Software Reliability

- Software reliability,
- SEI CMM,
- Characteristics of software maintenance,
- software reverse engineering,
- software reengineering.

Software Reliability

- Reliability of a software product essentially denotes its trustworthiness or dependability.

- Reliability of a software product can also be defined as the probability of the product working “correctly” over a given period of time.

- Experimentally observed by analyzing the behavior of a large number of programs that 90% of the execution time of a typical program is spent in executing only 10% of the instructions in the program.

- These most used 10% instructions are often called the **core** of the program.

- The rest 90% of the program statements are called **non-core** and are executed only for 10% of the total execution time.

Reasons for software reliability being difficult to measure

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is highly observer dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

Hardware reliability vs. software reliability differ

- Hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part.

- Software product would continue to fail until the error is tracked down and either the design or the code is changed.

- For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred;

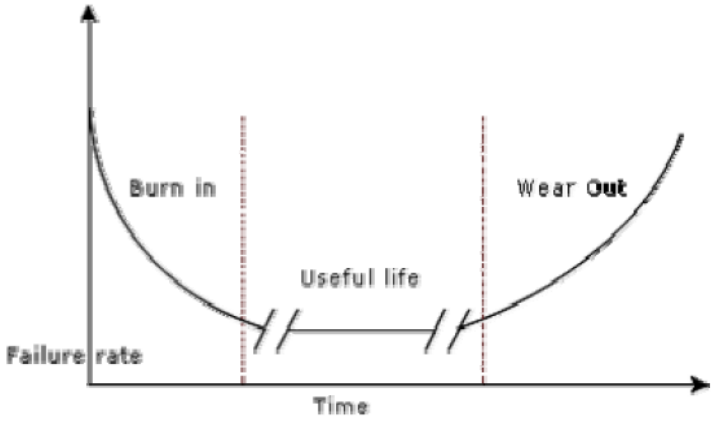
- Whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors).

- **Hardware reliability study is concerned with stability (for example, inter-failure**
- **times remain constant).**

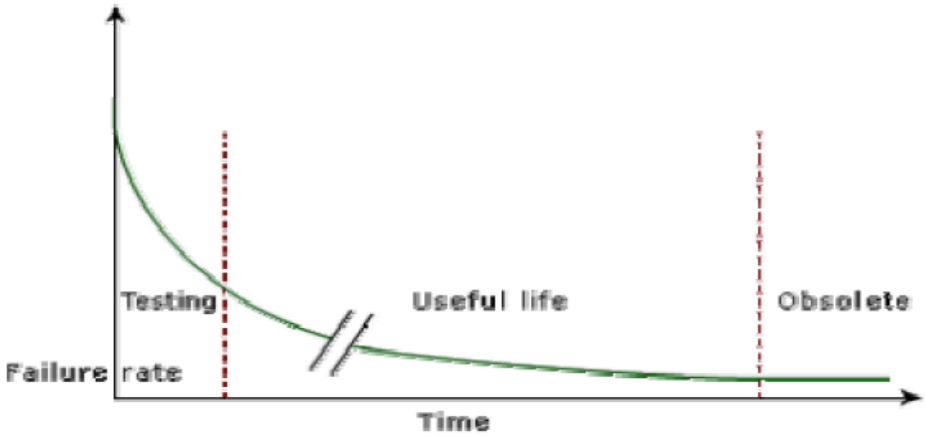
- **Software reliability study aims at reliability growth (i.e. inter-failure times increase).**

Failure rate over the product lifetime

- For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed.
- The system then enters its useful life. After some time (called product life time) the components wear out, and the failure rate increases.
- For software the failure rate is at it's highest during integration and test. As the system is tested, more and more errors are identified and removed resulting in reduced failure rate.
- This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no error corrections occurs and the failure rate remains unchanged.



(a) Hardware product



(b) Software product

Reliability metrics

Six reliability metrics which can be used to quantify the reliability of software products.

- **Rate of occurrence of failure (ROCOF).** ROCOF measures the frequency of occurrence of unexpected behavior (i.e. failures).
  - ROCOF measure of a software product can be obtained by observing the behavior of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.

- **Mean Time To Failure (MTTF).** MTTF is the average time between two successive failures, observed over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failu the time instants *t1, t2, …, tn. Then, MTTF can be* calculated: $\sum_{i=1}^{n} \frac{t_{i+1} - t_i}{(n-1)}$.

- **Mean Time To Repair (MTTR).** Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.

- **Mean Time Between Failure (MTBF).** MTTF and MTTR can be combined to get the MTBF metric: MTBF = MTTF + MTTR.
- Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours.

- **Probability of Failure on Demand (POFOD):** This metric does not explicitly involve time. Measures the likelihood of the system failing:
  - when a service request is made.
  - POFOD of 0.001 means: 1 out of 1000 service requests may result in a failure.

- **Availability.** Availability of a system is a measure of how likely shall the system be available for use over a given period of time.

  This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs.

Classification of software failures

- **Transient.** Transient failures occur only for certain input values while invoking a function of the system.

- **Permanent.** Permanent failures occur for all input values while invoking a function of the system.

- **Recoverable.** When recoverable failures occur, the system recovers
- with or without operator intervention.

- **Unrecoverable.** In unrecoverable failures, the system may need to be restarted.

- **Cosmetic.** These classes of failures cause only minor irritations, and do not lead to incorrect results.

- An example of a cosmetic failure is the case where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface.

SEI CMM

- SEI Capability Maturity Model helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.

- SEI CMM classifies software development industries into the following five maturity levels.
- **Level 1: Initial.** A software development organization at this level is characterized by ad hoc activities. Very few or no processes are defined and followed.

- **Level 2: Repeatable.** At this level, the basic project management practices such as tracking cost and schedule are established. Size and cost estimation techniques like function point analysis, COCOMO, etc. are used.

- **Level 3: Defined.** At this level the processes for both management and development activities are defined and documented.

- **Level 4: Managed.** At this level, the focus is on software metrics. Two types of metrics are collected.
  - Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc.

  - Process metrics reflect the effectiveness of the process being used, such as average defect correction time, productivity, average number of defects found per hour inspection, average number of failures detected during testing per LOC, etc.

- **Level 5: Optimizing.** At this stage, process and product metrics are collected. Process and product measurement data are analyzed for continuous process improvement.
  - For example, if from an analysis of the process measurement results, it was found that the code reviews were not very effective and a large number of errors were detected only during the unit testing, then the process may be fine tuned to make the review more effective.

Characteristics of software maintenance

- **Corrective:** Corrective maintenance of a software product is necessary to rectify the bugs observed while the system is in use.

- **Adaptive:** A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.

- **Perfective:** A software product needs maintenance to support the new features that users want it to support, to change different functionalities of

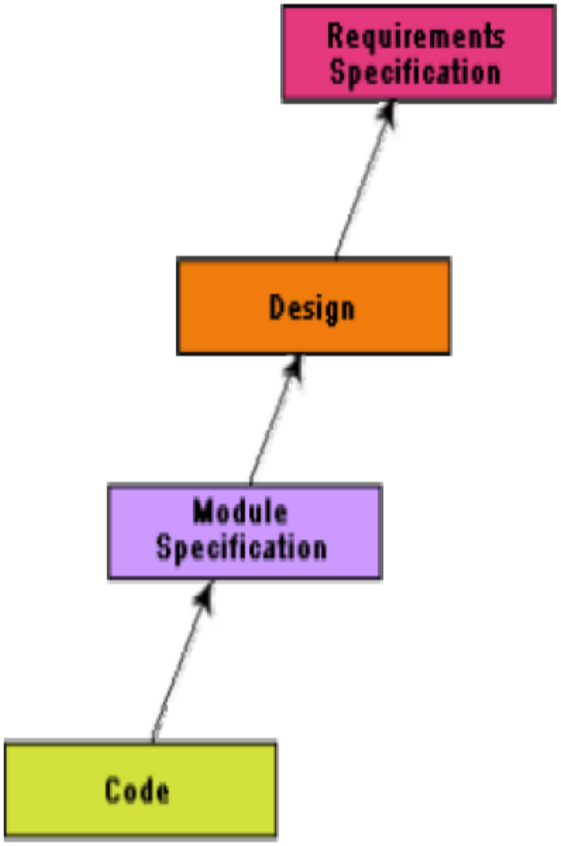
Software reverse engineering

- Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code.

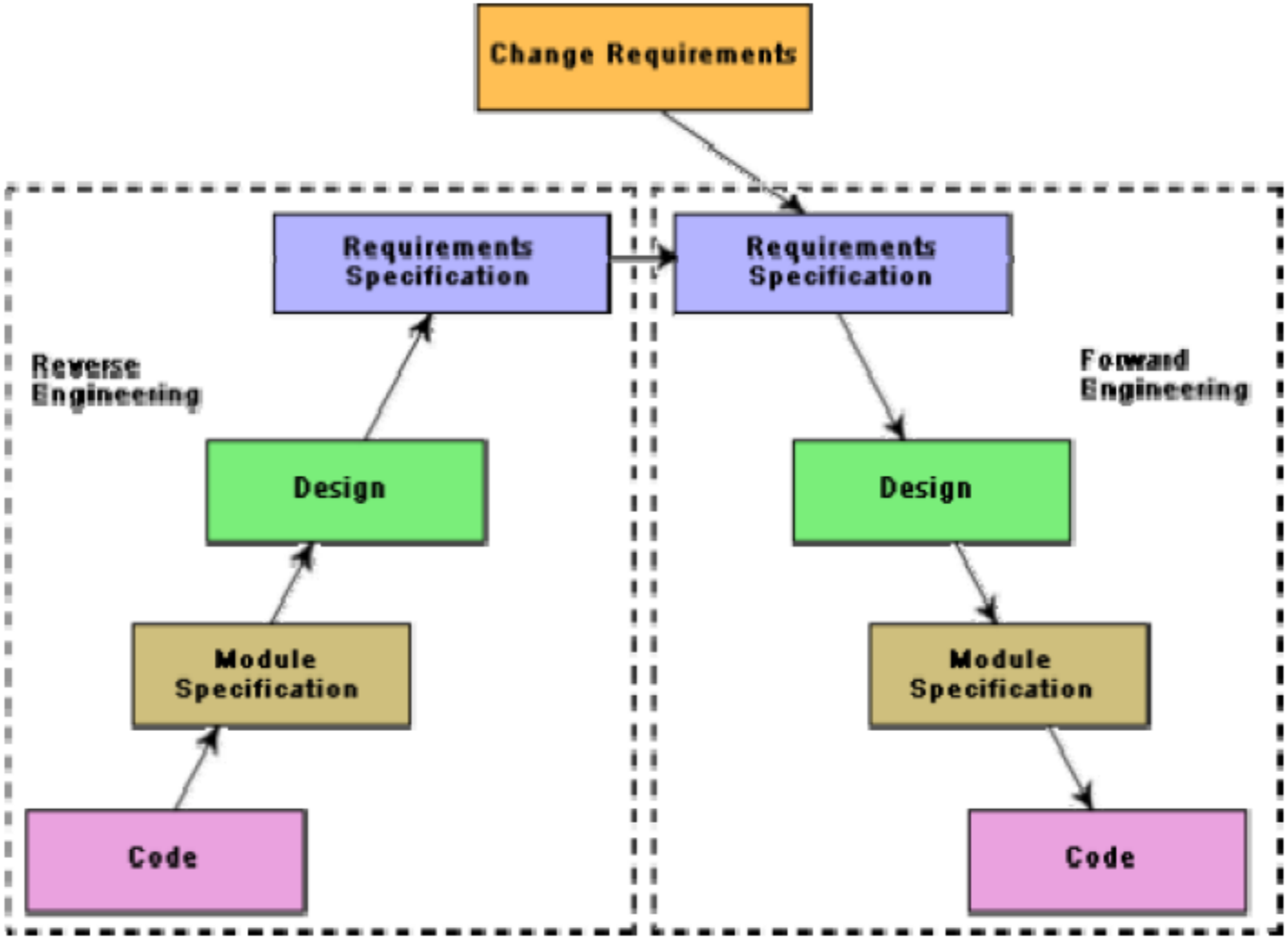
- The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.

- Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured.

- Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.



- Software reengineering is a combination of two consecutive processes i.e. software reverse engineering and software forward engineering



What is software reuse?

Software reuse is the process of implementing or updating software systems using existing software assets.

- The systematic development of reusable components
- The systematic reuse of these components as building blocks to create new system

The advantages of reuse

- Increase software productivity
- Shorten software development time
- Improve software system interoperability
- Develop software with fewer people
- Move personnel more easily from project to project
- Reduce software development and maintenance costs
- Produce more standardized software
- Produce better quality software and provide a powerful competitive advantage

What is reusable?

- Application system
- Subsystem
- Component
- Module
- Object
- Function or Procedure

Difference between defect, error, bug, failure and fault:

"A mistake in coding is called error ,error found by tester is called defect, defect accepted by development team then it is called bug ,build does not meet the requirements then it Is failure."

- **Error:** A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. This can be a misunderstanding of the internal state of the software, an oversight in terms of memory management, confusion about the proper way to calculate a value, etc.

- **Failure:** The inability of a system or component to perform its required functions within specified performance requirements. See: bug, crash, exception, and fault.

- **Bug**: A fault in a program which causes the program to perform in an unintended or unanticipated manner. See: anomaly, defect, error, exception, and fault. Bug is terminology of Tester.

- **Fault:** An incorrect step, process, or data definition in a computer program which causes the program to perform in an unintended or unanticipated manner. See: bug, defect, error, exception.

- **Defect:** Commonly refers to several troubles with the software products, with its external behavior or with its internal features.

99% Complete Syndrome

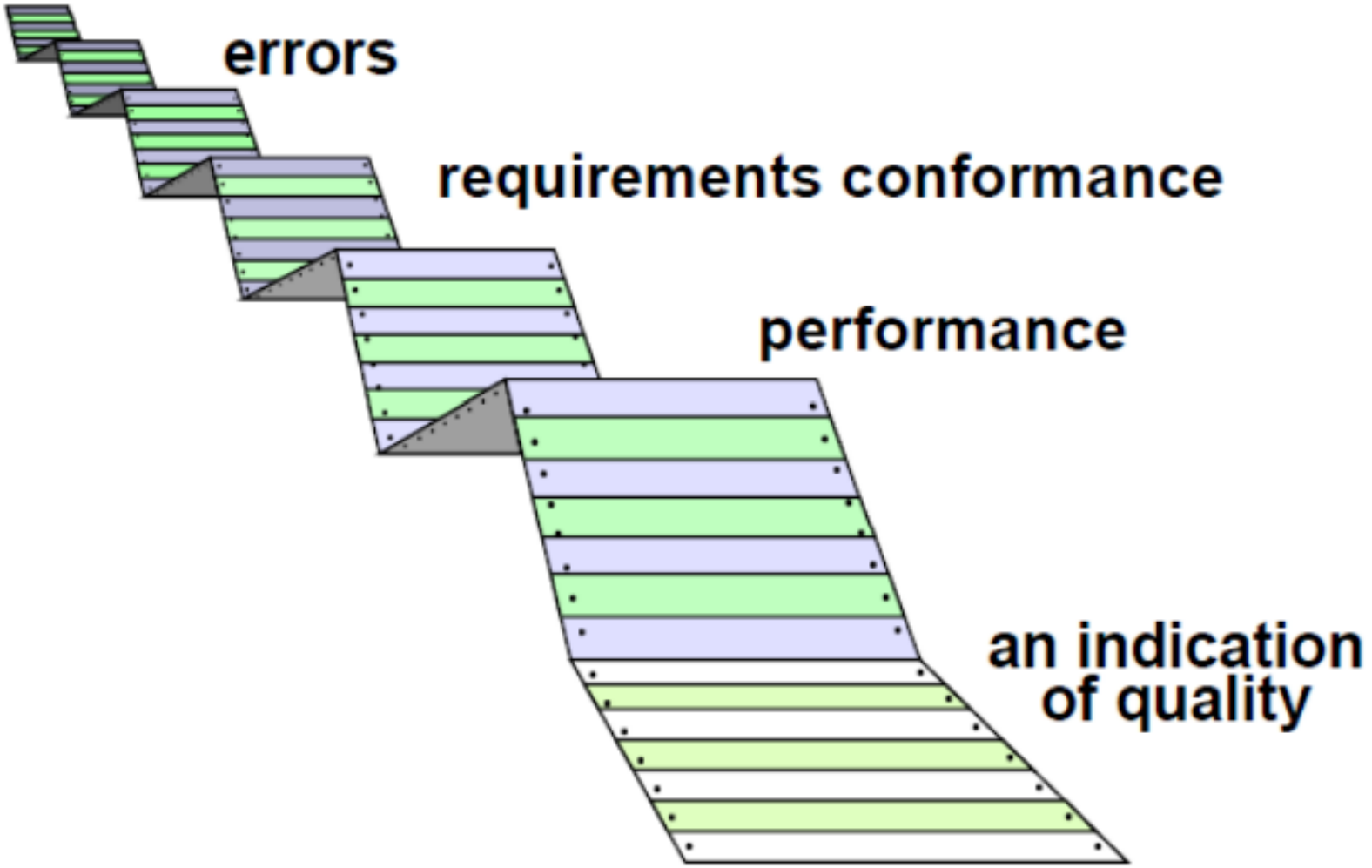
- Last 1% takes longer than the completed 99%. There are many examples to support this statement. For example progress bars we see on computers, the last period last minute, last minute of a match etc. The 90+ percent it runs fast and rest is sluggish.

- In software development if you do not follow SDLC (Software Development Life Cycle) 99% complete syndrome happens. If you ask developer about the progress he/she may say development is in final stage of just 1% is left .

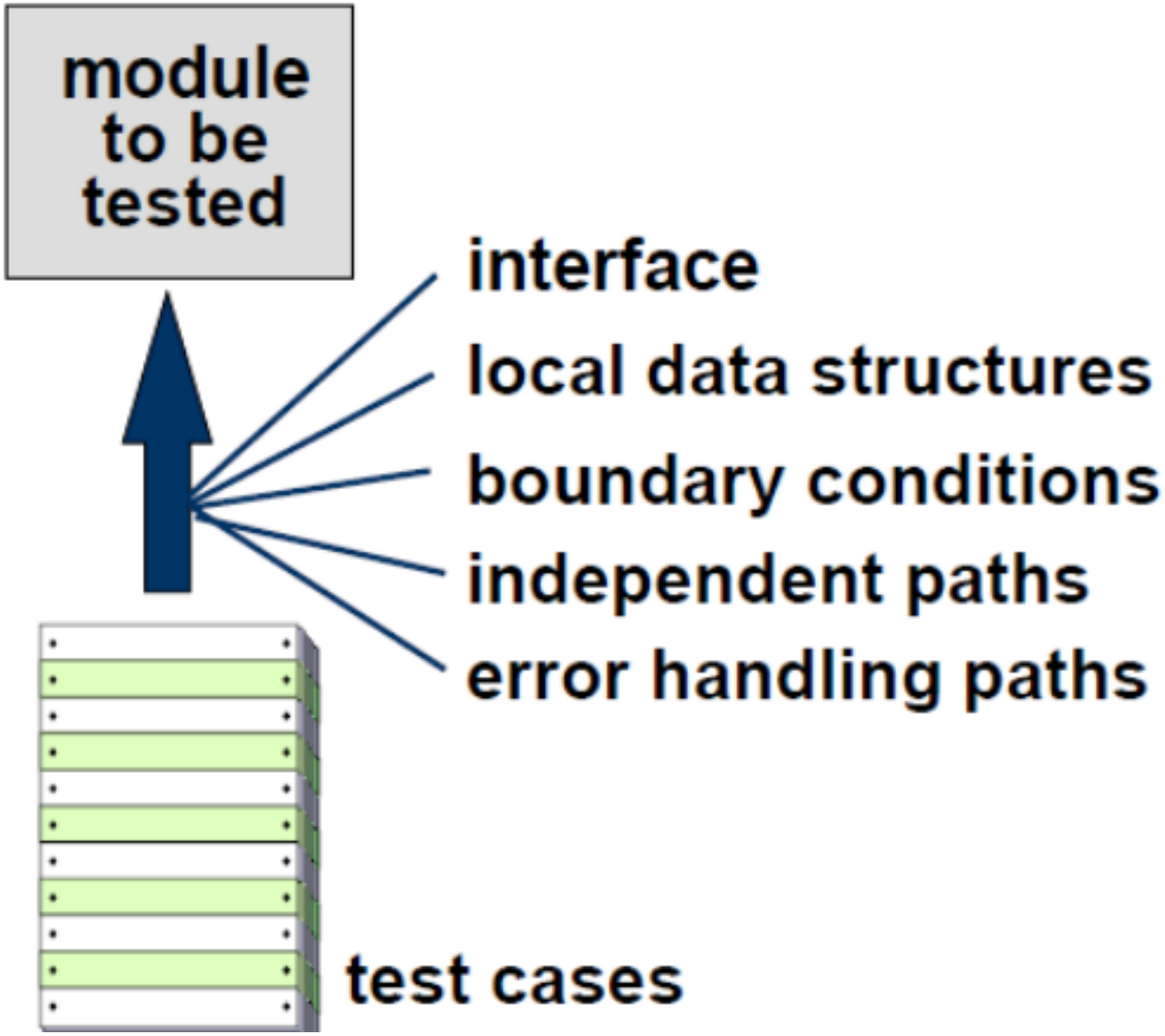
- Most probably, this last 1% will drag on to days, weeks, and even few months. Either something was wrong in the SDLC or they failed to follow up.

- Developers should evaluate the amount of effort needed to complete remaining tasks in hour. Usually instead of this people tend to consider the amount of work gone through against the original estimate.

What Testing Shows

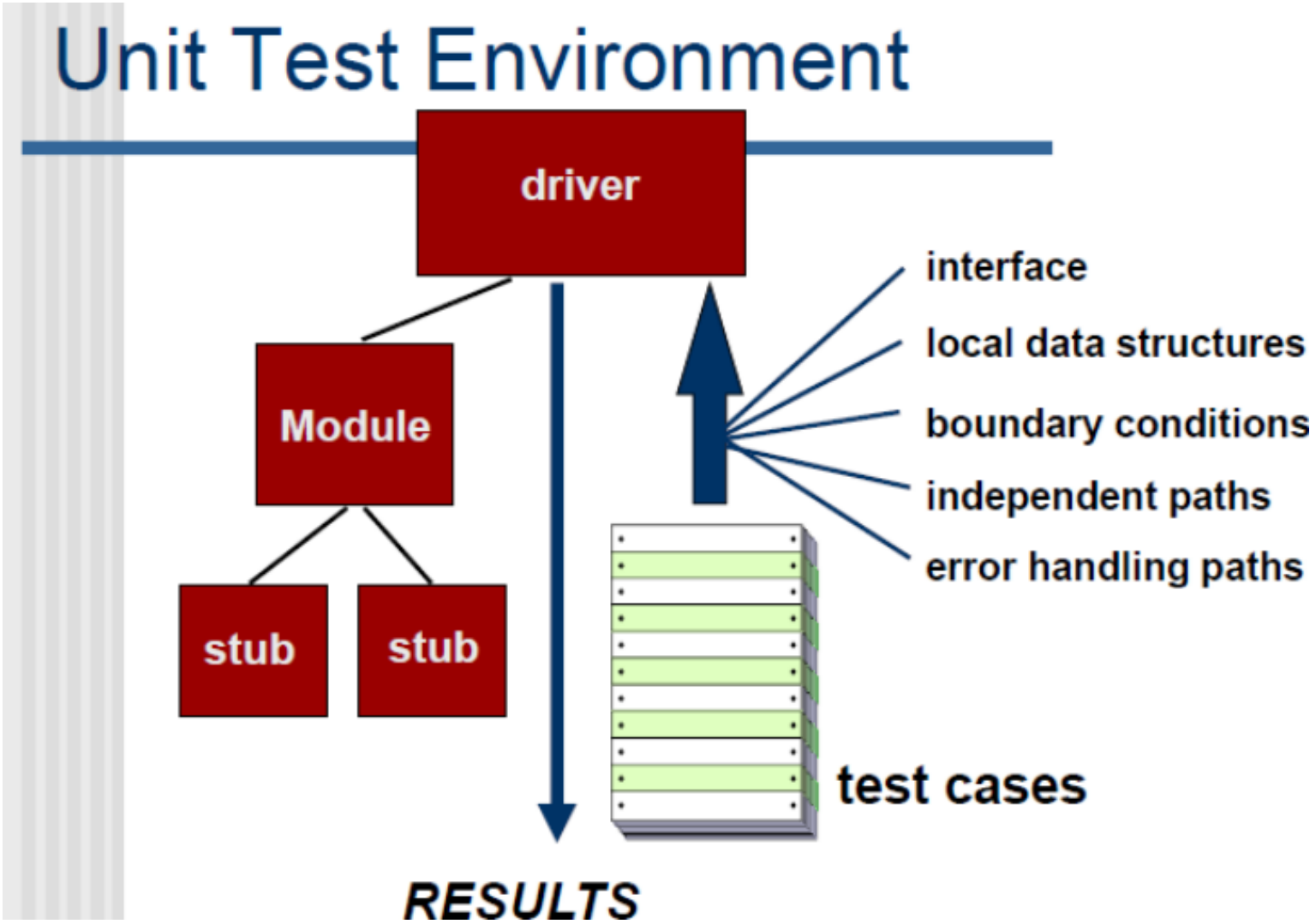


Unit Testing



- The **module** interface is tested to ensure that **information properly flows into and out of the program unit under test**.
- **Local data structures** are examined to ensure that **data stored** temporarily maintains its **integrity** during all steps in an **algorithm's execution**.
- All **independent paths** through the **control structure** are exercised to ensure that all statements in a module have been executed at least once.
- **Boundary conditions** are tested to ensure that the module operates properly at **boundaries** established to **limit or restrict processing.**
- Finally, all **error-handling paths** are tested

Unit testing is simplified when a component with high cohesion is designed.



Stubs and Drivers

- They do not implement the entire programming logic of the software module but they **simulate data communication with the calling module** while testing.
- **Stub:** Is called by the **Module under Test**.
- **Driver: Calls the Module** to be tested.

Integration Testing

- **Integration testing** is a systematic technique for constructing the **software architecture** while at the same time **conducting tests to uncover errors associated with interfacing**.

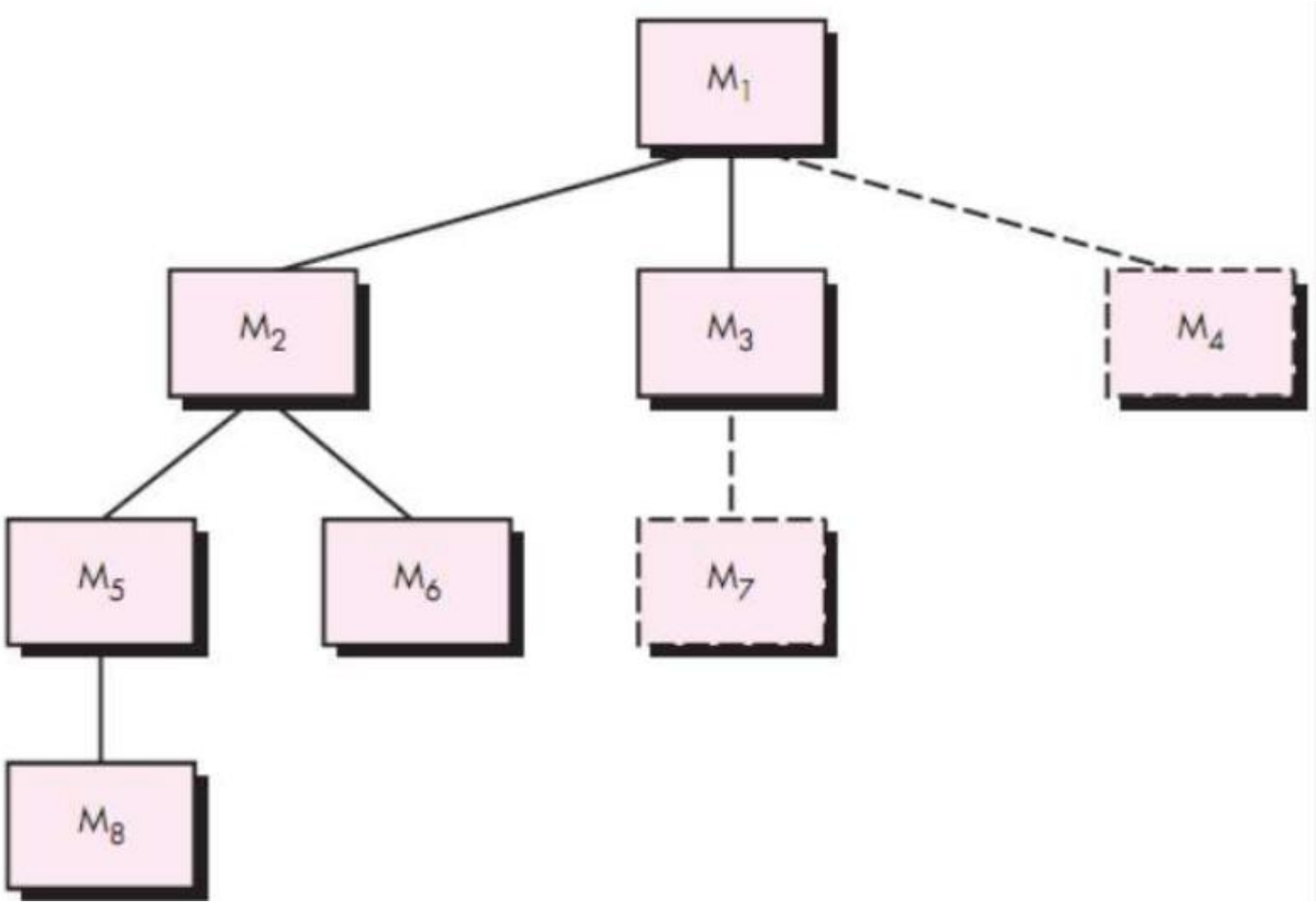
Different Integration Testing Strategies :

- Top-down testing
- Bottom-up testing
- Regression Testing
- Smoke Testing

Top-down testing

- Top-down integration testing is an **incremental approach** to construction of the software architecture.
- Modules are integrated by **moving downward through the control hierarchy,** beginning with the main control module (main program).
- Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a **depth-first or breadth-first manner.**

Top-down testing



Integration Testing

The integration process is performed in a series of five steps 1.

1. The **main control** module is used as a **test driver and stubs** are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), **subordinate stubs** are replaced one at a time with actual components.
3. **Tests** are conducted as each component is integrated.
4. On completion of each set of **tests**, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced.

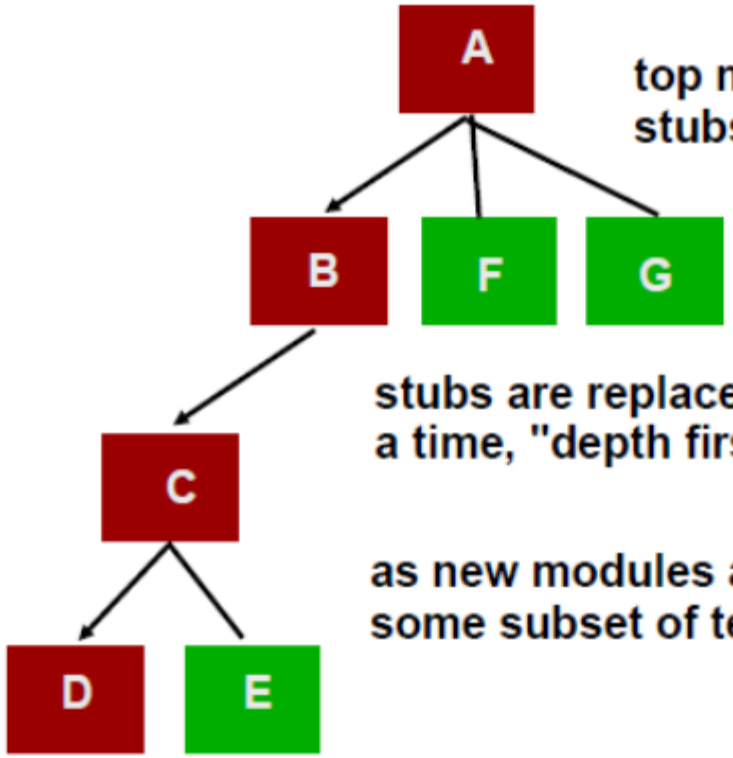
The process continues from step 2 until the entire program structure is built.

Problem encountered during top-down integration testing

- Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise.
- The most common of these problems occurs when processing at **low levels in the hierarchy is required to adequately test upper levels.**
- Stubs replace low-level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure.
- As a tester, you are left with three choices:

1. **delay many tests until stubs are replaced** with actual modules,
2. **develop stubs that perform limited functions** that simulate the actual module, or
3. **integrate the software from the bottom** of the hierarchy upward.

Top Down Integration



top module is tested with stubs

stubs are replaced one at a time, "depth first"

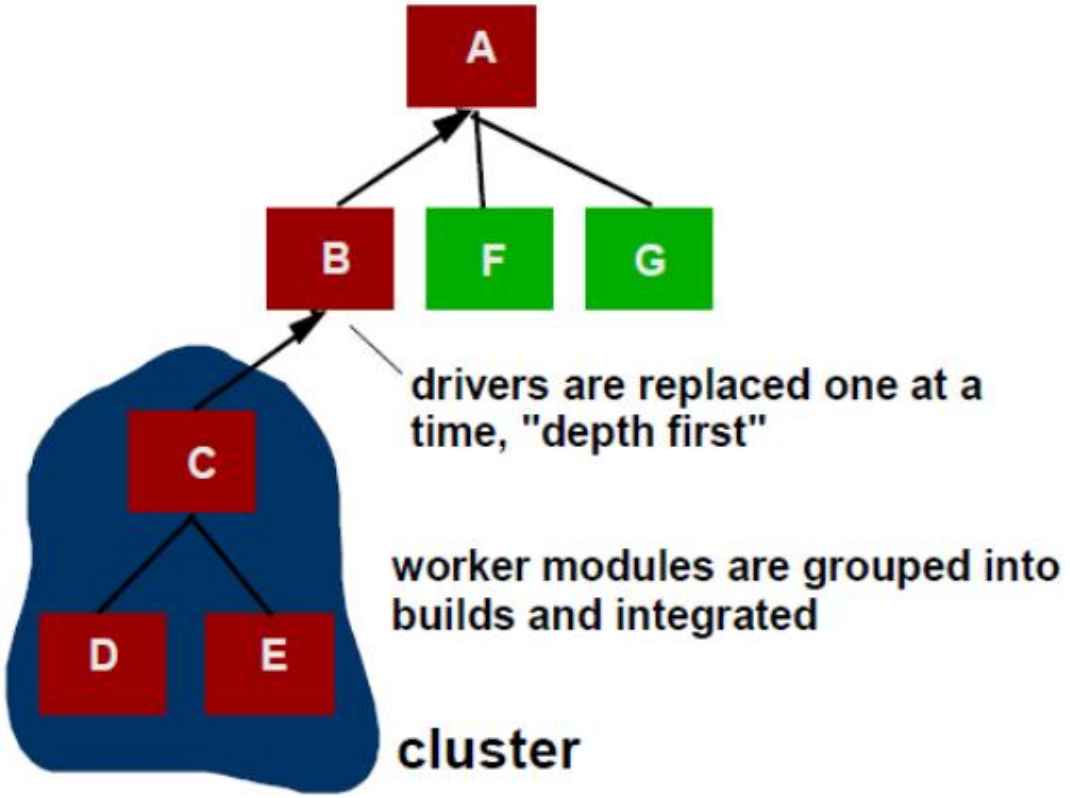
as new modules are integrated, some subset of tests is re-run

Bottom-up Integration Testing

- Bottom-up integration testing, It begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).
- Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.
- A bottom-up integration strategy may be implemented with the following steps…

1. Low-level components are combined into clusters (**sometimes called builds**) that perform a specific software sub function.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

Bottom-Up Integration



Smoke Testing

- A common approach for creating **“daily builds”** for product Software
- Smoke testing steps:
  - Software components that have been translated into code are integrated into a “build.”
  - **A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.**
  - A series of tests is designed to expose errors that will keep the build from properly performing its function.
  - The intent should be to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
  - The integration approach may be top down or bottom up.

System Testing

- System testing is a series of different tests whose purpose is to fully exercise the computer based system
- Recovery testing

✓ Tests for recovery from system faults

✓ Forces the software to fail in a variety of ways and verifies that recovery is properly performed

- Security testing

  Verifies that protection mechanisms built into a system will, in fact, protect it from improper access

- Stress testing

  Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume

System Testing cont'd….

- Performance testing
  – Tests the run-time performance of software within the context of an integrated system
  – Often coupled with stress testing and usually requires both hardware and software instrumentation
  – Can uncover situations that lead to degradation and possible system failure
- Deployment testing
  – Also known as configuration testing
  – It examines all installations procedures that will be used by customers