# Systems III Test II

## Coding Standards

---

Proper standards have rules encompassing all aspects of code. Standards should be verified in a code inspection. All source code should look as if it was written by one developer in one session.

### Motivation

Maintenance is 40% - 80% of the lifetime of a piece of software. Most software is **not** maintained by the original author. Code conventions allow the developer to understand and parse the code quicker, and more thoroughly.

- Personal preference shouldn't be imposed on others code.
- Arguments over style are ***pointless***.
- Write code for maximum clarity, not personal preference.

### New Projects

Coding standards should be established at the **inception** of a software project.

### Existing System

When incorperating **existing** code or while in **maintenance**, the coding standard should state how to deal with existing code.

## Categories of Standards

### Names

- Variables names should tell *what*, not *how*.
- Avoid names that expose implementation, which can change.
- For example: Use `GetNextStudent()`, not `GetNextArrayElm()`.
- Should be long enough to be meaningful, while not being wordy.
- Expressive names.
- Should contain standards for:
    - Classes
    - Methods
    - Variables
    - Tables
    - File/Directory names, Packages

### Comments

- Start of file comments
- Class and Method comments
- In line comments

### Format

Formatting makes the logical organization of the code stand out. Must be consistent.

- Indentation
- Mixed case usage
- Open/Close braces
- Line length
- White space

#### Spelling

- US vs. ***PROPER*** Spelling
    - `color` vs. **colour**.
- Case
    - Upper
    - lower
    - Camel
        - UpperCamel
        - lowerCamel

### Principle of Least Astonishment - POLA

"*If a necessary feature has a hight astonishment factor, it may be necessary to redesign the feature.*"

## Intro to Software Testing

---

### Bugs

*What is a Bug?*

- The software doesn't do something that the specification says it **should** do.
- The software does something that the specification says it **shouldn't** do.
- The software does something the product specification doesn't **mention**.

Main cause is the **specification**.

- May not exist

- Not thorough
- Constantly changing
- Not communicated

### Terms

#### Most Generic

- Problem
- Error
- Bug
- `glitch`

#### Most Severe

- Defect
- Fault
- Failure

#### Unintended

- Incident
- Anomaly
- Variance
- Inconsistency
- "*Feature*"

## Testing in Agile

"*Test early; Test often*"

- Test Driven Development (TDD)
- Tests are written first (based on acceptance criteria)
- Testing is performed all the way during the SDLC.
- Testers are involved from the start

### Testing Considerations

- **Risk-based**
- Cannot show that there are no bugs
- The more bugs found, the more bugs there are
- "*Pesticide Paradox*" - code becomes **immune** to testing
- Not every bug can be fixed

### Terms

#### Verification vs. Validation

- Verification - process confirming that software meets its **specification**.
- Validation - process confirming that software meets **user requirements**.

Something can meet specification (verification), but not original requirements (validation), or vice-versa.

#### Quality vs. Reliability

- Quality - Features, speed, etc.
- Reliability - One aspect of quality - how often a system crashes.

# Test Levels; Types of Testing

---

## Static and Dynamic Testing

- Static Testing
    - Testing by **examining** and reviewing.
    - e.g. code inspections
- Dynamic Testing
    - Testing by **running** and using the software.

## Black Box vs. White Box

- Black Box Testing
    - Tester knows what software is supposed to do, but cannot look inside the box to see how it **operates**.
- White Box Testing
    - Tester has access to the code and can examine in for **clues** to help with testing.
    - Tester may determine that certain conditions are more or less likely to fail, and tailor teting based on that.

#### Static White Box Testing

- Methodically reviewing software design, architecture, or code without running it.
- aka **structural analysis***.
- Increases chances of finding an error, or early defect.

## Test Level #1: Unit Testing

- Validate each unit of software works as designed.
    - A unit is the **smallest** testable part of software.
- **White** box testing
- Made as unit is developed.
- Performed by the developer.

## Test Level #2: Integration Testing

- Expose faults in the **interactions** between units.
- **White** or **Black** box testing
- After unit tests, before system tests.
- Developers or Testers

#### Approaches

- Big Bang - All units are combined and testing **at once**.
- Top Down - Top level units are tested first, and lower level units are tested after.
    - Advantageous if major flaws occur towards the top.
    - Early skeletal Program allows demonstrations and boosts morale.
    - Stub modules must be produces
    - Stubs can be complicated
    - Representation of test cases in stubs can be difficult
    - Conditions may be hard to create
- 
    - Observation of output is more difficult
- Bottom Up - Bottom levels are tested first, and upper level units are tested after.
    - Advantageous if major flaws occur towards the bottom.
    - Test conditions are easier to create
    - Observation of test results is easier
    - Drivers must be produced
    - The program does not exist until last module is added
- Sandwich/Hybrid - **Combination** of the previous two.

## Test Level #3: System Testing

- Evaluate system's compliance with **specifications**.
- **Black** box testing
- After integration testing and before acceptance testing
- **Independent** testers

## Test Level #4: Acceptance Testing

- Evaluate system's compliance with **business** requirements
- **Black** box testing
- After system testing
- **End users** or **customers**.

## Types of Testing

#### Regression Testing

- **Retesting** of previously tested programs following changes.
- Ensures no faults have been introduced.

- Automated

**Exploratory Testing**

- Interact in whatever way the want to force functionality.
- ***CHAOS MONKEY***.
- Test design and execution at the **same time**
- Popular in Agile
- Testers risk wasting time wandering around
- Very ad-hoc

**User Interface Testing**

- Self-explanatory

**Usability Testing**

1. Accessibility
2. Responsiveness
3. Efficiency
4. Comprehensibility

**Security Testing**

- Self-explanatory

**Interoperability Testing**

- Refers to the diverse systems ability to work together.
- End-to-end functionality.

**Compatibility Testing**

- System needs to work with other applications
- Different operating systems, browsers, hardware, etc.

**Reliability Testing**

- "*How long will it run before it breaks?*"

**Scalability Testing**

- Can the DB scale?
- Network a bottleneck?
- Hardware requirements?

**Performance/Stress Testing**

- "*My web application got front page on Reddit/Hacker News/ Product Hunt and got DDoS'd!*"

# Black Box Testing

---

### Static Black Box Testing

- Testing the specification
- Perform a high-level review of the specification
- Pretend to be the customer
- Research existing standards
- Review and test similar software

### Dynamic Black Box Testing

- Testing the Software
    - No details of **code**.
    - Focuses on meeting **functional** requirements
    - Needs **definition**
    - Can be **exploratory**.

### Black Box Test Strategies

#### Equivalence Partitioning

Partition tests into a finite number of sets of test inputs.

For example, Boolean needs a test for both `true`, and `false`.

#### Boundary Value Analysis (BVA)

#### *EDGE CASES*

#### Use Cases

Identify paths in the user flow, test each path.

## White Box Testing

---

### Code Coverage and Code Coverage Analysis

*Code coverage* is how many lines/blocks/arcs of code are tested during automated testing.

*Code Coverage Analysis* is the process of analyzing code, in hopes in creating meaningful and thorough test cases.

Good at:

- Finding areas not exercised by test cases
- Creating additional test cases
- Determining a quantitative measure of code coverage, and quality
- Catch redundant tests that do not increase coverage

Assumes:

- Bugs relate to flow, and you can expose bugs by changing the flow.
- Tester understands what a correct version of the program does
- Specifications, no errors of omission, no dead code

## Basic Metrics

### Statement Coverage

- Covers each LoC
- aka *line coverage*
- Does not care about logical operations, just covers each line once.
- Only covers `true` conditions

### Decision Coverage

- Identifies each decision that can be `true`/`false`.
    - `do-while`
    - `if/else`
    - `switch/case`
- Makes sure every condition has been met
- Different from statement coverage, looks at false conditions

### Condition Coverage

- Identifies each decision that can be `true`/`false`.
- Makes sure every condition has been met
- Different from decision coverage, as it includes alternate paths to same conditions

### Path Coverage

- Identifies every possible path in a program
- Has test cases for each path, not just each condition
- Different from condition coverage, as it covers paths, not conditions

## Cyclomatic Complexity

- Metric, measures complexity of a program
- Higher means more risk of errors

Make a Flow Graph of the system, calculate:
`CC = E - N + 2 * P`

where

`CC` is the Cyclomatic Complexity

`E` is the number of edges in the graph (lines)

N is the number of nodes (shapes)

P is the number of exit points

# Test Planning & Defect Management

---

## Test Case Design

- A test case tests one **particular** situation
- System test case: **end-to-end**, complete flow

## Requirements Traceability

- Need to be able to follow **chain** from requirements
- Need to show that you have **tested** every requirement.

## Test Case Format Example

| Test Case | Input Data | Calculation | Expected Result | Checked |
|---|---|---|---|---|
| 1 | 2, 2 | Addition | 2 | Y |
| 2 | 11, 1 | Multiplication | 11 | Y |
| 3 | 5, 3 | Subtraction | 2 | N |

## Test Execution

- Need someone else to test the code you write
- Share testing responsibilities
    - "***Bug Bash***"
    - Beta Testing
    - Outsourcing Testing
- Use whatever means possible

## Defect Logging and Tracking

- Report bugs as soon as possible
- Effectively describe the bugs
- Be **nonjudgmental** when reporting bugs
- Follow up on bug reports

### Effective Bug Description

- Unique **identifier**
- Steps to **reproduce**
- What you **expected** to see
- What you **actually** saw

### Isolating and Reproducing

- Narrow down the cause
- Log even if you cannot reproduce

**Severity/Priority**

- How bad the bug is, how it **impacts** the product
    - Critical: Will not run
    - High: System crash, fatal errors
    - Medium: Malfunction in a feature
    - Low: A E T H E T I C S M A C H I N E B R O K E
- Usually a number from 1 - 4
- Prority is how **important** the bug is to fix
    - Immediate
    - Before Release
    - If time permits
    - Post-release
- Usually a number form 1 - 4