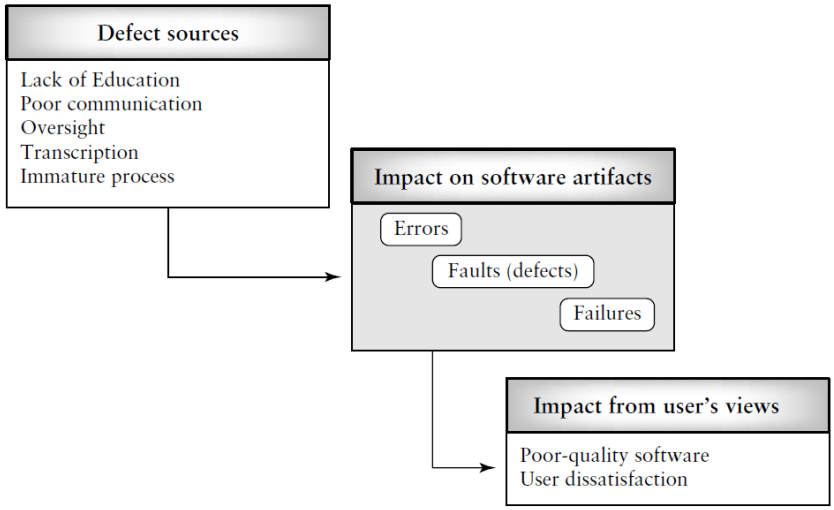
The fundamental principles of testing are as follows:

1. **The goal of testing is to find defects before customers find them out.**
2. **Exhaustive testing is not possible**; program testing can only show the presence of defects, never their absence.
3. **Testing applies all through the software life cycle** and is not an end-of-cycle activity.
4. **Understand the reason behind the test.**
5. **Test the tests first.**
6. **Tests develop immunity** and have to be revised constantly.
7. **Defects occur in convoys or clusters,** and testing should focus on these convoys.
8. **Testing encompasses defect prevention.**
9. **Testing is a fine balance of defect prevention and defect detection.**
10. **Intelligent and well-planned automation** is key to realizing the benefits of testing.
11. **Testing requires talented, committed people** who believe in themselves and work in teams.

Defects, Hypotheses, and Tests

1. Origins of Defects

Defects in software originate from various sources. Even with the best development processes, errors occur, leading to defects. These defects, when not addressed, can cause software failures. The primary sources of defects are:



1.1 Education

- A software engineer may not have the necessary educational background to correctly develop a software component.
- Example: If a developer does not understand operator precedence in a programming language, they might introduce a calculation defect in the code.

1.2 Communication

- Poor communication between developers can introduce defects.
- Example: If two engineers work on interfacing modules but fail to communicate properly, one may assume error-checking exists while the other omits it, leading to a defect.

1.3 Oversight

- Developers may forget to include necessary components in the software.
- Example: A missing initialization statement for a variable can cause unexpected behavior.

1.4 Transcription

- Even when a developer knows what to do, they may make mistakes while writing code.
- Example: A developer may misspell a variable name, causing a defect.

1.5 Process Issues

- A flawed development process can lead to defects.
- Example: If a project does not allocate enough time for a detailed specification, defects can emerge from vague or incomplete requirements.

2. Defect Hypotheses and Software Testing

Testers use the hypotheses to:

- design test cases;
- design test procedures;
- assemble test sets;
- select the testing levels (unit, integration, etc.) appropriate for the tests;
- evaluate the results of the tests

To identify defects, software testers use a hypothesis-based approach:

1. Testers **develop hypotheses** about possible defects.
2. They **design test cases** based on these hypotheses.
3. **Tests are executed**, and the results are analyzed.
4. A **successful test reveals a defect**, confirming the hypothesis.

This approach is similar to a doctor's diagnosis process. A doctor:

- Observes symptoms
- Hypothesizes about diseases
- Conducts tests to confirm the illness
- Prescribes treatment

Similarly, a tester:

- Analyzes the software for possible defect areas
- Creates test cases targeting those defects
- Runs tests to verify the presence of defects
- Reports and tracks defects for resolution

Fault Model

- A fault model links errors (e.g., a missing requirement or incorrect design element) to faults in the software.
- It helps testers **predict where defects are likely to occur** and develop test cases accordingly.
- Example: A common fault is "incorrect operator precedence," which can cause

3. Defect Classes, the Defect Repository, and Test Design

Defects are classified based on their **point of origin** in the software life cycle:

3.1 Requirements and Specification Defects

These defects arise due to incomplete, ambiguous, or incorrect specifications.

Types of Requirement Defects:

1. **Functional Description Defects** – Errors in defining what the software should do. “Features may be described as distinguishing characteristics of a software component or system.”

2. **Feature Defects** – Missing, incorrect, or unnecessary features.
3. **Feature Interaction Defects** – Incorrect behavior when multiple features interact.
4. **Interface Description Defects** – Miscommunication about how software interacts with external components.

Example: If a requirement says "the software must allow adding customers," but does not specify the maximum number of customers allowed, a defect may arise.

3.2 Design Defects

Design defects occur when software components are incorrectly structured.

Types of Design Defects:

1. **Algorithmic and Processing Defects** – Incorrect calculations or missing logic.
2. **Control, Logic, and Sequence Defects** – Improper flow of execution (e.g., missing conditions in loops).
3. **Data Defects** – Incorrect data structures or missing fields.
4. **Module Interface Defects** – Errors in function calls and parameter types.
5. **External Interface Defects** – Miscommunication between software and external systems.

Example: If a system calculates tax incorrectly because of a missing multiplication step in an algorithm, it's a **design defect**.

3.3 Coding Defects

These defects arise from mistakes made during implementation.

Types of Coding Defects:

1. **Algorithmic and Processing Defects** – Issues like incorrect operator precedence.
2. **Control, Logic, and Sequence Defects** – Improper branching, missing loops, or infinite loops.
3. **Typographical Defects** – Syntax errors, misspellings in variable names.
4. **Initialization Defects** – Using a variable before initializing it.
5. **Data Flow Defects** – Misusing variables (e.g., using an uninitialized variable).
6. **Module Interface Defects** – Incorrect function calls.
7. **Code Documentation Defects** – Poorly written comments leading to misunderstandings.
8. **External Interface Defects** – Incorrect API calls or system interactions.

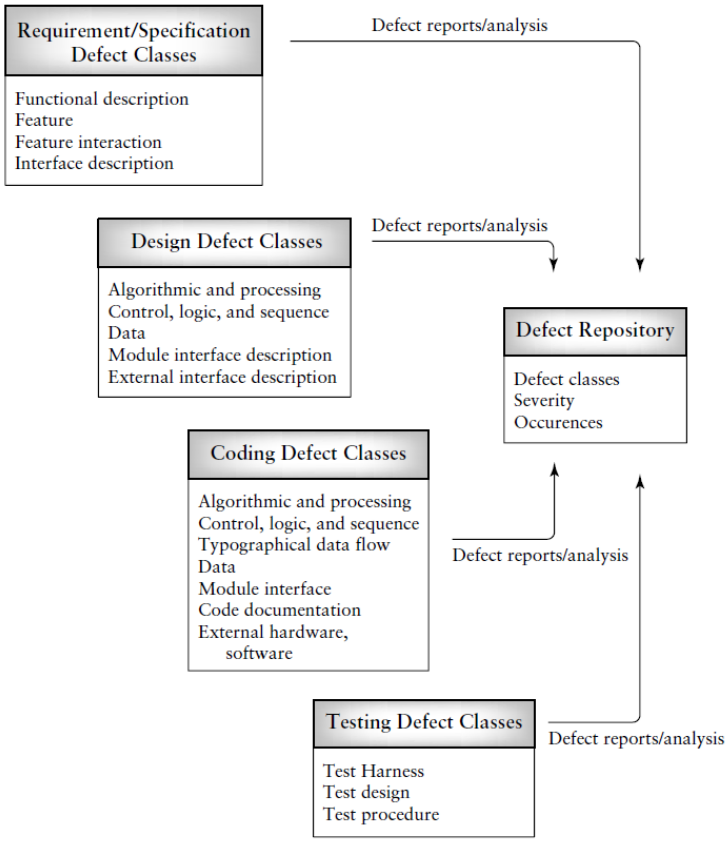
Example: Writing `i=1` instead of `i==1` in a condition may cause incorrect behavior.

3.4 Testing Defects

Even test cases and test environments can have defects.

Types of Testing Defects:

1. **Test Harness Defects** – Errors in test support code.
2. **Test Case Design Defects** – Missing or incorrect test cases.



4. Defect Examples – The Coin Problem

A sample problem illustrates how defects propagate across development phases.

Specification for Program `calculate_coin_value`

This program calculates the total dollars and cents value for a set of coins. The user inputs the amount of pennies, nickels, dimes, quarters, half-dollars, and dollar coins held. There are six different denominations of coins. The program outputs the total dollar and cent values of the coins to the user.

Inputs: `number_of_coins` is an integer
Outputs: `number_of_dollars` is an integer
`number_of_cents` is an integer

4.1 Specification Defects

- Missing input validation: The system allows negative coin values.
- Lack of user guidance: No prompts specify input format.

```
Program calculate_coin_values
 number_of_coins is integer
 total_coin_value is integer
 number_of_dollars is integer
 number_of_cents is integer
 coin_values is array of six integers representing
 each coin value in cents
 initialized to: 1,5,10,25,25,100
begin
 initialize total_coin_value to zero
 initialize loop_counter to one
 while loop_counter is less than six
 begin
 output "enter number of coins"
 read (number_of_coins)
 total_coin_value = total_coin_value +
 number_of_coins * coin_value[loop_counter]
 increment loop_counter
 end
 number_dollars = total_coin_value / 100
 number_of_cents = total_coin_value - 100 * number_of_dollars
 output (number_of_dollars, number_of_cents)
 end
```

```
/*************************************************************
program calculate_coin_values calculates the dollar and cents
value of a set of coins of different denominations input by the user
denominations are pennies, nickels, dimes, quarters, half dollars,
and dollars
*************************************************************/
main ()
{
    int total_coin_value;
    int number_of_coins = 0;
    int number_of_dollars = 0;
    int number_of-cents = 0;
    int coin_values = {1,5,10,25,25,100};
    {
        int i = 1;
        while ( i < 6)
```

```
        {
            printf("input number of coins\n");
            scanf ("%d", number_of_coins);
            total_coin_value = total_coin_value +
            (number_of_coins * coin_value{i});
        }
        i = i + 1;
        number_of_dollars = total_coin_value/100;
        number_of_cents = total_coin_value - (100 *
number_of_dollars);
        printf("%d\n", number_of_dollars);
        printf("%d\n", number_of-cents);
    }
}
/*************************************************************/
```

4.2 Design Defects

- Incorrect loop condition (`while i < 6` should be `while i <= 6`).
- Missing error handling for invalid inputs.
- Incorrect data: The `coin_values` array mistakenly has `{1,5,10,25,25,100}` instead of `{1,5,10,25,50,100}`.

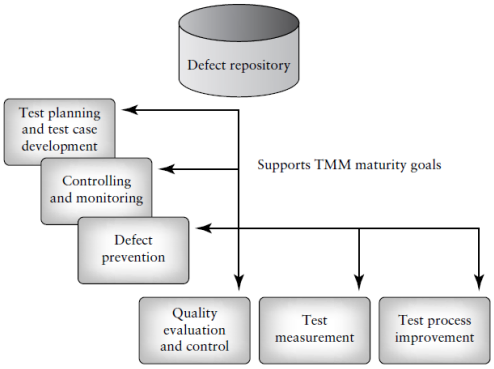
4.3 Coding Defects

- `i = i + 1` placed outside the loop, causing an infinite loop.
- `scanf("%d", number_of_coins);` should be `scanf("%d", &number_of_coins);` (missing address-of operator).
- Uninitialized variable `total_coin_value`.

This example shows how **defects introduced in early phases propagate and worsen in later phases**.

5. Developer/Tester Support for a Defect Repository

A **defect repository** helps track defects and improve software quality.



5.1 Benefits of a Defect Repository

- Helps in **test planning** by identifying common defects.
- Improves **debugging efficiency** by cataloging previous issues.
- Supports **defect prevention** by tracking root causes.
- Enhances **test measurement and process improvement**.

5.2 Steps to Develop a Defect Repository

1. **Define a defect classification scheme** (e.g., requirements, design, coding defects).
2. **Log defects systematically**, noting their severity and impact.
3. **Analyze defect data** to refine testing strategies.
4. **Use defect trends** to improve the development process.

****

1. What are the typical origins of defects? From your own personal experiences, what are the major sources of defects in the software artifacts that you have developed?

Typical origins of defects include:

- **Education**: Developers may lack knowledge of certain programming concepts.
- **Communication**: Misunderstandings between team members can lead to errors.
- **Oversight**: Missing initialization, improper logic, or forgetting necessary components.
- **Transcription**: Typographical mistakes, incorrect syntax, or misspelled variable names.
- **Process Issues**: Inadequate requirements, rushed development, or lack of proper reviews.

Personal Experience:

- While developing a web application, I once **forgot to initialize a variable**, leading to a runtime error.
- In a team project, **miscommunication about API endpoints** caused integration issues.

- **Not handling edge cases properly** in an algorithm resulted in unexpected behavior.

2. Programmer A and Programmer B are working on a group of interfacing modules. Programmer A tends to be a poor communicator and does not get along well with Programmer B. Due to this situation, what types of defects are likely to surface in these interfacing modules? What are the likely defect origins?

Likely Defects:

- **Interface Description Defects**: Misalignment in how data is passed between modules.
- **Module Interface Defects**: Incorrect parameter types, missing arguments, or improper function calls.
- **Feature Interaction Defects**: Misunderstanding of how different features interact.

Likely Defect Origins:

- **Poor Communication**: Lack of proper discussions about function signatures and module dependencies.
- **Misunderstandings in Specifications**: If one developer assumes the presence of an error-checking mechanism while the other does not implement it.
- **Process Issues**: If proper interface documentation and team collaboration strategies are not enforced.

3. Suppose you are a member of a team that was designing a defect repository. What organizational approach would you suggest and why? What information do you think should be associated with each defect? Why is this information useful, and who would use it?

Suggested Organizational Approach:

- **Centralized Defect Repository**: A shared database where all defects from different projects are stored.
- **Defect Classification Scheme**: Categorizing defects based on requirements, design, coding, and testing.
- **Regular Updates & Reviews**: Periodically reviewing defect data to improve software quality.

Information Associated with Each Defect:

- **Defect ID**: Unique identifier for tracking defects.
- **Defect Type**: (Requirement, Design, Code, Testing, etc.)
- **Severity Level**: (Critical, High, Medium, Low)
- **Module Affected**: The component where the defect occurred.
- **Steps to Reproduce**: Instructions for replicating the defect.
- **Date & Time Logged**: When the defect was reported.
- **Status**: (New, In Progress, Fixed, Closed)

Why This Information is Useful:

- **Testers use it to design test cases for similar defects.**
- **Developers use it to avoid repeating common mistakes.**
- **Project Managers use it to measure software quality and improve processes.**

4. What type of defect classification scheme is used by your university or organization? How would you compare it to the classification scheme used in this text for clarity, learnability, and ease of use?

- Some universities/organizations use defect classification schemes like **IEEE Standard Classification for Software Anomalies** or **custom defect tracking systems**.
- The classification scheme in the book (Requirements, Design, Code, Testing) is **clear and well-structured** compared to generalized schemes, making it easier to understand and apply.
- The book provides **detailed descriptions and examples**, which make it more **learnable and practical** for students and professionals.

5. Suppose you were reviewing a requirements document and noted that a feature was described incompletely. How would you classify this defect? How would you ensure that it was corrected?

Classification:

- **Requirement/Specification Defect → Functional Description Defect**

Correction Process:

1. **Raise the issue** in a defect tracking system or review meeting.
2. **Clarify with stakeholders** (customers, product owners) to understand the missing details.
3. **Revise the requirement document** with complete and unambiguous descriptions.
4. **Review and approve** the corrected document before proceeding to design and development.

6. Suppose you are testing a code component, and you discover a defect: it calculates an output variable incorrectly.

(a) How would you classify this defect?

- **Coding Defect → Algorithmic and Processing Defect**

(b) What are the likely causes of this defect?

- Incorrect mathematical operations or formulas.
- Misunderstanding of functional requirements.
- Typographical errors (e.g., using + instead of -).
- Data type conversion issues (e.g., integer division instead of floating-point division).

(c) What steps could have been taken to prevent this type of defect from propagating to the code?

- **Better Code Reviews**: Reviewing mathematical expressions and logic before implementation.
- **Unit Testing**: Writing test cases to validate outputs against expected values.
- **Pair Programming**: Having another developer review the logic during implementation.
- **Clear Requirements**: Ensuring that calculations are well-defined in the requirement phase.

7. Suppose you are testing a code component, and you find a mismatch in the order of parameters for two of the code procedures. Address the same three items that appear in question 6 for this scenario.

(a) How would you classify this defect?

- **Coding Defect → Module Interface Defect**

(b) What are the likely causes of this defect?

- Poor communication between developers about function signatures.
- Lack of adherence to API documentation.
- Copy-paste errors or incorrect function calls.
- Inconsistent parameter ordering across modules.

(c) What steps could have been taken to prevent this type of defect from propagating to the code?

- **Better Interface Documentation**: Maintaining updated function signatures.
- **Code Reviews**: Ensuring correct function calls are used.
- **Automated Linting & Static Analysis**: Using tools like ESLint (for JavaScript) or Pylint (for Python) to catch mismatches.
- **Strong Typing & IDE Hints**: Using strongly typed languages (e.g., TypeScript, Java) that provide compile-time checks.