UNIT TESTING

CS4036

Lecture #4

Software Testing

Concept of Unit Testing

- Unit-Module
- all possible—or as much as possible—distinct executions are to be considered during unit testing
- A programmer will need to verify whether or not a code works correctly by performing unit-level testing. Intuitively, a programmer needs to test a unit as follows:
  - • Execute every line of code. This is desirable because the programmer needs to know what happens when a line of code is executed. In the absence of such basic observations, surprises at a later stage can be expensive.
  - • Execute every predicate in the unit to evaluate them to true and false separately.
  - • Observe that the unit performs its intended function and ensure that it contains no known errors.

Concept of Unit Testing

- Not everything pertinent to a unit can be tested in isolation because of the limitations of testing in isolation.
- The source code of a unit is not used for interfacing by other group members until the programmer completes unit testing and checks in the unit to the version control system.

Concept of Unit Testing

- Static Unit Testing
  - Code is examined over all possible behaviors that might arise during run time
  - Code of each unit is validated against requirements of the unit by reviewing the code
- Dynamic Unit Testing
  - A program unit is actually executed and its outcomes are observed
  - One observe some representative program behavior, and reach conclusion about the quality of the system
- Static unit testing is not an alternative to dynamic unit testing
- Static and Dynamic analysis are complementary in nature
- In practice, partial dynamic unit testing is performed concurrently with static unit testing
- It is recommended that static unit testing be performed prior to the dynamic unit testing

Static Unit Testing

- In static unit testing code is reviewed by applying techniques:
  - **Inspection:** It is a step by step peer group review of a work product, with each step checked against pre-determined criteria
  - **Walkthrough:** It is review where the author leads the team through a manual or simulated executed of the product using pre-defined scenarios
- The idea here is to examine source code in detail in a systematic manner
- The objective of code review is to *review* the code, and *not* to evaluate the author of the code
- Code review must be planned and managed in a professional manner
- The key to the success of code is to divide and conquer
  - An examiner inspect small parts of the unit in isolation
    - nothing is overlooked
    - the correctness of all examined parts of the module implies the correctness of the whole module

Static Unit Testing (Code Review)

- Step 1: **Readiness**
  - **Criteria**
    - **Completeness**
    - **Minimal functionality**
    - **Readability**
    - **Complexity**
    - **Requirements and design documents**
  - **Roles**
    - **Moderator**
    - **Author**
    - **Presenter**
    - **Record keeper**
    - **Reviewers**
    - **Observer**
      - **Reviewing is not reporting**
- Step 2: **Preparation**
  - **List of questions**
  - **Potential Change Request (CR) – Not defects**
  - **Suggested improvement opportunities**

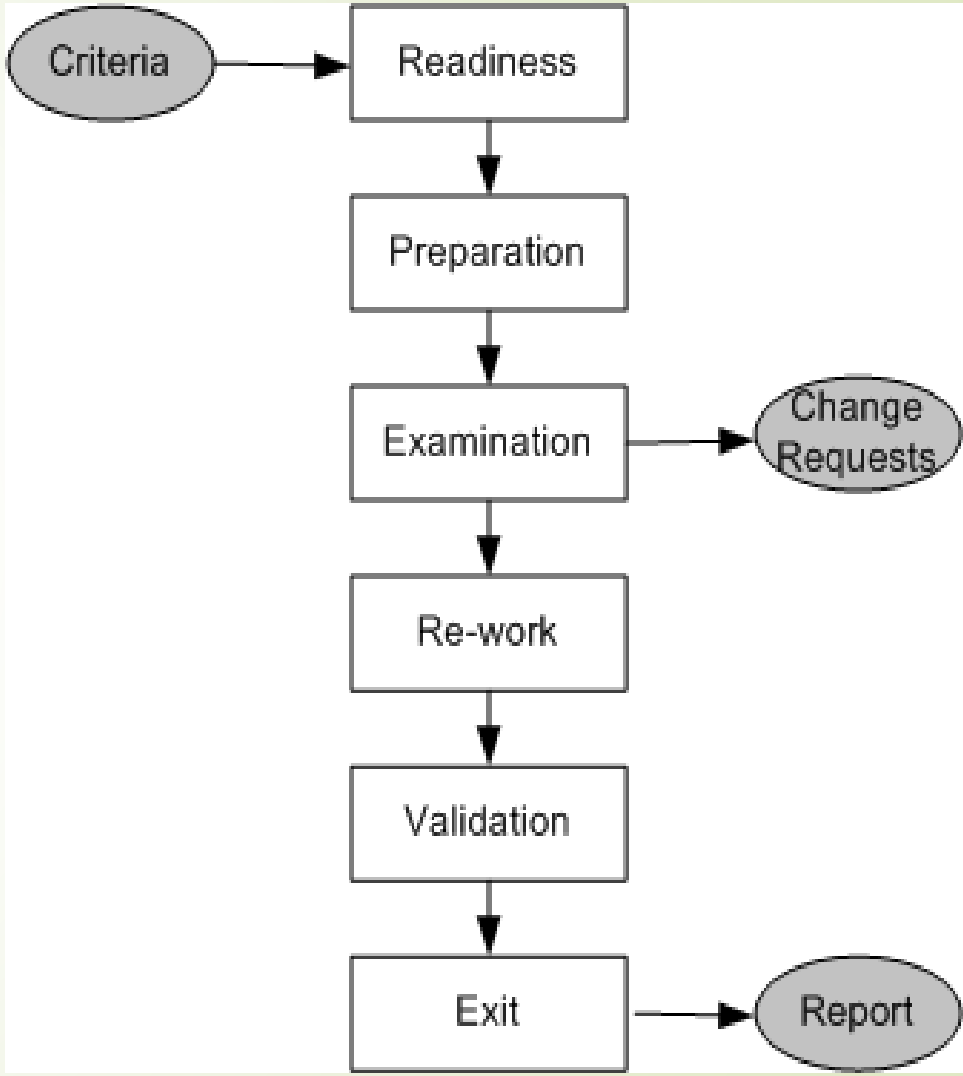


Figure 3.1: Steps in the code review process

Static Unit Testing (Code Review)

- Step 3: **Examination**
  - The author makes a presentation
  - The presenter reads the code
  - The record keeper documents the CR
  - Moderator ensures the review is on track
- Step 4: **Re-work**
  - Make the list of all the CRs
  - Make a list of improvements
  - Record the minutes meeting
  - Author works on the CRs to fix the issue
- Step 5: **Validation**
  - CRs are independently validated
- Step 6: **Exit**
  - A summary report of the meeting minutes is distributes

A **Change Request (CR)** includes the following details:

- Give a brief description of the issue
- Assign a priority level (major or minor) to a **CR**
- Assign a person to follow it up
- Set a deadline for addressing a **CR**

Static Unit Testing (Code Review)

The following metrics can be collected from a code review:

- The number of lines of code (LOC) reviewed per hour
- The number of CRs generated per thousand lines of code (KLOC)
- The number of CRs generated per hour
- The total number of hours spend on code review process

Static Unit Testing (Code Review)

- The code review methodology can be applicable to review other documents
- Five different types of system documents are generated by engineering department
  - Requirement
  - Functional Specification
  - High-level Design
  - Low-level Design
  - code
- In addition installation, user, and trouble shooting guides are developed by technical documentation group

Hierarchy of System Documents
Requirement: High-level marketing or product proposal.
Functional Specification: Software Engineering response to the marketing proposal.
High-Level Design: Overall system architecture.
Low-Level Design: Detailed specification of the modules within the architecture.
Programming: Coding of the modules

DEFECTPREVENTION

- It is in the best interest of the programmers in particular and the company in general to reduce the number of CRs generated during code review.
- This is because CRs are indications of potential problems in the code, and those problems must be resolved before different program units are integrated.
- Addressing CRs means spending more resources and potentially delaying the project.
- Therefore, it is essential to adopt the concept of defect prevention during code development.

Dynamic Unit Testing

- The environment of a unit is emulated and tested in isolation
- The caller unit is known as *test driver*
  - A *test driver* is a program that invokes the unit under test (UUT)
  - It provides input data to unit under test and report the test result
- The emulation of the units called by the UUT are called *stubs*
  - It is a dummy program
- The *test driver* and the *stubs* are together called *scaffolding*
- The low-level design document provides guidance for selection of input test data

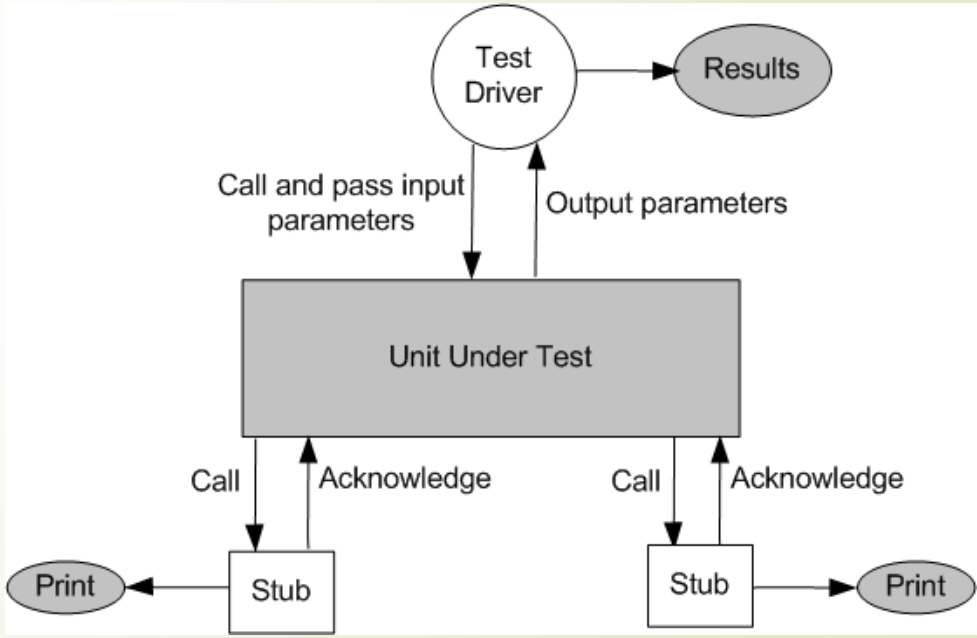


Figure 3.2: Dynamic unit test environment

Dynamic Unit Testing

Selection of test data is broadly based on the following techniques:

- Control flow testing
  - Draw a control flow graph (CFG) from a program unit
  - Select a few control flow testing criteria
  - Identify a path in the CFG to satisfy the selection criteria
  - Derive the path predicate expression from the selection paths
  - By solving the path predicate expression for a path, one can generate the data
- Data flow testing
  - Draw a data flow graph (DFG) from a program unit and then follow the procedure described in control flow testing.
- Domain testing
  - Domain errors are defined and then test data are selected to catch those faults
- Functional program testing
  - Input/output domains are defined to compute the input values that will cause the unit to produce expected output values

Mutation Testing: A mutation of a program is a modification of the program created by introducing a single, small, legal syntactic change in the code. A modified program so obtained is called a mutant.

Debugging:

Brut Force

Cause elimination

Backtracking

Mutation Testing

Mutation testing is often called **"Testing the Testers."** While most testing techniques look for bugs in your code, mutation testing looks for "bugs" (weaknesses) in your **test cases**.

Mutation testing is not a testing strategy like control flow or data flow testing. It should be used to supplement traditional unit testing techniques.

A mutation of a program is a modification of the program created by introducing a single, small, legal syntactic change in the code.

A modified program so obtained is called a mutant.

Mutation Testing: Workflow

Phase 1: The Healthy Baseline

You start with your original, correct code and a set of test cases that all pass.

Code: if (x > 0) return true;

Test: assert(isPositive(5) == true); (Status: **PASS**)

Phase 2: Creating the "Zombies" (Mutants)

A mutation tool automatically creates multiple versions of your code, each with exactly **one** small change (a "mutation"). Each version is called a **Mutant**.

Mutant 1: if (x >= 0) return true; (Relational operator change)

Mutant 2: if (x < 0) return true; (Logic reversal)

Mutant 3: if (x > 1) return true; (Value change)

Mutation Testing: Workflow

Phase 3: The Battle

You run your original test cases against every single mutant.

Phase 4: The Outcomes

Killed (Caught): If a test fails when running against a mutant, the mutant is "killed." This is **good**! It means your test was strong enough to notice the change.

Survived (Lived): If all your tests still pass despite the code change, the mutant "survived." This is **bad**. it means your tests are weak and missed a potential bug.

Mutation Testing: **Programming Example**


```
def can_vote(age):
    if age >= 18:
        return "Yes"
    else:
        return "No"
```

Your Current Test Suite:
Test A: can_vote(20) → Expected: "Yes"
Test B: can_vote(10) → Expected: "No"
The Mutation Attack:
The tool creates a mutant by changing the boundary operator:
Mutant 1: if age > 18: (Changed >= to >)
Running the Tests against Mutant 1:
Test A (20): Result is "Yes". (Expected "Yes"). **Passed.**
Test B (10): Result is "No". (Expected "No"). **Passed.**
The Result: The Mutant **Survived**. Your tests didn't notice that an 18-year-old would now be told "No" instead of "Yes."
The Solution: You must add a new test case: can_vote(18). This test would fail on the mutant, thus "killing" it and making your suite stronger.

Mutation Testing:

The Mutation Score

At the end, you calculate a score to see how effective your testing is:

Goal: A score as close to **100**

$$\text{Mutation Score} = \left(\frac{\text{Mutants Killed}}{\text{Total Mutants}} \right) \times 100$$

The "Equivalent Mutant" Problem

Sometimes a tool creates a mutant that is logically identical to the original code, even if it looks different.

Original: for (i=0; i < 10; i++)

Mutant: for (i=0; i != 10; i++)

Since they behave exactly the same, no test can ever kill this mutant. These are called **Equivalent Mutants** and are usually filtered out manually by the tester.

Mutation Testing:

Why do it? To find "blind spots" in your test suite.

The Catch: It is computationally very expensive (running thousands of tests against thousands of versions of code).

Connection to RIPR: Mutation testing forces **Infection** (changing the code) and **Propagation**, then checks if your test provides **Revealability**.

.

UNITTESTING IN EXTREME PROGRAMMING

A **TDD** approach to code development is used in the XP methodology.
The key aspect of the **TDD** approach is that a programmer writes low-level tests before writing production code.

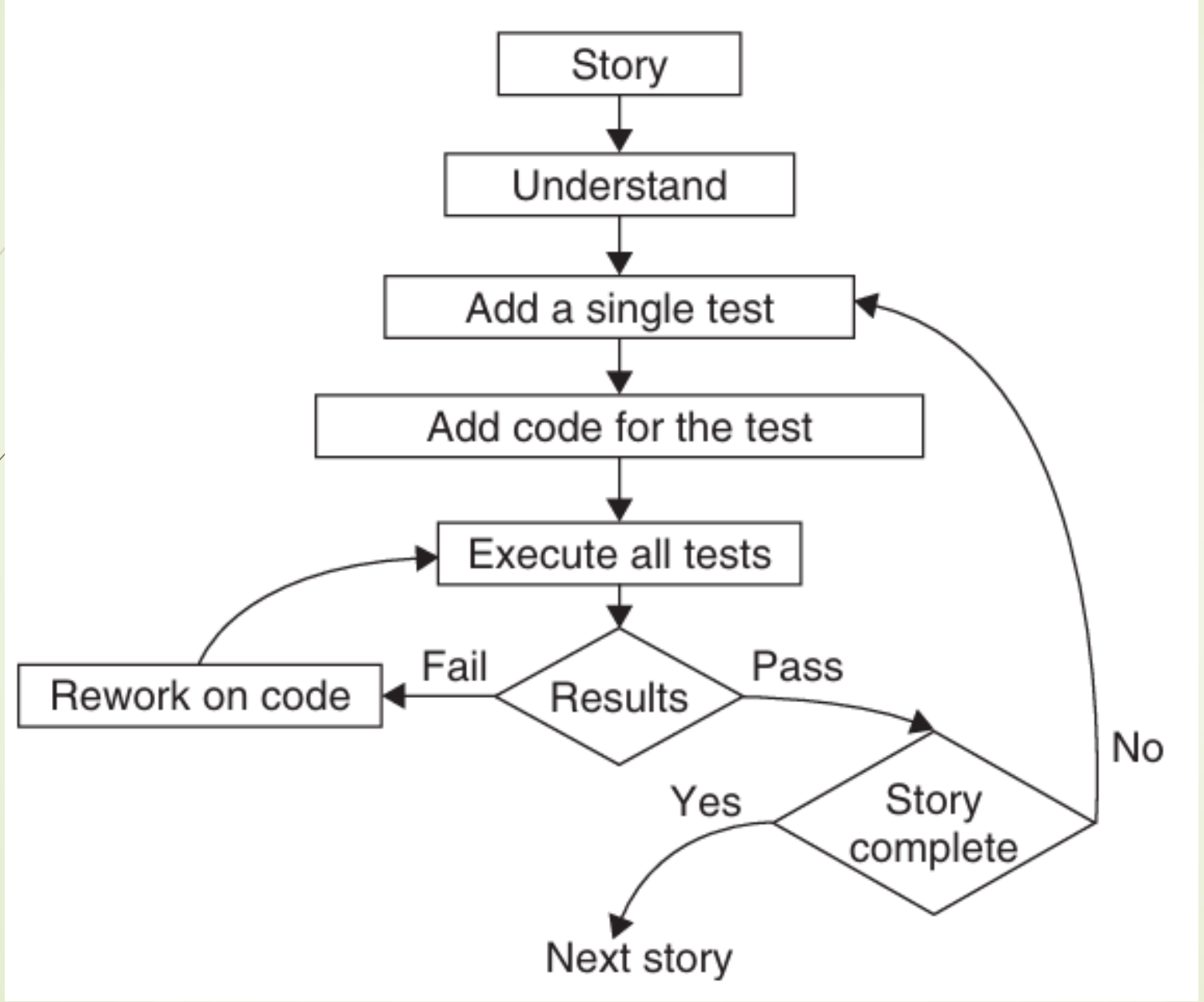
This is referred to as test first in software development.

In XP, a few unit tests are coded first, then a simple, partial system is implemented to pass the tests.

Then, one more new unit test is created, and additional code is written to pass the new test, but not more, until a new unit test is created.

The process is continued until nothing is left to test.

UNITTESTINGINEXTREMEPROGRAMMING



UNITTESTINGINEXTREMEPROGRAMMING

A TDD developer must follow the three laws proposed

- One may not write production code unless the first failing unit test is written.
- One may not write more of a unit test than is sufficient to fail.
- One may not write more production code than is sufficient to make the failing unit test pass.

Creating unit tests helps a developer focus on what needs to be done.

Requirements, that is, user stories, are nailed down firmly by unit tests.

Unit tests are released into the code repository along with the code they test.

UNITTESTINGINEXTREMEPROGRAMMING

Unit tests provide a safety net of regression tests and validation tests so that XP programmers can refactor and integrate effectively.

In **Extreme Programming (XP)**, code is constantly changing. We refactor (clean up) code daily and integrate our work into the main build every few hours. Without a "safety net," this would be a recipe for disaster.
This statement means that your suite of unit tests acts like a **24/7 automated security guard**. If you change a line of code and something—anything—goes wrong, a test will fail immediately.

This gives you the **courage** to improve the code without the fear of accidentally "breaking the world."

UNITTESTING IN EXTREME PROGRAMMING

Element	Role in XP
Validation	Proves the code does what the user asked for *right now*.
Regression	Proves that today's "cleanup" didn't break yesterday's "feature."
Refactoring	Possible only when you have tests to catch your "accidental" mistakes.
Integration	Possible only when you can prove your code doesn't break your neighbor's code.

EXTREME Programming

Extreme Programming (XP) is one of the foundational frameworks of the Agile movement.
When the Agile Manifesto was written in 2001, Kent Beck (the creator of XP) was one of the 17 original signatories.
Scrum provides the "managerial" framework for Agile, **XP** provides the "engineering" heart of it.

EXTREME Programming

How XP directly implements the four values of the Agile Manifesto:

1. **Individuals and Interactions over Processes and Tools**
    - XP Practice: Pair Programming and Collective Ownership.
    - The Logic: Instead of relying on a rigid ticketing system or complex documentation tools, XP mandates that two people sit together and talk through the code. It prioritizes the human interaction of "working together" to solve problems over the "process" of solo coding.

2. Working Software over Comprehensive Documentation

- XP Practice: Test-Driven Development (TDD) and Small Releases.
- The Logic: In XP, the "documentation" is often the test suite itself. Because the team releases small updates every week or two, the focus is always on having a functional, "working" system rather than a 200-page manual describing a system that hasn't been built yet.

EXTREME Programming

How XP directly implements the four values of the Agile Manifesto:

3. Customer Collaboration over Contract Negotiation

- XP Practice: On-site Customer.
- The Logic: XP requires a customer representative to be physically present (or digitally available at all times) with the team. They don't just sign a contract and leave; they collaborate daily, defining user stories and validating features as they are finished.

4. Responding to Change over Following a Plan

- XP Practice: Refactoring and Simplicity.
- The Logic: XP assumes the plan will change. By keeping the code simple and constantly "refactoring" (cleaning it up), the software remains flexible. The team doesn't fear change; they embrace it because their technical practices (like TDD) make it safe to change the code at any time.

EXTREME Programming

"Is XP the same as Scrum?" The distinction:

- Scrum is about Product Management: It defines roles (Scrum Master, Product Owner), ceremonies (Sprints, Daily Standups), and artifacts (Backlog).

- XP is about Software Engineering: It defines how you actually write the code (TDD, Pair Programming, CI, Refactoring).

Most successful Agile teams actually do "Scrum-XP"—they use Scrum to manage the project and XP practices to ensure the code doesn't turn into a "big ball of mud."