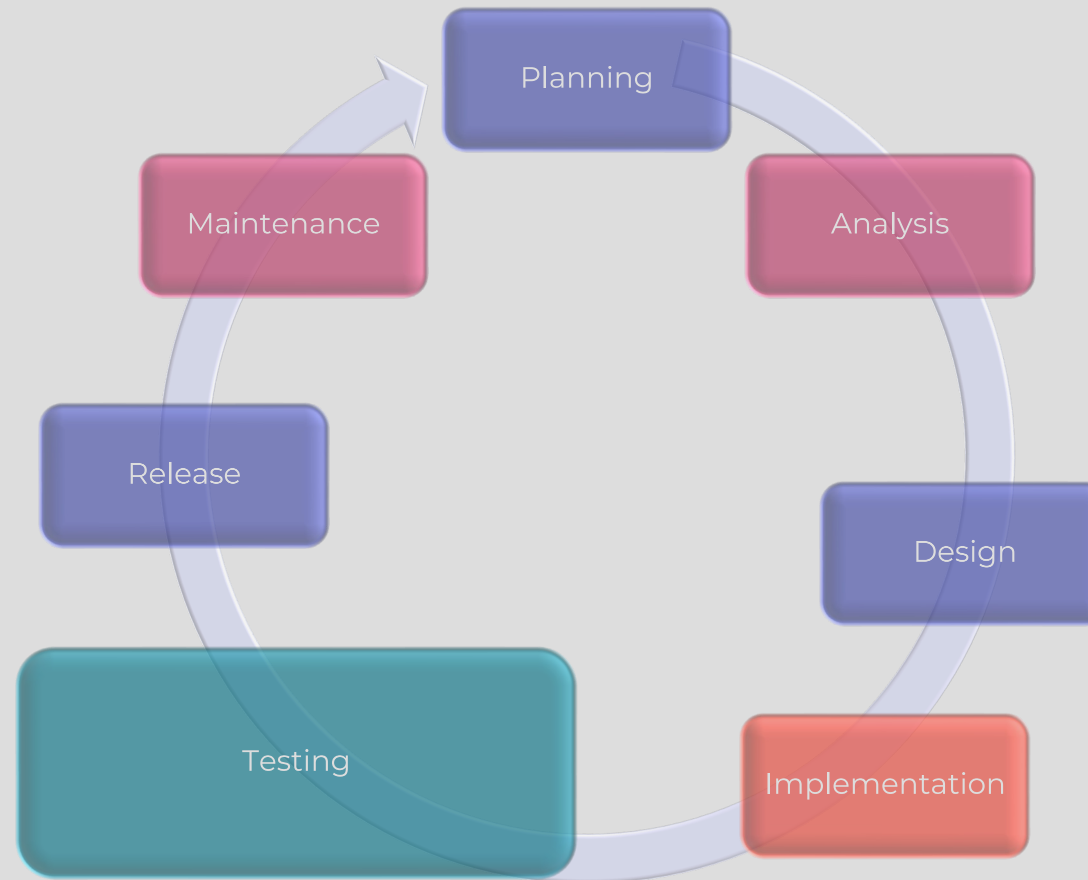
Types of Software Testing

- Functional testing: Checking functions by emulating business scenarios (black-box).
- Regression testing: Checking whether new features break or degrade functionality.
- Usability testing: Validating how easy a user can complete a task using the application.

Types of Software Testing

- Performance testing: Testing how the software runs under different workloads.
- Stress testing: Testing how much strain the system can take before it fails.
- Security testing: Validating that your software is not open to hackers or other malicious types of vulnerabilities

Software Development Lifecycle



DevOps & Agile

- Development & Operations
- ArchOps: Software architecture artifacts – models, first-class entities.
- Continuous Integration and Delivery (CI/CD): All about automation basically.

Why Testing

Yes, it does take time and costs money.

Can help to:

- uncover problems before going to market
- avoid defects, even late delivery
- protect brand reputation

Software failures in the US cost the economy USD 1.1 trillion in assets in 2016.

What's more, they impacted 4.4 billion customers.

Why Testing

- In April 2015, Bloomberg terminal in London crashed due to software glitch affected more than 300,000 traders on financial markets.
- Nissan cars recalled over 1 million cars from the market due to software failure in the airbag sensory detectors.
- In April of 1999, a software bug caused the failure of a $1.2 billion military satellite launch, the costliest accident in history.
- Fujitsu software bugs “helped” send innocent postal employees to prison in the UK.
- Crowdstrike, a software update causing a major outage in July 2024 (8.5 million devices were affected).

Testing Approaches

- Manual testing or ad hoc testing might be enough for small builds.
- Larger systems, frequently require tools used to automate tasks.
- Continuous testing.
  - Defect and Bug Tracking (cause)
  - Configuration management (what)
  - Testing environment (where)
  - Service virtualisation
  - Metrics and reporting

Unit Testing

- Test the smallest functional unit of code.
- Helps ensure code quality.
- It's an integral part of software development.
- Writing software as small, functional units is considered a best practice – so we can write a unit test for each code unit.
- Smallest unit of code: method, function, procedure.

Unit Testing

- The unit test needs to run in isolation.
- The code unit must be idempotent.
- Use mock-up/data stubs when unit of code access external data.
- A code unit can have a set of unit tests – test cases.

Unit Testing Strategies

- Write unit tests as code.
- Logic check.
- Boundary check.
- Error handling.
- State check.

Unit Testing Best Practices

- Use a unit test framework – ABL Unit, OE Unit, Pro Unit.
- Automate unit testing.
- Assert once.
- Keep it simple.
- Implement unit testing as part of development process.

ABL Unit – Test case

- Write a test case per each unit of code.
- Write separate test method for each scenario tested.
- Don’t bother to test all valid input, one will do along with boundaries and invalid.
- Use (some) naming convention.

ABL Unit – Test Case Annotations

- @Test [(expected = “ExpectedErrorType”)]
- @Before – once per class, before all tests
- @Setup – before each test
- @TearDown – after each test
- @After – once per class, after all tests

ABL Unit – Test case flavours

- Test class:
  - The class needs the default constructor, if defined must be public.
  - All tests are public (void) methods with no parameters.
  - Inheritance doesn’t “work”.

- Test procedure.
  - All tests are (nonprivate) internal procedures with no parameters.
  - Annotated functions are ignored.

ABL Unit – Test Case Assertions

- OpenEdge.Core.Assert
- Equals: Expected vs. Actual
- Argument name
- OpenEdge.Core.AssertionFailedError
- Assert:RaiseError

Equals (character, character)

S

Parameters:

a	CHARACTER
b	CHARACTER

IsFalse (logical, character)

S

Parameters:

plArgument	LOGICAL
pcName	CHARACTER

ABL Unit – Test suite

- Group related test cases.
- Test cases for unit of codes of the same object.
- Can be ran as regression test when covered functionality changes.
- Order of test cases should not matter.

ABL Unit – Test Suite Annotations

- @TestSuite (classes = “TestClass (, TestClass)*”)

- @TestSuite (procedures = “TestProcedure (, TestProcedure)*”)

- Annotations can have both parameters set (classes & procedures).
- You can mix annotations in the same suite (classes & procedures).
- There is a limit of characters for annotation’s parameters.
- You can use multiple annotations in the same suite.

ABL Unit – Test suite flavours

- Test Suite class:
  - The class constructor is ignored.

- Test Suite procedure.
  - Procedure main block is ignored.

- Only @TestSuite annotations are read.
- Supports both classes and procedures annotations.
- Classes, procedures not found makes the whole suite to fail.
- Classes, procedures that aren’t test cases are ignored.

ABL Unit – Error Handling

- Use block-level/routine-level.

- Don't bother to use NO-ERROR/CATCH in @Test methods.

- Catch errors in @Before (All) method.

- Errors in @Setup will mark all @Tests as error.
- Errors in @TearDown will mark all @Tests as error.
- Errors in @After will have no effect on the test result.

ABL Unit – Global State

- Avoid global state in test cases - unless complex to setup.
- If you need global state only alter that in @Before, read-only afterwards.
- If the @Test methods do update global state, make sure it is restored back.
- Do consider session global scope – session handle, statics.

ABL Unit – Persistence

- Persistence (databases) are to be treated as external systems.
- When database access is required, use data stubs.
- How/when you restore back the state.
- End (clean-up) vs. Beginning (zero-trust) of the @Test.
- Using @Test scoped transactions and rollback.
- Reload/restore database.

ABL Unit – External Systems

When external systems data is required, use mocks.

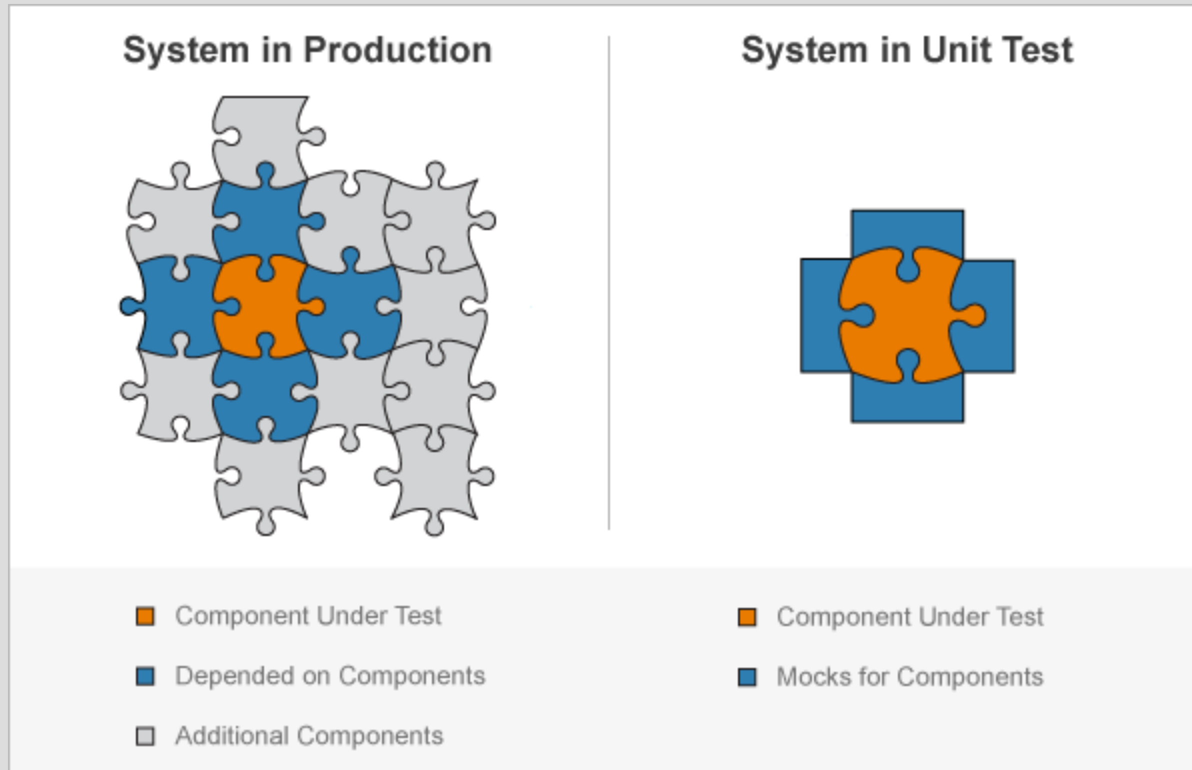


Image source: MathWorks

ABL Unit – Mocking

- @Setup expectations in mock
- @Test functionality
- Verify expectations
  - The state is correctly updated
  - The right methods have been called in mock.

- No mocking framework for Progress/OpenEdge

ABL Unit – Automation

- ABLUnit ANT task (Progress or PCT)
- Single test/batch test (include/exclude)
- haltOnFailure/haltOnError
- UI mode/Batch mode
- Jenkins/Docker

Wrap up

- To Test Or Not To Test (Shakespeare’s Hamlet or Schrödinger's cat)
- Keep on Testing in Production – Not an option!
- The sooner you start the better.
- Start with unit testing - new code, include it in development/maintenance process.
- Unit testing alone will help find issues earlier.
- As with anything doesn’t have to be perfect.
- Run regression tests before rollout.