UNIT - I

INTRODUCTION

Testing as an Engineering Activity – Testing as a Process – Testing axioms – Basic definitions – Software Testing Principles – The Tester's Role in a Software Development Organization – Origins of Defects – Cost of defects – Defect Classes – The Defect Repository and Test Design – Defect Examples – Developer/Tester Support of Developing a Defect Repository – Defect Prevention strategies.

In response to the demand for high-quality software, and the need for well-educated software professionals, there is a movement to change the way software is developed and maintained, and the way developers and maintainers are educated.

In fact, the profession of software engineering is slowly emerging as a formal engineering discipline. As a new discipline it will be related to other engineering disciplines, and have associated with it a defined body of knowledge, a code of ethics, and a certification process.

The education and training of engineers in each engineering discipline is based on the teaching of related scientific principles, engineering processes, standards, methods, tools, measurement and best practices

The IEEE Computer Society and the Association of Computing Machinery (ACM), the two principal societies for software professionals, have appointed joint task forces.

The goals of the task force teams are to

- define a body of knowledge that covers the software engineering discipline
- discuss the nature of education for this new profession, and
- to define a code of ethics for the software engineer

Engineering approach to software development implies that:

- The development process is well understood;
- Projects are planned;
- Life cycle models are defined and adhered to;
- Standards are in place for product and process;
- Measurements are employed to evaluate product and process quality;
- Components are reused;
- Validation and Verification processes play a key role in quality determination;
- Engineers have proper education, training, and certification.

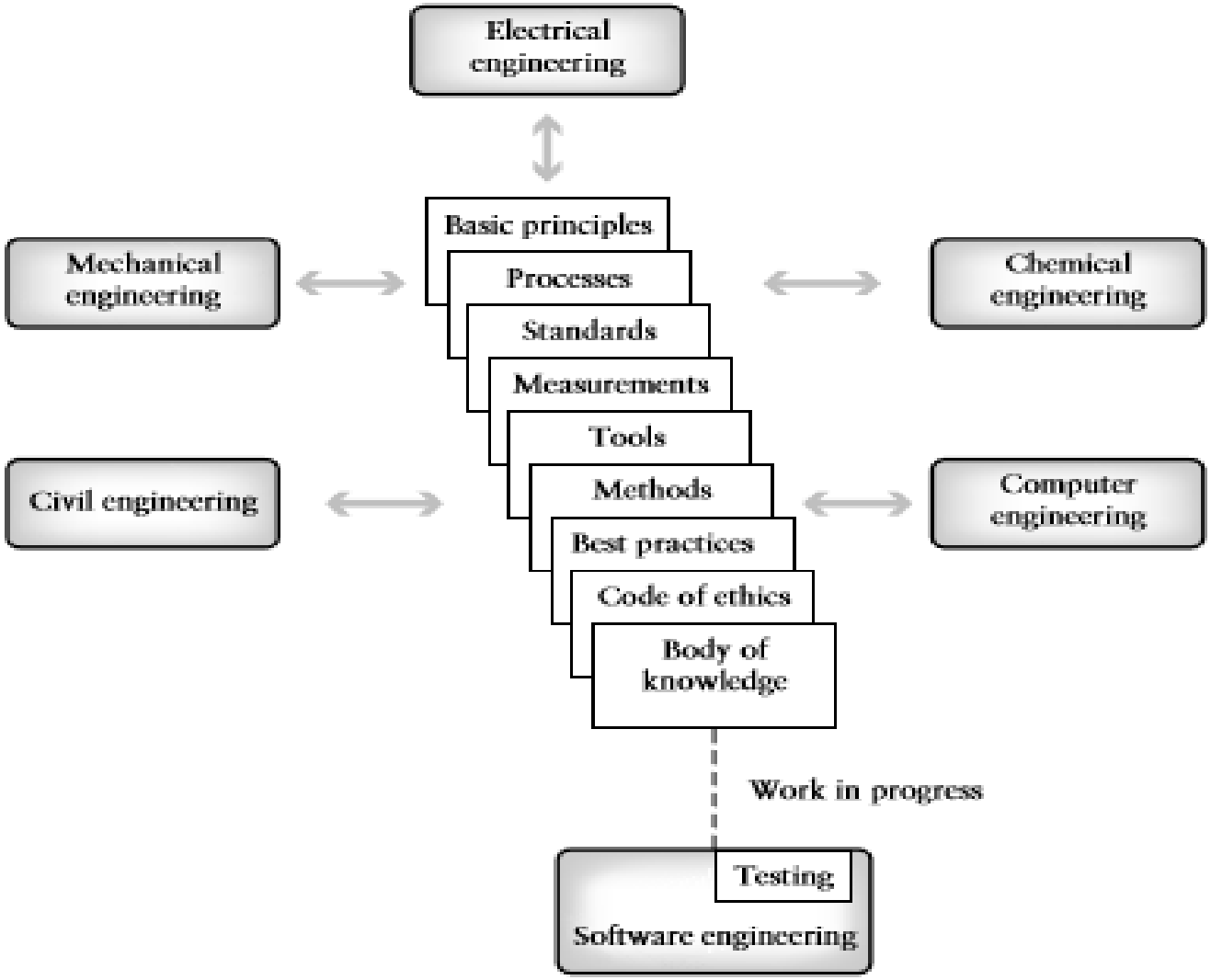


FIG. 1.1

Elements of the engineering disciplines.

Test Specialist

A test specialist is one whose education is based on the principles, practices, and processes that constitute the software engineering discipline, and whose specific focus is on one area of that discipline—software testing.

A test specialist who is trained as an engineer should have knowledge of test-related principles, processes, measurements, standards, plans, tools, and methods, and should learn how to apply them to the testing tasks to be performed.

The Role of Process in Software Quality

The practices that contribute to the development of high-quality software are

- project planning,
- requirements management,
- development of formal specifications,
- structured design with use of information hiding and encapsulation,
- design and code reuse,
- inspections and reviews,
- product and process measures,
- education and training of software professionals,
- development and application of CASE tools,
- use of effective testing techniques, and
- integration of testing activities into the entire life cycle.

Process:

Process, in the software engineering domain, is the set of methods, practices, standards, documents, activities, policies, and procedures that software engineers use to develop and maintain a software system and its associated artifacts, such as project and test plans, design documents, code, and manuals.

Testing is a vital component of a quality software process, and is one of the most challenging and costly activities carried out during software development and maintenance. In spite of its vital role in the production of quality software, existing process evaluation and improvement models such as the CMM, Bootstrap, and ISO-9000 have not adequately addressed testing process issues.

The Testing Maturity Model (TMM), was developed to address this issue

Testing as a Process

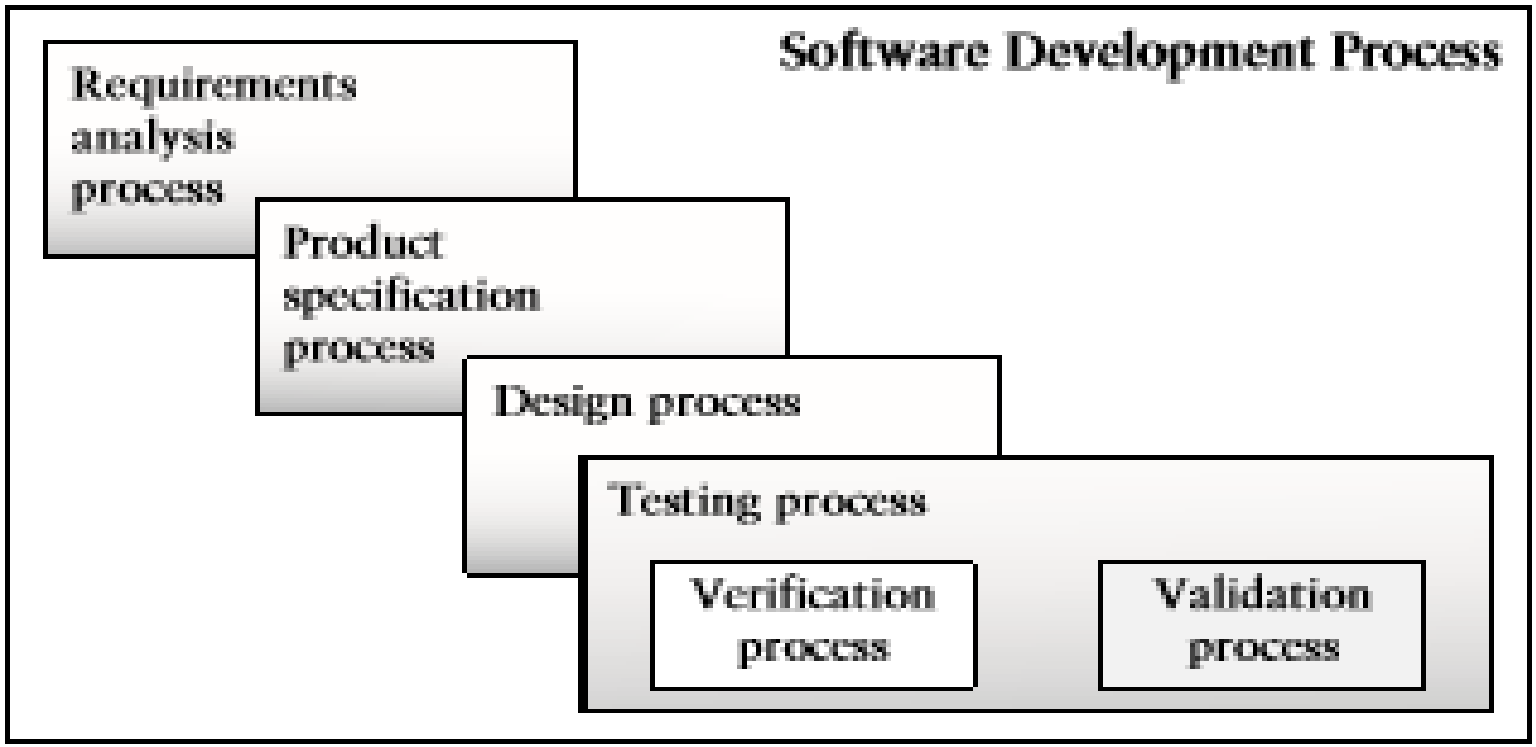


FIG. 1.3

Example processes embedded in the software development process.

Definition : Testing

- Testing is generally described as a group of procedures carried out to evaluate some aspect of a piece of software.
- Testing can be described as a process used for revealing defects in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes.

- Testing itself is related to two other processes called verification and validation

Validation (Are we building the right product) is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements

Verification (Are we building the product right) is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase

Activities of Testing domain

- Technical reviews,
- Test planning,
- Test tracking,
- Test case design,
- Unit test,
- Integration test,
- System test,
- Acceptance test, and
- Usability test

Testing is a dual-purpose process which contains processes for

- revealing defects, and
- evaluating quality attributes of the software such as reliability, security, usability, and correctness.

Debugging

Debugging, or fault localization is the process of
(1) locating the fault or defect,
(2) repairing the code, and
(3) retesting the code.

Testing as a process must be managed economically, technically and organizationally. Economic factors constitute time and resources. Technical factors include techniques, methods, measurements that guarantee that the s/w is defect free and reliable.

An organizational policy for testing must be defined and documented. Testing procedures and steps must be defined and documented. Testing must be planned, testers should be trained, the process should have associated quantifiable goals that can be measured and monitored. Testing as a process should be able to evolve to a level where there are mechanisms in place for making continuous improvements.

An Overview of the Testing Maturity Model

Benefits of test process improvement are the following:

- smarter testers
- higher quality software
- the ability to meet budget and scheduling goals
- improved planning
- the ability to meet quantifiable testing goals

TMM Levels

TMM also follows what is called a staged architecture for process improvement models. It contains stages or levels through which an organization passes as its testing process evolves from one that is ad hoc and unmanaged to one that is managed, defined, measured, and optimizable

There are **five levels** in the TMM that prescribe a maturity hierarchy and an evolutionary path to test process improvement. The characteristics of each level are described in terms of testing capability organizational goals, and roles/responsibilities for the key players in the testing process, the managers, developers/testers, and users/clients.

Each level with the exception of level 1 has a structure that consists of the following

- A set of maturity goals.
- Supporting maturity sub goals.
- Activities, tasks and responsibilities (ATR)

FIG. 1.4

The internal structure of TMM maturity levels.

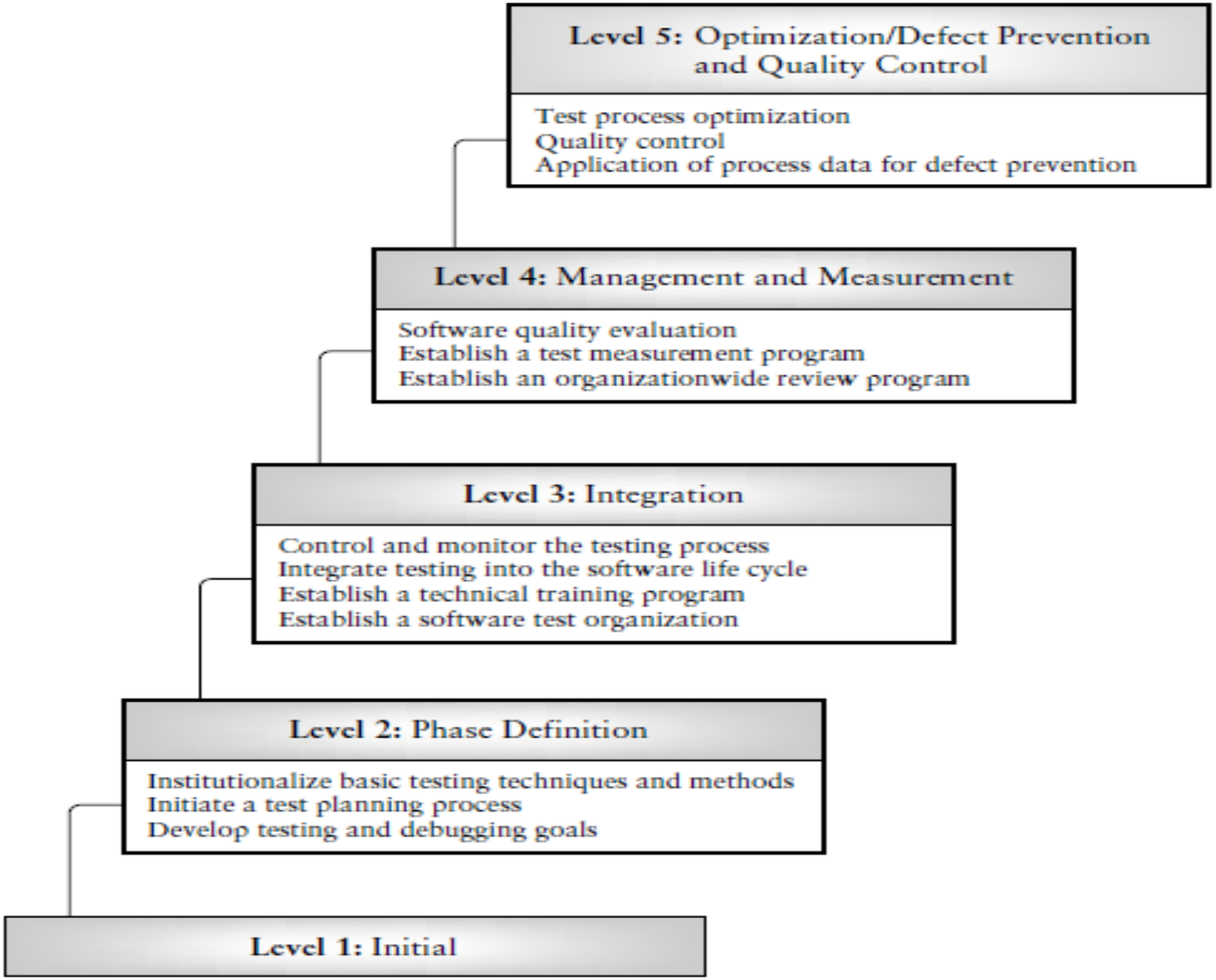


FIG. 1.5

The 5-level structure of the testing maturity model.

Level 1—Initial: (No maturity goals)

At TMM level 1, testing is a chaotic process; it is ill-defined, and not distinguished from debugging.

- A documented set of specifications for software behaviour often does not exist.
- Tests are developed in an ad hoc way after coding is completed. Testing and debugging are interleaved to get the bugs out of the software.
- The objective of testing is to show the software works (it is minimally functional)
- Software products are often released without quality assurance.
- There is a lack of resources, tools and properly trained staff. This type of organization would be at level 1 of the CMM.

Level 2—Phase Definition: (Goal 1: Develop testing and debugging goals; Goal 2: Initiate a testing planning process; Goal 3: Institutionalize basic testing techniques and methods)

- At level 2 of the TMM, testing is separated from debugging and is defined as a phase that follows coding.
- It is a planned activity; however, test planning at level 2 may occur after coding for reasons related to the immaturity of the testing process.
- It should be planned only when the code is complete.
- The primary goal of testing at this level of maturity is to show that the software meets its stated specifications. Basic testing techniques and methods are in place; for example, use of black box and white box testing strategies, and a validation cross-reference matrix.
- Testing is multileveled: there are unit, integration, system, and acceptance levels.
- Many quality problems at this TMM level occur because test planning occurs late in the software life cycle.
- In addition, defects are propagated from the requirements and design phases into the code.

Level 3—Integration: (Goal 1: Establish a software test organization; Goal 2: Establish a technical training program; Goal 3: Integrate testing into the software life cycle; Goal 4: Control and monitor testing)

- At TMM level 3, testing is no longer a phase that follows coding, but is integrated into the entire software life cycle. Organizations can build on the test planning skills they have acquired at level 2. Unlike level 2, planning for testing at TMM level 3 begins at the requirements phase and continues throughout the life cycle supported by a version of the V-model
- Test objectives are established with respect to the requirements based on user/client needs, and are used for test case design.
- There is a test organization, and testing is recognized as a professional activity.
- There is a technical training organization with a testing focus.
- Testing is monitored to ensure it is going according to plan and actions can be taken if deviations occur.
- Basic tools support key testing activities, and the testing process is visible in the organization.

Level 4—Management and Measurement: (Goal 1: Establish an organization wide review program; Goal 2: Establish a test measurement program; Goal 3: Software quality evaluation)

- Testing at level 4 becomes a process that is measured and quantified.
- Reviews at all phases of the development process are now recognized as testing/quality control activities. They are a compliment to execution based tests to detect defects and to evaluate and improve software quality.
- An extension of the V-model can be used to support the implementation of this goal.
- Software products are tested for quality attributes such as reliability, usability, and maintainability.
- Test cases from all projects are collected and recorded in a test case database for the purpose of test case reuse and regression testing.
- Defects are logged and given a severity level.

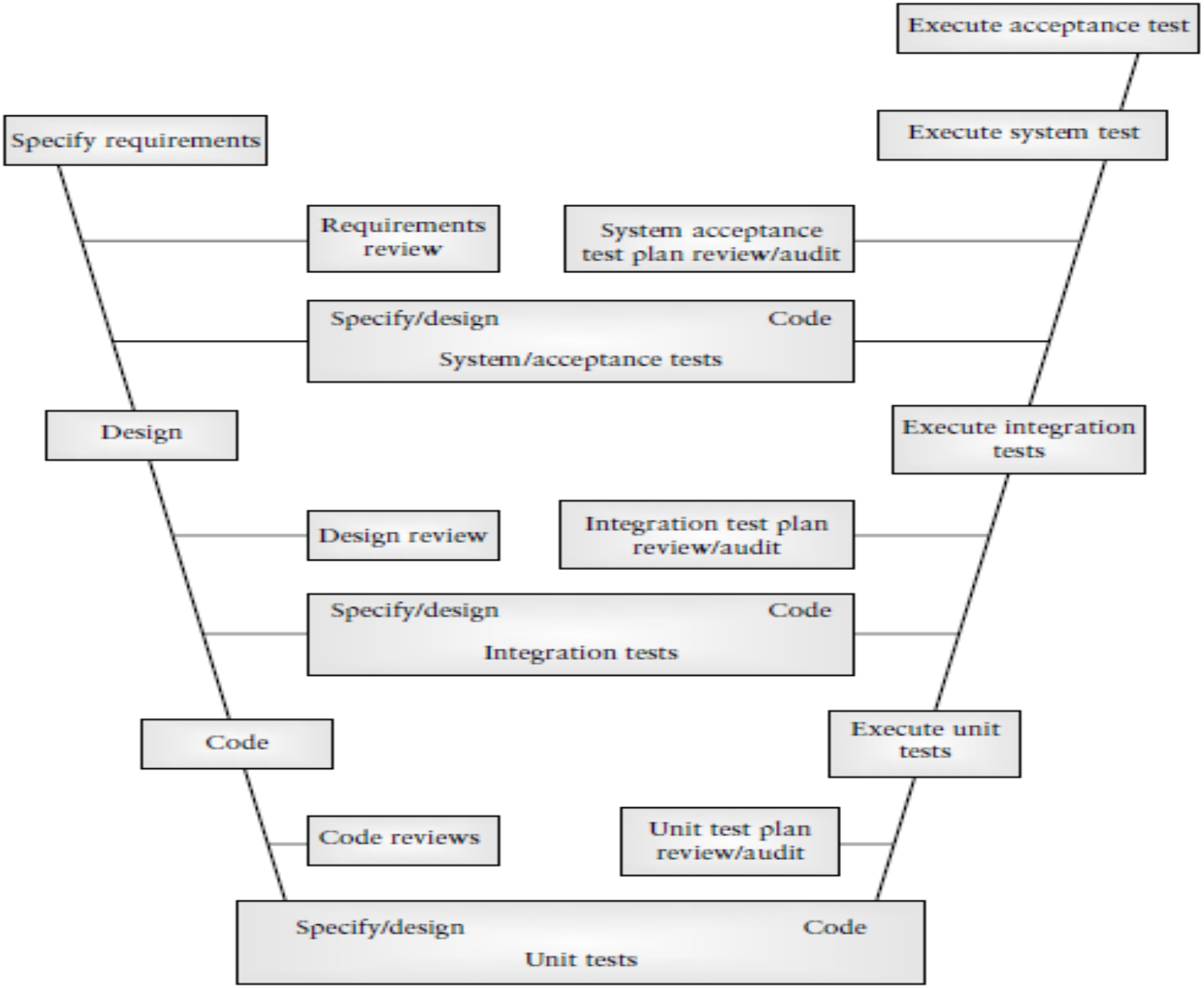


FIG. 1.6

The Extended/Modified V-model.

Level 5—Optimization/Defect Prevention/Quality Control: (Goal 1: Defect prevention; Goal 2: Quality control; Goal 3: Test process optimization)

- Through achievement of the maturity goals at levels 1–4 of the TMM, the testing process is now said to be defined and managed; its cost and effectiveness can be monitored.
- At level 5, mechanisms are in place so that testing can be fine-tuned and continuously improved. Defect prevention and quality control are practiced.
- Statistical sampling, measurements of confidence levels, trustworthiness, and reliability drive the testing process.
- Automated tools totally support the running and rerunning of test cases. Tools also provide support for test case design, maintenance of test-related items, and defect collection and analysis. The collection and analysis of test-related metrics also has tool support.
- Process reuse is also a practice at TMM level 5 supported by a Process Asset Library (PAL).

Basic Definitions

- **Errors**

An error is a mistake, misconception, or misunderstanding on the part of a software developer.

- **Faults (Defects)**

A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification.

- **Failures**

A failure is the inability of a software system or component to perform its required functions within specified performance requirements.

- **Test**

A test is a group of related test cases, or a group of related test cases and test procedures.

- **Test Oracle**

A test oracle is a document, or piece of software that allows testers to determine whether a test has been passed or failed.

- **Test case**

A test case in a practical sense is a test-related item which contains the following information:

1. *A set of test inputs*. These are data items received from an external source by the code under test. The external source can be hardware, software, or human.

2. *Execution conditions*. These are conditions required for running the test, for example, a certain state of a database, or a configuration of a hardware device.

3. *Expected outputs*. These are the specified results to be produced by the code under test.

- **Test Bed**

A test bed is an environment that contains all the hardware and software needed to test a software component or a software system.

- **Software Quality**

Two concise definitions for quality are found in the *IEEE Standard Glossary of Software Engineering Terminology* :

1. Quality relates to the degree to which a system, system component, or process meets specified requirements.
2. Quality relates to the degree to which a system, system component, or process meets customer or user needs, or expectations.

- **Metric**

A metric is a quantitative measure of the degree to which a system, system component, or process possesses a given attribute.

- **Quality metric**

A quality metric is a quantitative measurement of the degree to which an item possesses a given quality attribute.

Quality attributes :

- **correctness—**the degree to which the system performs its intended function
- **reliability—**the degree to which the software is expected to perform its required functions under stated conditions for a stated period of time
- **usability—**relates to the degree of effort needed to learn, operate, prepare input, and interpret output of the software
- **integrity—**relates to the system's ability to withstand both intentional and accidental attacks
- **portability—**relates to the ability of the software to be transferred from one environment to another
- **maintainability—**the effort needed to make changes in the software
- **interoperability—**the effort needed to link or couple one system to another.

- **Testability**

1. the amount of effort needed to test the software to ensure it performs according to specified requirements (relates to number of test cases needed),

2. the ability of the software to reveal defects under testing conditions (some software is designed in such a way that defects are well hidden during ordinary testing conditions).

- **Software Quality Assurance Group**

The software quality assurance (SQA) group is a team of people with the necessary training and skills to ensure that all necessary actions are taken during the development process so that the resulting software conforms to established technical requirements.

- **Reviews**

A review is a group meeting whose purpose is to evaluate a software artifact or a set of software artifacts.

Software Testing Principles

Testing principles are important to test specialists/ engineers because they provide the foundation for developing testing knowledge and acquiring testing skills. They also provide guidance for defining testing activities as performed in the practice of a test specialist.

A principle can be defined as:

1. a general or fundamental, law, doctrine, or assumption;

2. a rule or code of conduct;

3. the laws or facts of nature underlying the working of an artificial device.

- **Principle 1.**

  Testing is the process of exercising a software component using a selected set of test cases, with the intent of (i) revealing defects, and (ii) evaluating quality.

- **Principle 2.**

  When the test objective is to detect defects, then a good test case is one that has a high probability of revealing a yet undetected defect(s).

- **Principle 3.**

  Test results should be inspected meticulously.

- **Principle 4.**

  A test case must contain the expected output or result.

- **Principle 5.**

  Test cases should be developed for both valid and invalid input conditions.

- **Principle 6.**

  The probability of the existence of additional defects in a software component is proportional to the number of defects already detected in that component.

- **Principle 7.**

  Testing should be carried out by a group that is independent of the development group.

- **Principle 8.**

  Tests must be repeatable and reusable.

- **Principle 9.**

  Testing should be planned.

- **Principle 10.**

  Testing activities should be integrated into the software life cycle.

- **Principle 11.**

  Testing is a creative and challenging task

Difficulties and challenges for the tester

- A tester needs to have comprehensive knowledge of the software engineering discipline.
- A tester needs to have knowledge from both experience and education as to how software is specified, designed, and developed.
- A tester needs to be able to manage many details.
- A tester needs to have knowledge of fault types and where faults of a certain type might occur in code constructs.
- A tester needs to reason like a scientist and propose hypotheses that relate to presence of specific types of defects.

Difficulties and challenges for the tester (contd.)

- A tester needs to have a good grasp of the problem domain of the software that he/she is testing. Familiarity with a domain may come from educational, training, and work-related experiences.

- A tester needs to create and document test cases. To design the test cases the tester must select inputs often from a very wide domain. Those selected should have the highest probability of revealing a defect (Principle 2). Familiarity with the domain is essential.

- A tester needs to design and record test procedures for running the tests.

- A tester needs to plan for testing and allocate the proper resources.

- A tester needs to execute the tests and is responsible for recording results.

Difficulties and challenges for the tester (contd.)

- A tester needs to analyze test results and decide on success or failure for a test. This involves understanding and keeping track of an enormous amount of detailed information. A tester may also be required to collect and analyze test-related measurements.

- A tester needs to learn to use tools and keep abreast of the newest test tool advances.

- A tester needs to work and cooperate with requirements engineers, designers, and developers, and often must establish a working relationship with clients and users.

- A tester needs to be educated and trained in this specialized area and often will be required to update his/her knowledge on a regular basis due to changing technologies.

The Tester's Role in a Software Development Organization

- The tester's job is to reveal defects, find weak points, inconsistent behaviour, and circumstances where the software does not work as expected.
- A tester requires extensive programming experience in order to understand how code is constructed, and where, and what kind of, defects are likely to occur.
- Goal of a tester is to work with the developers to produce high-quality software that meets the customers' requirements.
- Projects should have an appropriate developer/tester ratio. The ratio will vary depending on available resources, type of project, and TMM level
- Testers also need to work along side with requirements engineers to ensure that requirements are testable, and to plan for system and acceptance test (clients are also involved in the latter).

The Tester's Role in a Software Development Organization (Contd.)

- Testers also need to work with designers to plan for integration and unit test. In addition, test managers will need to cooperate with project managers in order to develop reasonable test plans, and with upper management to provide input for the development and maintenance of organizational testing standards, polices, and goals.
- Finally, testers also need to cooperate with software quality assurance staff and software engineering process group members.
- In view of these requirements for multiple working relationships, communication and team working skills are necessary for a successful career as a tester.

- At TMM levels 1 or 2, there is no independent software test function in the organization.
- Even at levels 3 and higher of the TMM the testers may not necessarily belong to a independent organizational entity, although that is the ideal case
- Testers are specialists, their main function is to plan, execute, record, and analyze tests. They do not debug software. When defects are detected during testing, software should be returned to the developers who locate the defect and repair the code.
- Developers have a detailed understanding of the code, and are the best qualified staff to perform debugging.
- Finally, testers need the support of management. Developers, analysts, and marketing staff need to realize that testers add value to a software product in that they detect defects and evaluate quality as early as possible in the software life cycle.
- This ensures that developers release code with few or no defects, and that marketers can deliver software that satisfies the customers' requirements, and is reliable, usable, and correct.

Origins of Defects

Defects have detrimental affects on software users, and software engineers work very hard to produce high-quality software with a low number of defects.

Defects as shown in Fig stem from the following sources:

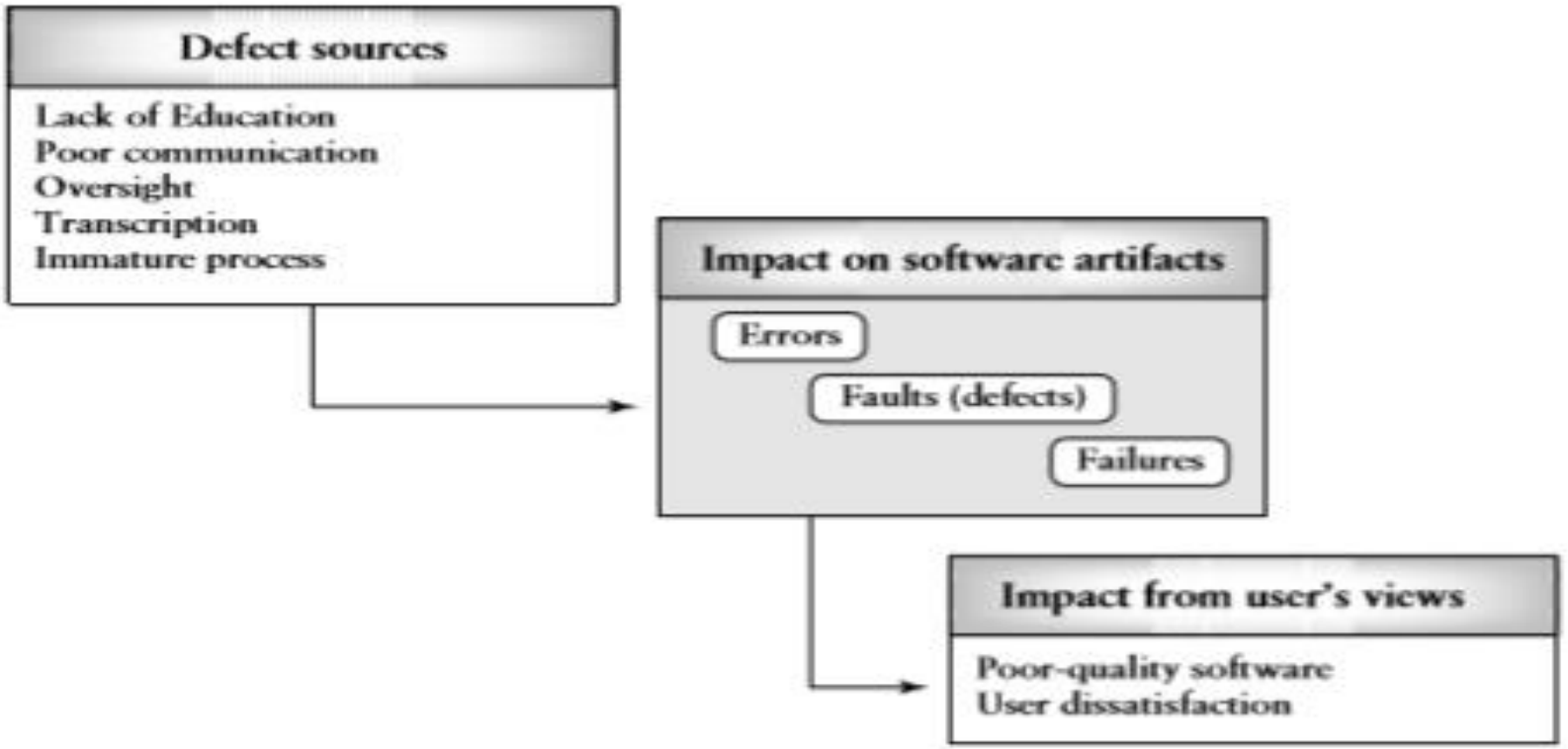
1. **Education**: The software engineer did not have the proper educational background to prepare the software artifact. For example, a software engineer who did not understand the precedence order of operators could inject a defect in an equation.

2. **Communication**: The software engineer was not informed about something by a colleague. For example, if engineer 1 and engineer 2 are working on interfacing modules, and engineer 1 does not inform engineer2 that a no error checking code will appear in the interfacing module he is developing, engineer 2 might make an incorrect assumption relating to the presence/absence of an error check, and a defect will result.

3. **Oversight**: The software engineer omitted to do something. For example, a software engineer might omit an initialization statement.

4. **Transcription**: The software engineer knows what to do, but makes a mistake in doing it. A simple example is a variable name being misspelled when entering the code.

5. **Process:** The process used by the software engineer misdirected her actions. For example, a development process that did not allow sufficient time for a detailed specification to be developed and reviewed could lead to specification defects.



When defects are present due to one or more of these circumstances, the software may fail, and the impact on the user ranges from a minor inconvenience to rendering the software unfit for use.

Testers discover these defects preferably before the software is in operation by designing test cases that have a high probability of revealing defects.

One approach is to think of software testing as an experimental activity. The results of the test experiment are analyzed to determine whether the software has behaved correctly.

In this experimental scenario a tester develops hypotheses about possible defects. Test cases are then designed based on the hypotheses. The tests are run and results analyzed to prove, or disprove, the hypotheses.

Myers describes the successful test as one that reveals the presence of a (hypothesized) defect. He compares the role of a tester to that of a doctor who is in the process of constructing a diagnosis for an ill patient. Testers as doctors need to have knowledge about possible defects (illnesses) in order to develop defect hypotheses

They 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.

A successful testing experiment will prove the hypothesis is true—that is, the hypothesized defect was present. Then the software can be repaired.

A **fault (defect) model** can be described as a link between the error made (e.g., a missing requirement, a misunderstood design element, a typographical error),and the fault/defect in the software.

A simple example of a fault model a software engineer might have in memory is “an incorrect value for a variable was observed because the precedence order for the arithmetic operators used to calculate its value was incorrect.” This could be called “an incorrect operator precedence order” fault.

The probable cause is a lack of education on the part of the programmer. Repairs include changing the order of the operators or proper use of parentheses.

To increase the effectiveness of the testing and debugging processes, software organizations need to initiate the creation of a **defect database**, or **defect repository**. The defect repository concept supports storage and retrieval of defect data from all projects in a centrally accessible location.

A defect classification scheme is a necessary first step for developing the repository. The defect repository can be organized by projects and for all projects, defects of each class are logged, along their frequency of occurrence, impact on operation, and any other useful comments.

Defects found both during reviews and execution-based testing should be cataloged. Supplementary information like defect root causes can be added.

Staff members can use this data for test planning, test design, and fault/defect diagnosis. The data can also be used for defect prevention and process improvement efforts at higher levels of testing process maturity

Defect Classes

Defects can be classified in many ways. It is important for an organization to adapt a single classification scheme and apply it to all projects.

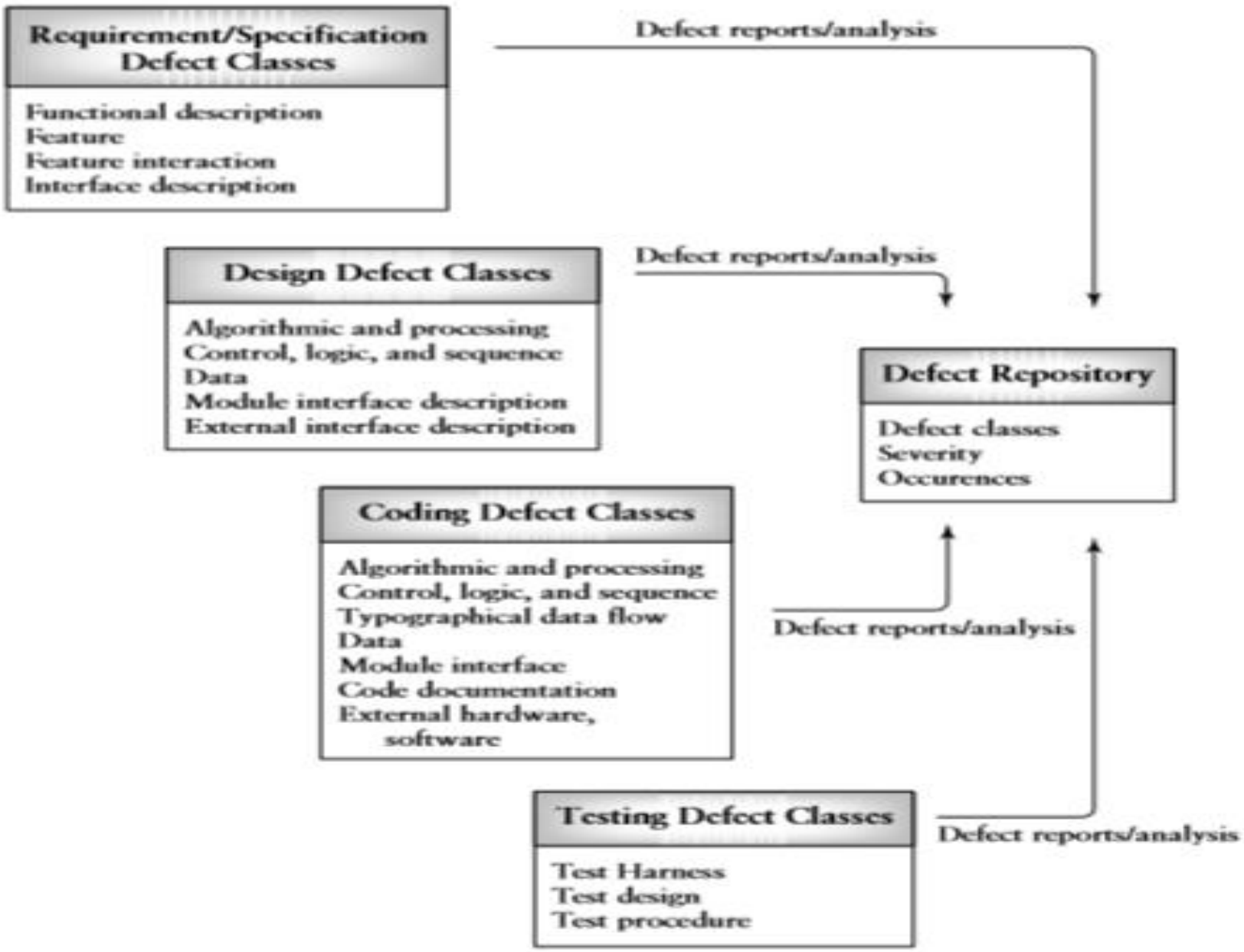
Defects, are assigned to four major classes reflecting their point of origin in the software life cycle—the development phase in which they were injected. These classes are: **requirements/ specifications, design, code, and testing defects.**

i. Requirements and Specification Defects

The beginning of the software life cycle is critical for ensuring high quality in the software being developed. Defects injected in early phases can persist and be very difficult to remove in later phases.

Many organizations have introduced the use of formal specification languages that, when accompanied by tools, help to prevent incorrect descriptions of system behavior. Some specific requirements/specification defects are:

Defect classes and the Defect Repository



1. **Functional Description Defects**

The overall description of what the product does, and how it should behave (inputs/outputs), is incorrect, ambiguous, and/or incomplete.

2. **Feature Defects**

Features may be described as distinguishing characteristics of a software component or system. Feature defects are due to feature descriptions that are missing, incorrect, incomplete, or superfluous.

3. **Feature Interaction Defects**

These are due to an incorrect description of how the features should interact. For example, suppose one feature of a software system supports adding a new customer to a customer database.

This feature interacts with another feature that categorizes the new customer. The classification feature impacts on where the storage algorithm places the new customer in the database, and also affects another feature that periodically supports sending advertising information to customers in a specific category.

4. **Interface Description Defects**

These are defects that occur in the description of how the target software is to interface with external software, hardware, and users. For detecting many functional description defects, black box testing techniques, which are based on functional specifications of the software, offer the best approach.

Black box–based tests can be planned at the unit, integration, system, and acceptance levels to detect requirements/specification defects.

ii. **Design Defects**

Design defects occur when system components, interactions between system components, interactions between the components and outside software/hardware, or users are incorrectly designed.

This covers defects in the design of algorithms, control, logic, data elements, module interface descriptions, and external software/hardware/user interface descriptions.

1. **Algorithmic and Processing Defects**

These occur when the processing steps in the algorithm as described by the pseudo code are incorrect. For example, the pseudo code may contain a

calculation that is incorrectly specified, or the processing steps in the algorithm written in the pseudo code language may not be in the correct order. In the latter case a step may be missing or a step may be duplicated.

2. **Control, Logic, and Sequence Defects**

Control defects occur when logic flow in the pseudo code is not correct. For example, branching to soon, branching to late, or use of an incorrect branching condition.

Logic defects usually relate to incorrect use of logic operators, such as less than (<), greater than (>), etc. These may be used incorrectly in a Boolean expression controlling a branching instruction.

3. **Data Defects**

These are associated with incorrect design of data structures. For example, a record may be lacking a field, an incorrect type is assigned to a variable or a field in a record, an array may not have the proper number of elements assigned, or storage space may be allocated incorrectly.

Software reviews and use of a data dictionary work well to reveal these types of defects.

4. **Module Interface Description Defects**

These are defects derived from, for example, using incorrect, and/or inconsistent parameter types, an incorrect number of parameters, or an incorrect ordering of parameters.

5. **Functional Description Defects**

The defects in this category include incorrect, missing, and/or unclear design elements. For example, the design may not properly describe the correct functionality of a module. These defects are best detected during a design review.

6. **External Interface Description Defects**

These are derived from incorrect design descriptions for interfaces with COTS components, external software systems, databases, and hardware devices (e.g., I/O devices).

Other examples are user interface description defects where there are missing or improper commands, improper sequences of commands, lack of proper messages, and/or lack of feedback messages for the user.

iii. Coding Defects

Coding defects are derived from errors in implementing the code. Coding defects classes are closely related to design defect classes especially if pseudo code has been used for detailed design.

Some coding defects come from a failure to understand programming language constructs, and miscommunication with the designers. Others may have transcription or omission origins.

1. **Algorithmic and Processing Defects**

Adding levels of programming detail to design, code-related algorithmic and processing defects include unchecked overflow and underflow conditions, comparing inappropriate data types, converting one data type to another, incorrect ordering of arithmetic, misuse or omission of parentheses, precision loss, and incorrect use of signs.

2. **Control, Logic and Sequence Defects**

On the coding level these would include incorrect expression of case statements, incorrect iteration of loops(loop boundary problems),and missing paths.

3. **Typographical Defects**

These are principally syntax errors, for example, incorrect spelling of a variable name, that are usually detected by a compiler, self-reviews, or peer reviews.

4. **Initialization Defects**

These occur when initialization statements are omitted or are incorrect. This may occur because of misunderstandings or lack of communication between programmers, and/or programmers and designers, carelessness, or misunderstanding of the programming environment.

5. **Data-Flow Defects**

There are certain reasonable operational sequences that data should flow through. For example, a variable should be initialized, before it is used in a calculation or a condition. It should not be initialized twice before there is an intermediate use. A variable should not be disregarded before it is used.

Occurrences of these suspicious variable uses in the code may, or may not, cause anomalous behavior. However, their presence indicates an error has occurred and a problem exists that needs to be addressed.

6. **Data Defects**

These are indicated by incorrect implementation of data structures. For example, the programmer may omit a field in a record, an incorrect type or access is assigned to a file, an array may not be allocated the proper number of elements. Other data defects include flags, indices, and constants set incorrectly.

7. **Module Interface Defects**

As in the case of module design elements, interface defects in the code may be due to using incorrect or inconsistent parameter types, an incorrect number of parameters, or improper ordering of the parameters.

In addition to defects due to improper design, and improper implementation of design, programmers may implement an incorrect sequence of calls or calls to nonexistent modules.

8. **Code Documentation Defects**

When the code documentation does not reflect what the program actually does, or is incomplete or ambiguous, this is called a code documentation defect. Incomplete, unclear, incorrect, and out-of-date code documentation affects testing efforts.

Testers may be misled by documentation defects and thus reuse improper tests or design new tests that are not appropriate for the code. Code reviews are the best tools to detect these types of defects.

9. **External Hardware, Software Interfaces Defects**

These defects arise from problems related to system calls, links to databases, input/output sequences, memory usage, resource usage, interrupts and exception handling, data exchanges with hardware, protocols, formats, interfaces with build files, and timing sequences (race conditions may result).

Many initialization, data flow, control, and logic defects that occur in design and code are best addressed by white box testing techniques applied at the unit (single-module) level.

iv. Testing Defects

Defects are not confined to code and its related artifacts. Test plans, test cases, test harnesses, and test procedures can also contain defects. Defects in test plans are best detected using review techniques.

1. **Test Harness Defects**

   In order to test software, especially at the unit and integration levels, auxiliary code must be developed. This is called the test harness or scaffolding code.

   The test harness code should be carefully designed, implemented, and tested since it is a work product and much of this code can be reused when new releases of the software are developed. Test harnesses are subject to the same types of code and design defects that can be found in all other types of software.

2. **Test Case Design and Test Procedure Defects**

   These would encompass incorrect, incomplete, missing, inappropriate test cases, and test procedures. These defects are again best detected in test plan reviews. Sometimes the defects are revealed during the testing process itself by means of a careful analysis of test conditions and test results. Repairs will then have to be made.

Defect Examples: The Coin Problem

The Fig shows a sample informal specification for a simple program that calculates the total monetary value of a set of coins. The program could be a component of an interactive cash register system to support retail store clerks. This simple example shows requirements/specification defects, functional description defects, and interface description defects.

A sample specification with defects:

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

The functional description defects arise because the functional description is ambiguous and incomplete. It does not state that the input, number_of_coins, and the output, number_of_dollars and number _of_cents, should all have values of zero or greater. The number_of_coins cannot be negative, and the values in dollars and cents cannot be negative in the real-world domain.

As a consequence of these ambiguities and specification incompleteness, a checking routine may be omitted from the design, allowing the final program to accept negative values for the input number_of_coins for each of the denominations, and consequently it may calculate an invalid value for the results.

A **precondition** is a condition that must be true in order for a software component to operate properly. In this case a useful precondition would be one that states for example:

number_of_coins >= 0

A **postcondition** is a condition that must be true when a software component completes its operation properly.

A useful postcondition would be:

number_of_dollars, number_of_cents >= 0.

In addition, the functional description is unclear about the largest number of coins of each denomination allowed, and the largest number of dollars and cents allowed as output values.

Interface description defects relate to the ambiguous and incomplete description of user–software interaction. It is not clear from the specification how the user interacts with the program to provide input, and how the output is to be reported. Because of ambiguities in the user interaction description the software may be difficult to use.

Likely origins for these types of specification defects lie in the nature of the development process, and lack of proper education and training. A poor-quality development process may not be allocating the proper time and resources to specification development and review. In addition, software engineers may not have the proper education and training to develop a quality specification.

All of these specification defects, if not detected and repaired, will propagate to the design and coding phases.

The below Figure shows the specification transformed into a design description. There are numerous design defects, some due to the ambiguous and incomplete nature of the specification; others are newly introduced.

Design defects include the following:

Control, logic, and sequencing defects: The defect in this subclass arises from an incorrect “while” loop condition (should be less than or equal to six)

Algorithmic, and processing defects: These arise from the lack of error checks for incorrect and/or invalid inputs, lack of a path where users can correct erroneous inputs, lack of a path for recovery from input errors.

The lack of an error check could also be counted as a functional design defect since the design does not adequately describe the proper functionality for the program.

A sample design specification code with defects

```
Design Description for Program calculate_coin_values

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 then 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
```

Data defects: This defect relates to an incorrect value for one of the elements of the integer array, coin_values, which should read 1,5,10,25,50,100.

External interface description defects: These are defects arising from the absence of input messages or prompts that introduce the program to the user and request inputs. The user has no way of knowing in which order the number of coins for each denomination must be input, and when to stop inputting values.

There is an absence of help messages, and feedback for user if he wishes to change an input or learn the correct format and order for inputting the number of coins. The output description and output formatting is incomplete.

There is no description of what the outputs means in terms of the problem domain. The user will note that two values are output, but has no clue as to their meaning.

Control, logic, and sequence defects: These include the loop variable increment step which is out of the scope of the loop. Note that incorrect loop condition (i < 6) is carried over from design and should be counted as a design defect.

Algorithmic and processing defects: The division operator may cause problems if negative values are divided, although this problem could be eliminated with an input check.

Data Flow defects: The variable total_coin_value is not initialized .It is used before it is defined. (This might also be considered a data defect.)

Data Defects: The error in initializing the array coin_values is carried over from design and should be counted as a design defect.

External Hardware, Software Interface Defects: The call to the external function “scanf” is incorrect. The address of the variable must be provided (&number_of_coins).

A code example with defects

```
/*************************************************************
program calculate_coin_values  calculates the dollar and cents
value of a set of coins of different dominations 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);
}

/*************************************************************/
```

Code Documentation Defects: The documentation that accompanies this code is incomplete and ambiguous. It reflects the deficiencies in the external interface description and other defects that occurred during specification and design. Vital information is missing for any one who will need to repair, maintain or reuse this code.

The control, logic, and sequence, data flow defects could be detected by using a combination of white and black box testing techniques. Black box tests may work well to reveal the algorithmic and data defects.

The code documentation defects require a code review for detection. The external software interface defect would probably be caught by a good compiler.

The poor quality of this small program is due to defects injected during several of the life cycle phases with probable causes ranging from lack of education, a poor process, to oversight on the part of the designers and developers. Even though it implements a simple function the program is unusable because of the nature of the defects it contains.

Developer/Tester Support for Developing a Defect Repository

It is important for a member of a test organization to illustrate to store defect information. Software engineers and test specialists should follow the examples of engineers in other disciplines who have realized the usefulness of defect data.

A requirement for repository development should be a part of testing and/or debugging policy statements. The task begins with development of a defect classification scheme and then the collection defect data from organizational projects is initiated. Forms and templates will need to be designed to collect the data.

Each defect after testing should be recorded with the frequency of occurrence for each of the defect types. Defect monitoring should continue for each on-going project. The distribution of defects will change as changes are made in processes.

The defect data is useful for test planning, a TMM level 2 maturity goal. It helps to select applicable testing techniques, design (and reuse) the test cases needed, and allocate the amount of resources needed to devote to

detecting and removing these defects. This in turn will allow to estimate testing schedules and costs.

The defect data can support debugging activities as well. The below fig shows, a defect repository supporting the achievement and continuous implementation of several TMM maturity goals including controlling and monitoring of test, software quality evaluation and control, test measurement, and test process improvement

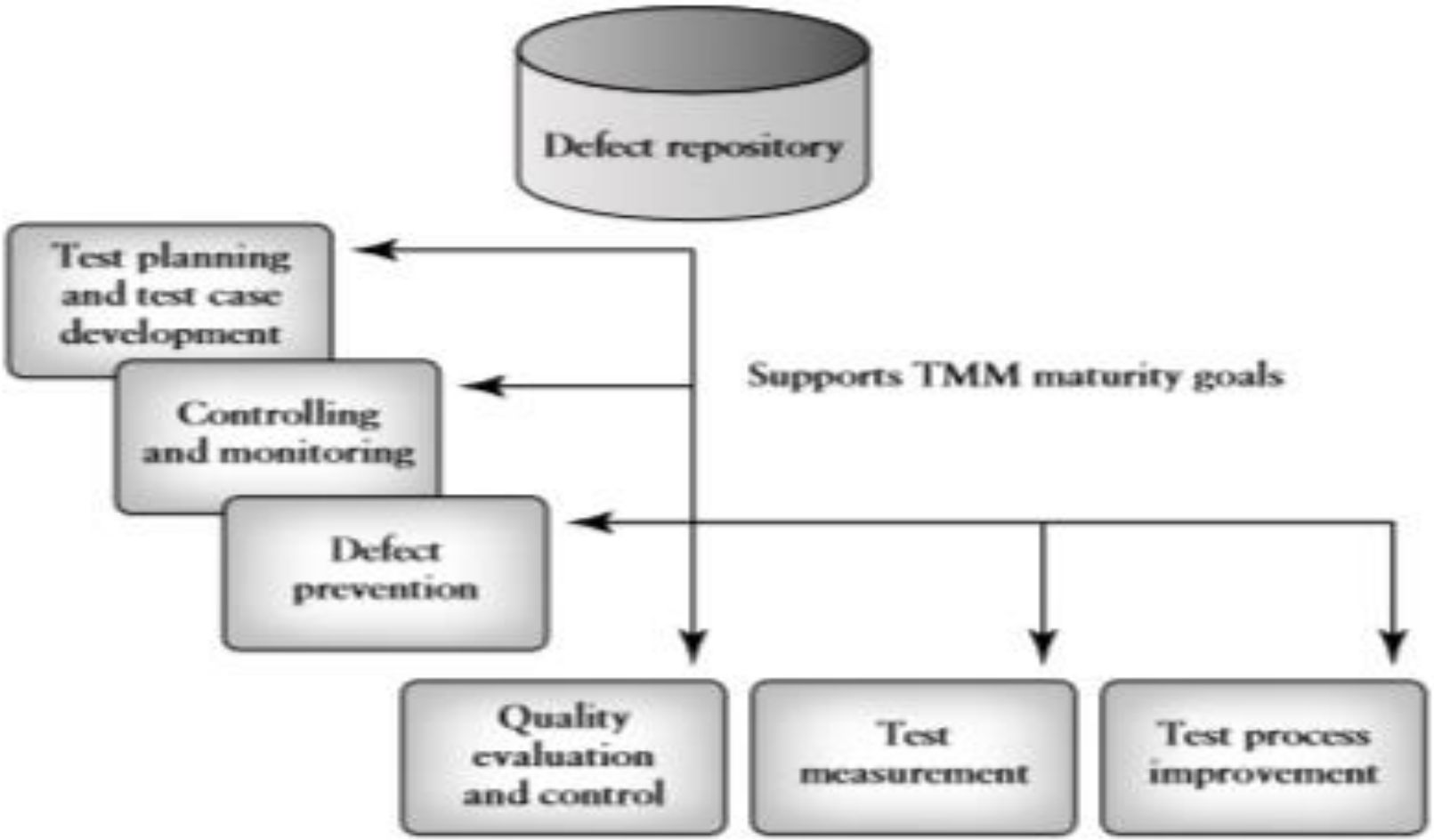


FIG. 3.6

The defect repository, and support for TMM maturity goals.

Defect Analysis and Prevention

Processes and Defects

Testers have an additional set of defect- related goals. These goals are to:

• analyze defects to find their root causes;

• take actions and make changes

—in our overall development processes;

—in our testing process;

• prevent defects from reoccurring.

- Defect analysis and defect prevention are activities that are of increasing importance as the software we develop becomes more complex and has greater and greater impact on our safely, health, and financial well- being.

- The goal of defect prevention is not entirely new, nor is it strictly limited to the software industry as the reader will note in the next section.

- An interesting point that should be made here is that carrying out the goal of defect prevention to its limit, in theory, could lead to defect-free software. Some argue that the latter is a feasible goal for software engineers.

- However, given the case that software is designed, developed, and tested by humans who under the best of circumstances may introduce defects, a more practical objective would be to improve our processes so that we can develop software with a very low defect content.

- The emphasis should be on process because the nature of the development/test process, the tools and techniques used, the quality of the staff—all have a great impact on the defect content.

- Given these circumstances, we should carefully note that testers alone are not able to produce, nor are they solely responsible for producing, very low defect software.

As far as the own testing process is concerned, we should aim for identification of weaknesses that allow:

- defects to be injected into our test work products;
- defects to escape from our defect-filtering activities and propagate into the software product.

History of Defect Analysis and Prevention

Defect analysis/prevention processes help to reduce the costs of developing and maintaining software by reducing the number of defects that require our attention in both review and execution-based testing activities.

Defect localization and repair are in the main unplanned activities.

Defect analysis/prevention processes help to improve software quality. If our software contains fewer defects, this reduces the total number of problems we must look for; the sheer volume of problems we need to address may be significantly smaller.

- If we can eliminate noncritical (trivial) defects, then we as testers are in a better position to apply our time and expertise to detecting the fewer, but perhaps more serious, defects.

- This, in turn, could increase the effectiveness of our testing process. Defect analysis/prevention processes provide a framework for overall process improvement activities. They can serve as drivers for process improvement.

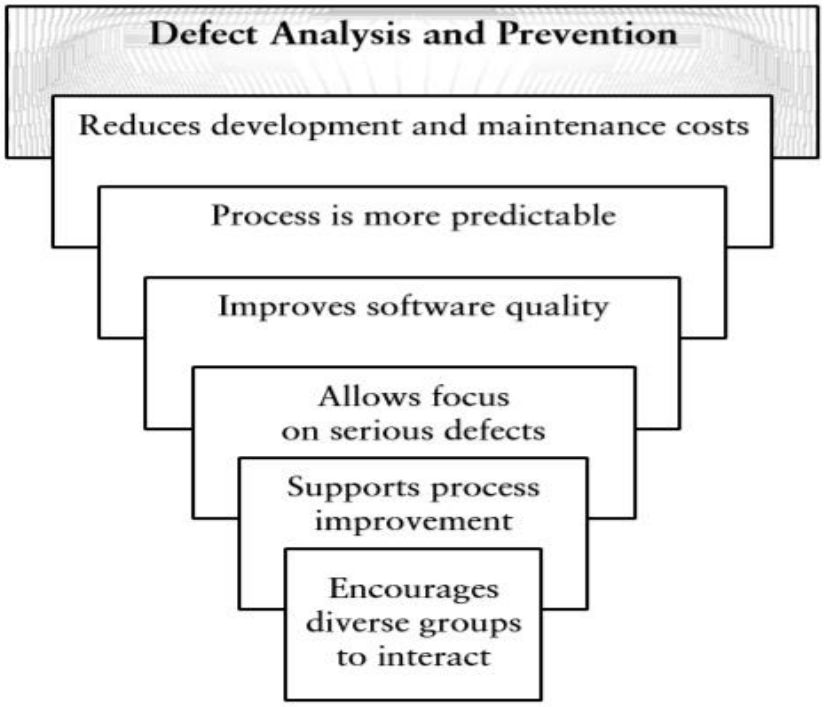
- Defect analysis/prevention activities not only help to fine-tune an organizations' current process and practices, but also support the identification and implementation of new methods and tools so that current process continues to evolve and comes closer to being optimized.

- Defect analysis/prevention activities encourage interaction between a diverse number of staff members, for example, project managers, developers, testers, and SQA staff.

- The close interrelationships between specialized group activities and the quality of internal and external deliverables becomes more apparent.

- The success of a defect prevention program depends on cooperation between, and the participation of, members of these groups.

Benefits of defect analysis and Prevention Process



Necessary Support for a Defect Prevention Program

- When an organization decides to implement a defect prevention program it needs to have attained a certain level of process maturity in the sense that there should be an infrastructure in place consisting of polices, goals, staff, methods, tools, measurements, and organizational structures to support the program.

- CMM has a Key Process Area called “defect prevention,” which is at its highest maturity level.

- Placing this goal at the highest maturity levels of the models ensures that the infrastructure described above is in place, and that gives the program the necessary support to become a success.

- An organization also must have managerial commitment to successfully implement a defect prevention program.

- Management must provide leadership, resources, and support cultural changes. An additional precondition is the existence of a defect repository and a defect (problem) reporting system.

- You cannot prevent defects if you do not consistently classify, count, and record them as they occur.

- In this way you know they exist, where, and when they occur, and their frequency of occurrence.

There are several other essential elements necessary to implement a defect prevention program. These are:

1. A training program to prepare staff members for defect analysis and defect prevention activities (supported by TMM level 3 maturity goal). This includes training in defect causal analysis techniques and use of statistical analysis tools.

2. A defect causal analysis process where defects from all projects are analyzed using the techniques described in the next section. Process goals are to identify the mechanisms by which each particular defect is injected into a software deliverable. When the cause is identified, preventive actions (process changes) can be suggested.

3. Action teams to implement and oversee the suggested process changes as applied to pilot projects.

4. A tracking system to monitor the process changes, and provide feed-back on the usefulness of the changes, and their impact on software quality.

5. A technology transfer, or process improvement group, that will ensure that defect prevention becomes a standard set of practices and that process changes to support defect prevention are implemented throughout the organization (supported by the TMM level 5 maturity goal, “Test process optimization”).

Defect Causal Analysis

- Defect causal analysis is a component of what we call a defect prevention program.

- Action planning, action implementation, tracking, and feed back, as well as process evolution, are the other components of this program.

- A team or task force should be established to initiate and oversee a defect prevention program.

- The team members should be trained and motivated. Testers, developers, project managers, SQA staff, and process improvement group members are good candidates to carry out the tasks and responsibilities essential for initiation of the program.

- The team is responsible for developing policies and goals that relate to defect analysis and prevention. These policies/goals should be documented and made available to appropriate parties.

- Defect causal categories should be established as part of the policy statement.

- Two separate groups that are actually involved in implementing a defect prevention program. These are (i) the causal analysis group, and (ii) the action planning/tracking team.

- The mechanism for implementing defect causal analysis is usually a meeting. Attendees, need to bring all necessary items such as problem reports, test logs, defect fix reports, defect repository reports, and specially requested designer/developer reports.

- The agenda for the meetings focus on

(i) finding a defect cause or origin for each type of defect,

(ii) exploring and developing process change suggestions, or actions, for preventing the reoccurrence of each defect type in future projects, and

(iii) establishing priorities for the actions that are proposed.

- Humphrey suggests that causal analysis meetings should be held at various times both before and after the release of the software.

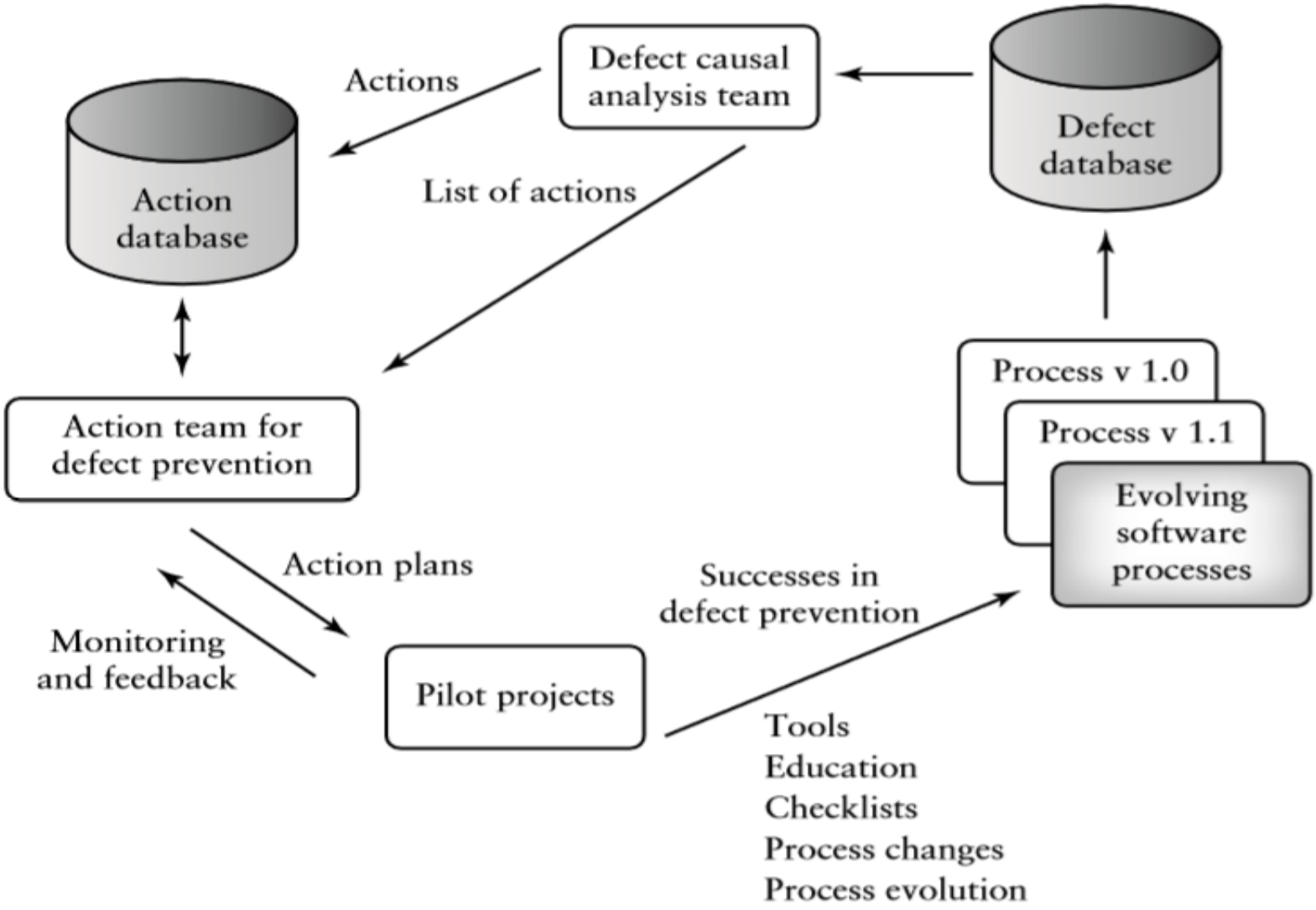
(i) shortly after the detailed design stage is complete for all system modules (analyze problems/defects found by reviews);

(ii) shortly after each module has completed a test phase or level of testing(analyze defects found at that level of test);

(iii) after release of the software, and there are a reasonable number of problems or defects reported in operation;

(iv) on an annual basis after the software has been released even if the number of problems/defect reported is relatively low.

Defect Prevention Process



Defect causal analysis meetings on two levels:

1. Defect analysis on the general process level.
2. Defect analysis for defects confined to the testing process.

They are particularly interested in identifying, finding causes for, and preventing defects that occur during testing.

The Action Team: Making Process Changes

- When the defect causal analysis team has completed its work, the action team now initiates the preventative actions.

- The action team should be organized so there is a manager or leader.

- This person is often a member of a process improvement team, and plays a leadership role on that team.

- If the causal analysis team has been focusing on test defects, then an experienced test manager should play a leading role on the action team since it is likely there will be changes in the testing process.

The action team members need specialized skills. Using these skills, team members fulfil roles such as:

- **Tool expert:** Team member who can develop tools, or is familiar with procedures to evaluate and acquire tools from outside the organization.

- **Education coordinator:** Member of the training group who can educate/train staff with respect to aspects of the new process.

- **Technical writer/communicator:** Person who can prepare the necessary documentation for the newly changed process, prepare reports for managers, develop articles for the organizational newsletter on action-related topics, and make presentations to ensure

(a) Visibility for the implemented actions in the organization and

(b) that all key staff contributing staff are recognized.

- **Planner**: Team member who can develop plans for the actions to be implemented.

- **Measurement expert:** Team member who can select appropriate measurements for monitoring the changed process, and who can apply measurement results to evaluate the changes.

Benefits of a Defect Prevention Program

- A defect prevention program provides a nucleus for process changes. It can be represented as a series of reoccurring cycles that support continuous process improvement.
- Defect prevention involves many diverse staff members who have the opportunity to provide input for the process changes.
- It can be applied to any process in any organization, hence its prominent role in the TMM at level 5.
- Defect prevention programs do have associated costs.
- These include costs of defect analysis and costs of the action plans as applied to projects. However, finding and repairing defects usually has larger costs.

- Appraisal costs—which include testing, reviewing, and inspecting—and
- failure costs—which include rework, complaint resolution, and warranty work—usually cost much more than defect prevention activities.

- Defect prevention activities should lower appraisal and failure costs.

- Beside the benefits of possible cost reductions in the appraisal and failure areas of quality there are also other benefits for a defection prevention program which include:
- a more aware and motivated staff;
- a more satisfied customer;
- a more reliable, manageable and predictable process;
- cultural changes that bring quality issues in focus;
- a nucleus for continuous process improvement.

Defect Prevention and the Three Critical Views

Managers Role

- Managers should ensure that a defect prevention team is well staffed, and has appropriate resources and organizational support.
- Managers participate and serve as leaders in defect prevention activities such as action planning and monitoring. They are the leaders in project kick-off meetings.
- Managers should promote discussion and distribute lists of common defects and process change information to project team members.
- They should also promote the inclusion of defect prevention activities as part of the project/test plans, and follow-up to ensure that all approved process changes are reflected in process documents and standards.

Testers Role

- The tester’s role includes the collection of defect data and entry of the data into the defect database.
- Defect data records must be updated as a result of the meeting and action plans.

- Testers also form causal analysis teams that specially address test-related defects.
- Where appropriate, testers go to training classes to learn causal analysis and defect prevention techniques such as the preparation of Pareto and fishbone diagrams.
- They are especially responsible for planning and implementing actions that make changes in the test process.
- They monitor the changes to the test process as applied to pilot projects, and report results. They serve as ambassadors for cultural changes.
- The tester’s role includes the collection of defect data and entry of the data into the defect database.
- Defect data records must be updated as a result of the meeting and action plans.
- Testers also form causal analysis teams that specially address test-related defects.
- Where appropriate, testers go to training classes to learn causal analysis and defect prevention techniques such as the preparation of Pareto and fishbone diagrams.

- They are especially responsible for planning and implementing actions that make changes in the test process.
- They monitor the changes to the test process as applied to pilot projects, and report results. They serve as ambassadors for cultural changes.

Users/Clients Role

- Users/clients have a limited role in a defect prevention program.
- Their role is confined to reporting defects and problems in operating software so these can be entered into the defect database for causal analysis.