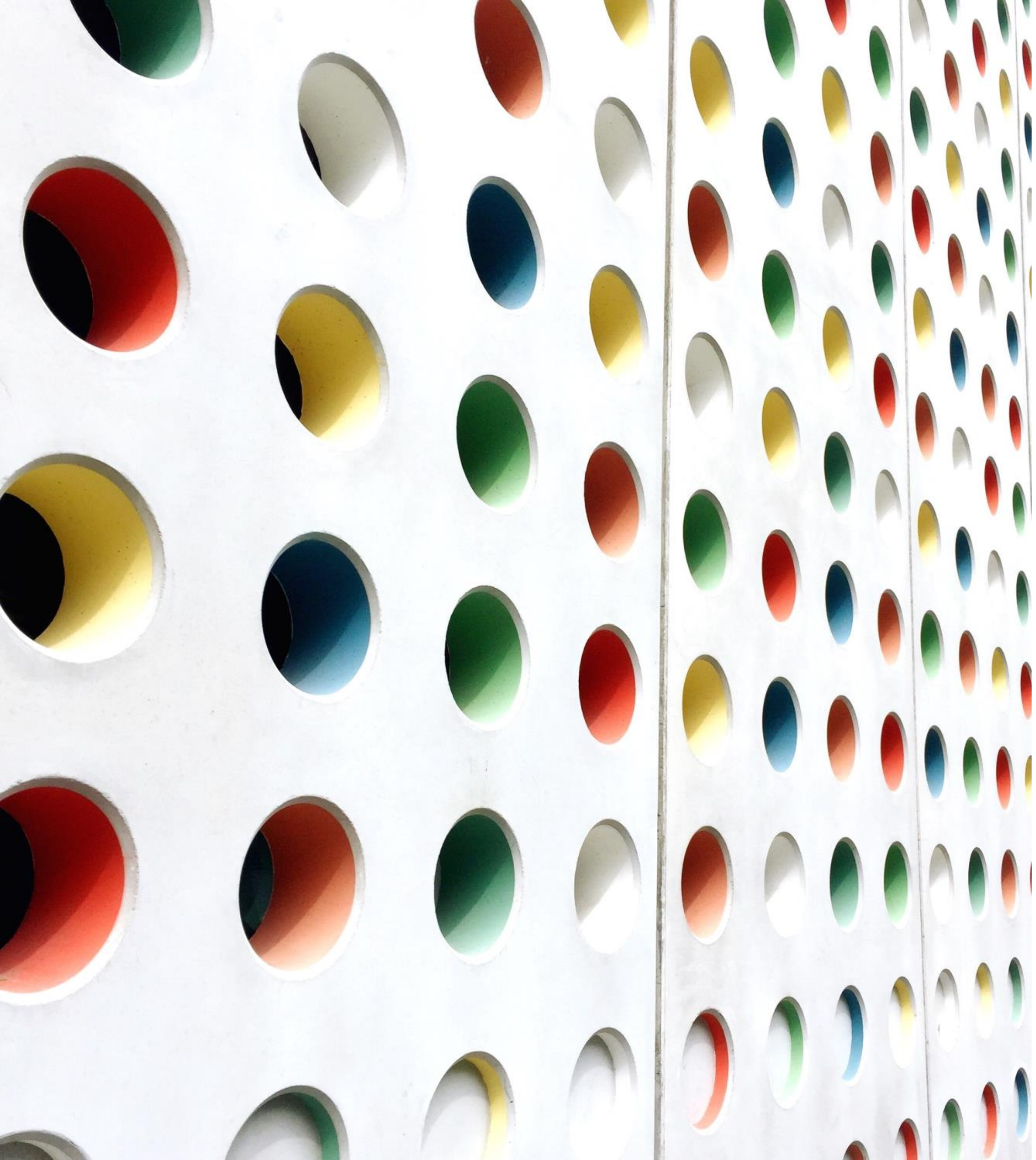


Software Testing

YESODA BHARGAVA

Introduction

1. Objective of testing a system : to identify all defects existing in the software, remove them, and achieve error-free operation.
2. Testing is vital for the success of a system.
3. Inadequate testing or non-testing leads to errors which may not appears until months later.
4. This creates two problems:
   1. The time lag between the cause and the appearance of the problem (the longer the time interval, the more complicated the problem has become)
   2. The effect of the system errors on files and records within the system.

Some examples of software failures in history

For over 2 years **Nissan** recalled over a million cars, thanks to a software glitch in the airbag sensory detectors.

In September 2014, **Apple** faced an embarrassment after it had to pull out its new iOS software update only after a few hours of its release

In February 2014, **Toyota Motor** recalled 1.9 million newest-generation Prius vehicles worldwide due to a programming error that caused the car's gas-electric hybrid systems to shut down.

Big Billion Day Sale **Flipkart** 2014.

Software failures in the US cost the economy $1.1 trillion in assets in 2016. What's more, they impacted 4.4 billion customers.

In 1996, a software bug in the bank software application caused the bank accounts of 823 customers of a major US bank to be credited with 920 million US dollars.

Objective of System Testing

I. Small system error can explode into a much larger problem.

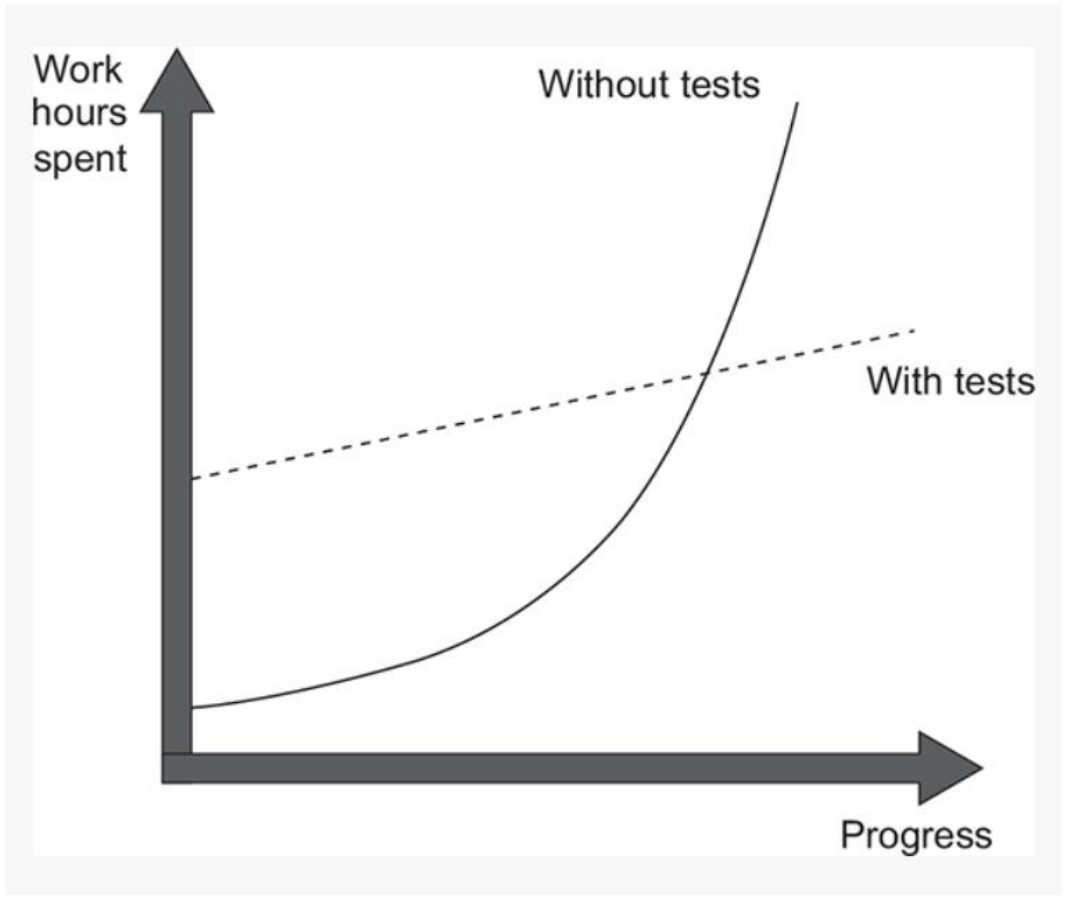
II. Effective testing early in the process translates directly into long-term cost savings from a reduced number of errors.

III. Important for ensuring customer satisfaction and reliability with the product in the long term.

IV. To point out defects and errors made during the software implementation or development phase.

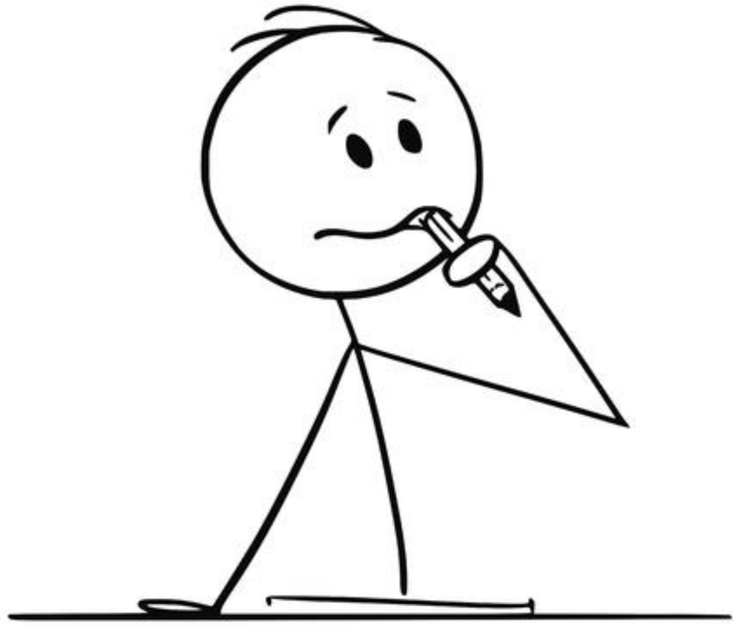
V. Better business optimization (low optimization cost).

VI. Helps in improving the overall quality of the system.



The difference in growth dynamics between projects with and without tests. A project without tests has a head start but quickly slows down to the point that it's hard to make any progress.

- You start off quickly because there's nothing dragging you down.
- No bad architectural designs have been made yet, and there isn't any existing code to worry about.
- As time goes by, you have to put in more and more hours to make the same amount of progress you showed at the beginning.
- Eventually, the development speed slows down significantly, sometimes even to the point where you cannot make any progress whatsoever.
- This phenomenon of quickly decreasing development speed is also known as *software entropy*.



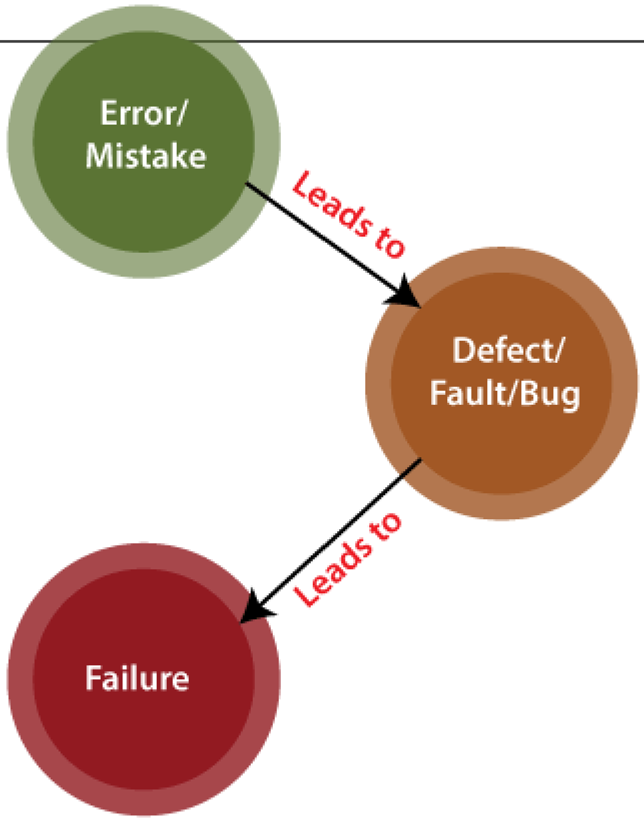
You are thinking...wait!!!
What?
What is the difference between a bug, error in software testing, software defect and fault?

The bugs introduced by programmer inside the code is called as Defect. Defect is defined as the deviation from the actual and expected result of application or software. Developer finds the defect and corrects it by himself during the development phase.

If testers find any mismatch in the application/system in testing phase then they call it as Bug.

A fault is an incorrect step, process or data definition in a software product.

When a defect reaches the end customer, it is called as Failure.



We need a Test Plan for this project. Its important
Making Test Plan is a waste of time. I want to work now. No need of test plan

A

I am Manager do everything as I said

B

OK, let's me explain why we need a Test Plan

Planning your Software Testing

I. The purpose of a written test plan is to improve the communication among all the people involved in testing application software.

II. Help people outside the test team such as developers, business managers, customers **understand** the details of testing.

III. The plan specifies each person's role during testing.

IV. Also serves as a checklist, whether or not the master test plan has been completed or not. Test Plan **guides** our thinking.

V. Many times, a master test plan is not just a single document, but a collection of documents.

VI. A master test plan is a project within the overall system development project.

VII. The Overall Plan and Testing Requirements sections are like a baseline project plan for testing, with a schedule of events, resource requirements, and standards of practice outlined.

VIII. Testing managers are responsible for creating testing plans, establishing testing standards, integrating testing, and ensuring that test plans are completed.

Software Entropy

I. In software, entropy manifests in the form of code that tends to deteriorate.

II. Each time you change something in a code base, the amount of disorder in it, or entropy, increases.

III. If left without proper care, such as constant cleaning and refactoring, the system becomes increasingly complex and disorganized.

IV. Fixing one bug introduces more bugs, and modifying one part of the software breaks several others – it's like domino effect.

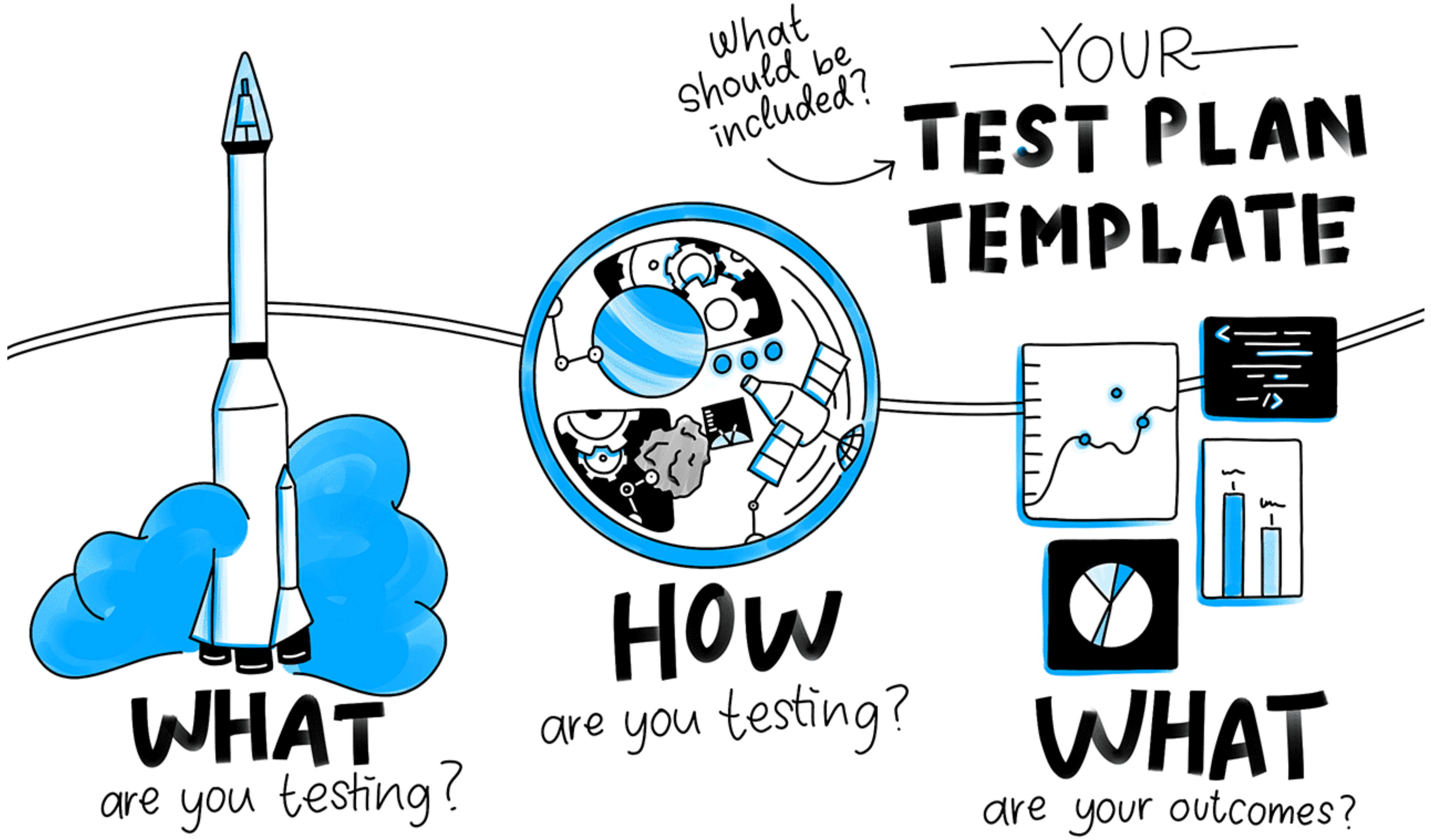
V. Eventually, the code base becomes unreliable.

VI. Worst of all, it is hard to bring it back to stability.

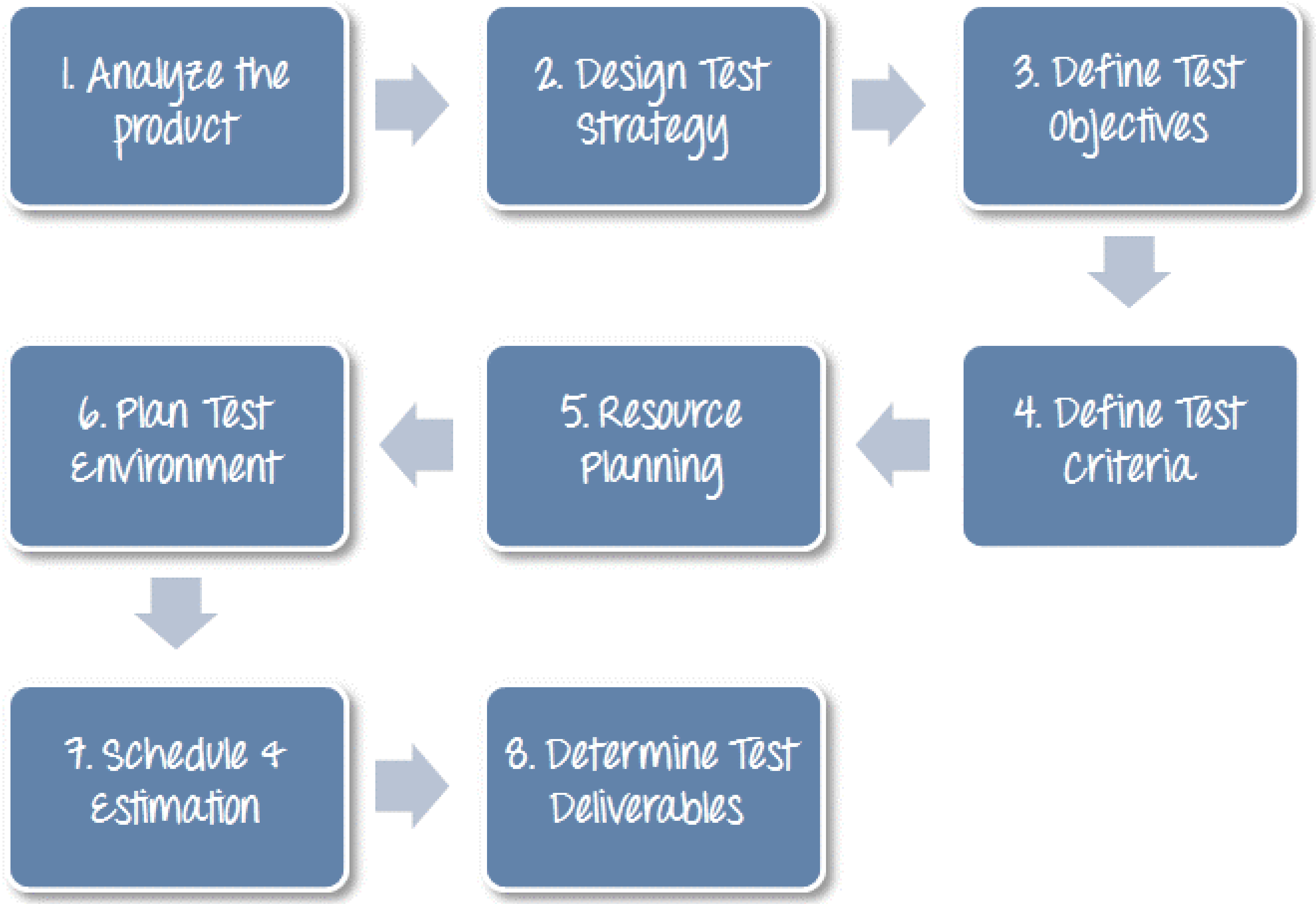
VII. Tests help overturn this tendency. They act as a safety net – a tool that provides insurance against a vast majority of regressions.

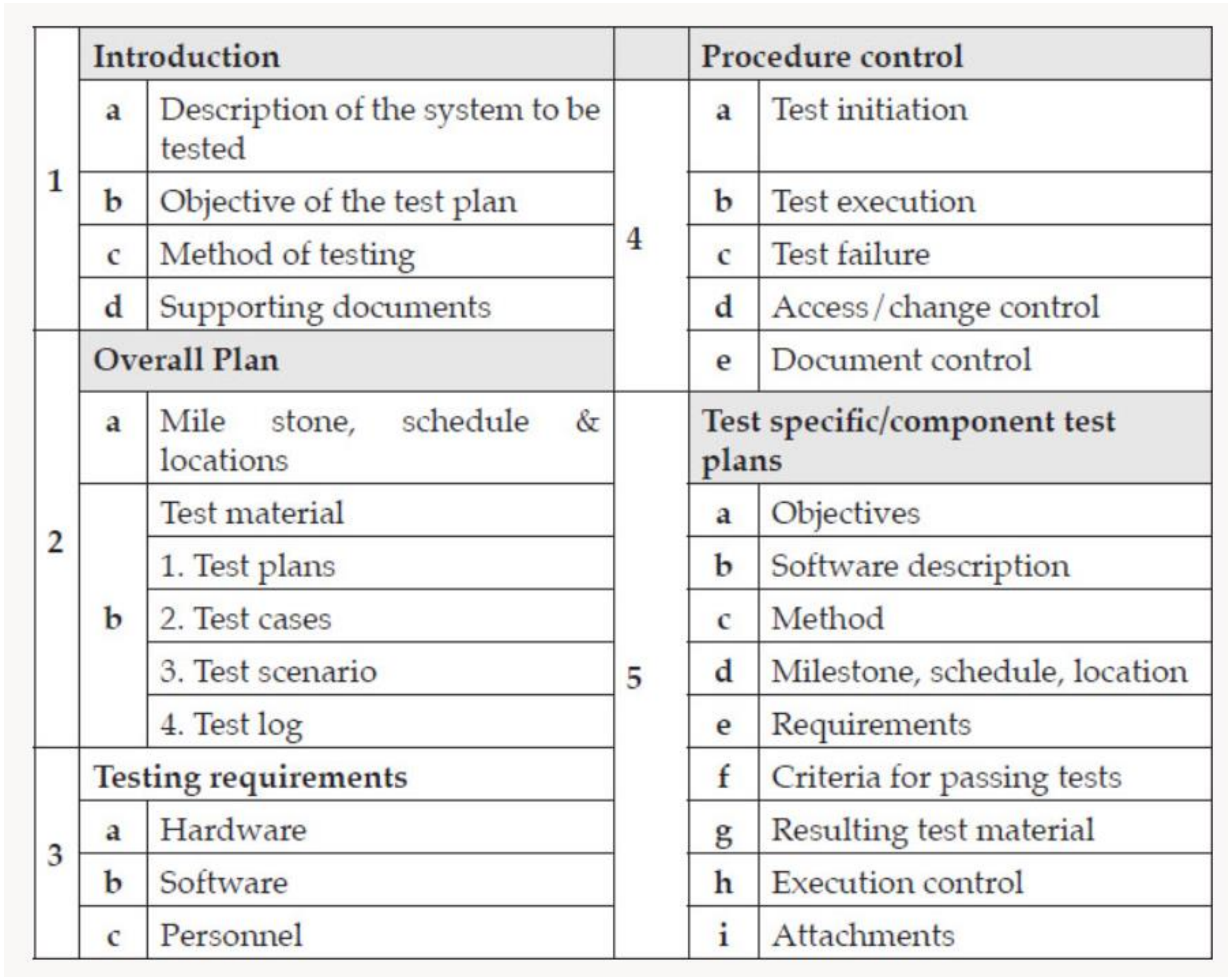
VIII. Tests help make sure the existing functionality works, even after you introduce new features or refactor the code to better fit the new requirements.

A regression is when a feature stops working as intended after a certain event (usually, a code modification). The terms *regression* and *software bug* are synonyms and can be used interchangeably.



Test planning in simple words.


1. Analyze the product
2. Design Test Strategy
3. Define Test Objectives
4. Define Test Criteria
5. Resource Planning
6. Plan Test Environment
7. Schedule & Estimation
8. Determine Test Deliverables



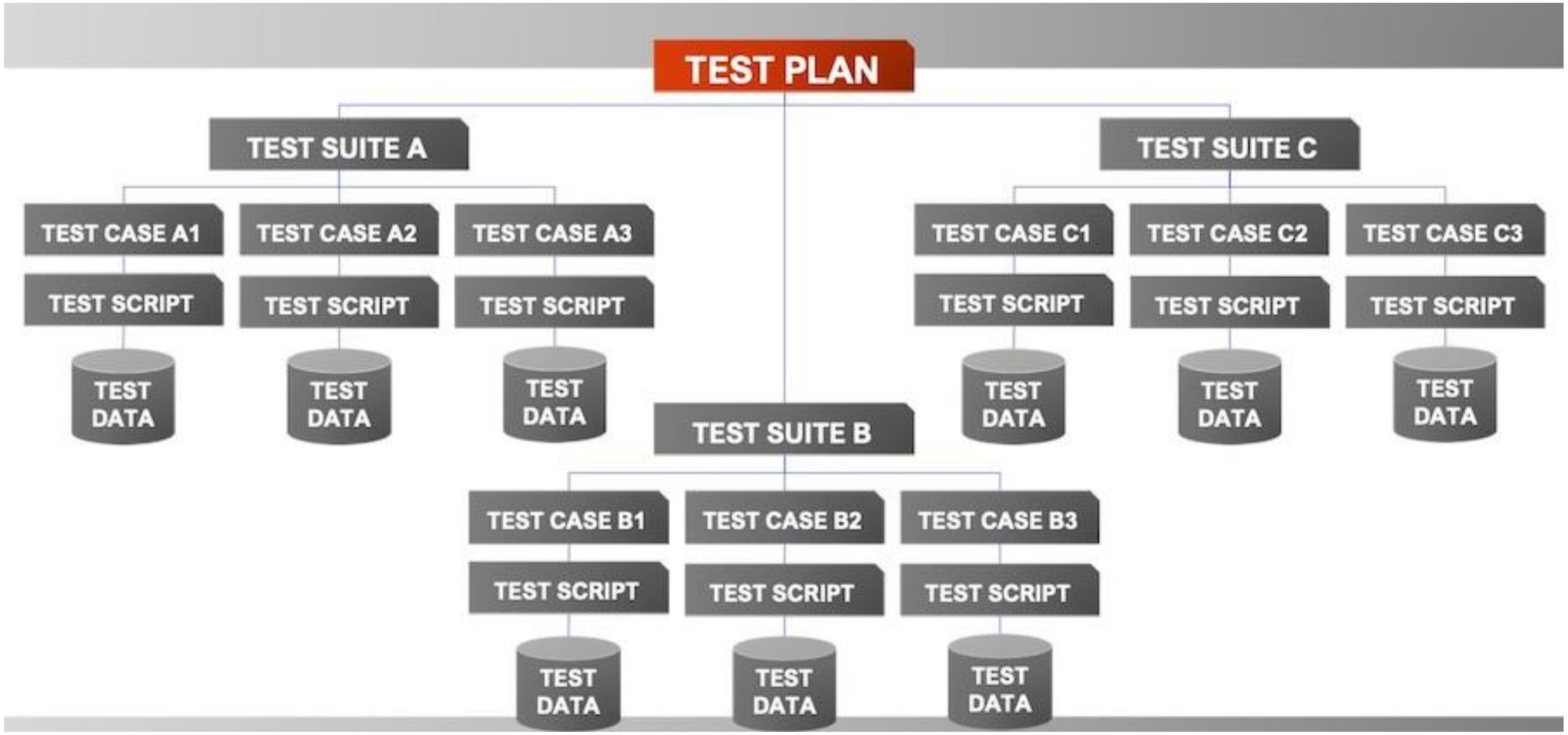
1	**Introduction**	
	a	Description of the system to be tested
	b	Objective of the test plan
	c	Method of testing
	d	Supporting documents
2	**Overall Plan**	
	a	Mile stone, schedule & locations
	b	Test material
		1. Test plans
		2. Test cases
		3. Test scenario
		4. Test log
3	**Testing requirements**	
	a	Hardware
	b	Software
	c	Personnel
4	**Procedure control**	
	a	Test initiation
	b	Test execution
	c	Test failure
	d	Access / change control
	e	Document control
5	**Test specific/component test plans**	
	a	Objectives
	b	Software description
	c	Method
	d	Milestone, schedule, location
	e	Requirements
	f	Criteria for passing tests
	g	Resulting test material
	h	Execution control
	i	Attachments

Table of content for a Master Test Plan.

Procedure Control explains how testing to be conducted.

Control procedures are the use of standard and consistent procedures in giving directions in a testing situation in order to control all but the variables being examined.

This means, in part, that in a testing or experimental procedure instructions are given in a "scripted' manner so that all participants get the exact same information in the exact same way so as not to compromise the experiment.



For not so complex projects.

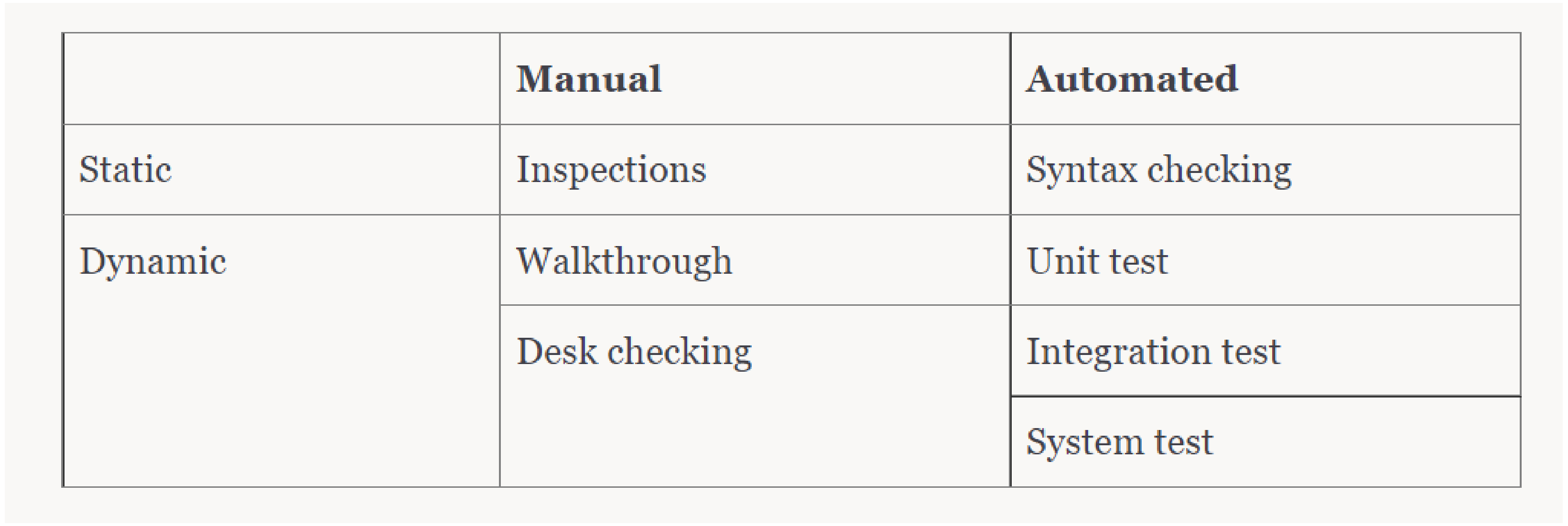
I. Software application testing is an umbrella term that covers several types of tests.

II. Mosely (1993) organizes the type of tests according to whether they employ static or dynamic techniques and whether the test is automated or manual.

III. *Static Testing* means that the code being tested is not executed. The results of running the code are not an issue for that particular test.

IV. *Dynamic Testing*, involves execution of code.

V. *Automated testing* means the computer conducts the test while *Manual Testing* means that people do.



	Manual	**Automated**
Static	Inspections	Syntax checking
Dynamic	Walkthrough	Unit test
	Desk checking	Integration test
		System test

Inspections, Walkthroughs

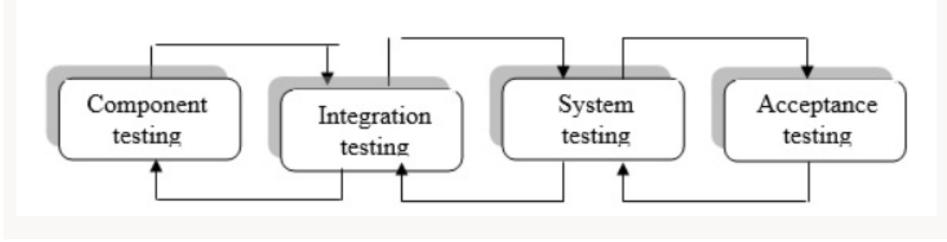
I. Formal group of activities where participants manually examine the code for occurrences of well-known errors.

II. Syntax, grammar, and some other routine errors can be checked by automated inspection software.

III. Walkthrough, in a structured manner, is a very effective method of detecting errors in code.

IV. Code walkthroughs should be done frequently when the pieces of work reviewed are relatively small and before the work is formally tested.

Unit Testing



I. Sometimes called module testing, often done automatically.

II. In unit testing, each module is tested alone in an attempt to discover any error that may exist in the module's code.

III. Except for small programs, system should not tested as a single, monolithic unit.

IV. Large programs consist of many smaller pieces – classes, methods, packages, etc.

V. "Unit" is a generic name for these smaller pieces.

VI. Three important types of testing are:
   I. Unit Testing (testing units in isolation)
   II. Integration Testing (test integrated units)
   III. System Testing (test entire system that is fully integrated)

VII. Unit testing is done to test the smaller pieces in isolation before they are combined with other pieces. Usually done by developers who write the code.

The current state of unit testing

I. For the past two decades there has been a push towards adopting unit testing.

II. Unit testing is now considered mandatory in most companies.

III. Unless you are working on a throwaway project, unit testing is necessary.

IV. The discussion has shifted from “Should we *write* unit tests?” to “What does it mean to write *good* unit tests?” This is where the main confusion still lies.

V. Results of this confusion are visible in software projects.
    I. Many projects have automated tests; they may even have lots of them.
    II. But the existence of those tests often doesn’t provide the results the developers hope for.
    III. Unit tests that are supposed to help do not seem to help when new features take forever to implement, new bugs constantly appear in the already implemented and accepted functionality.

Relationship between unit testing and code design

I. If you find a code is hard to unit test, it is a strong sign that the code needs improvement.

II. The poor quality usually manifests itself in *tight coupling*, which means different pieces of production code are not decoupled from each other enough, and it is hard to test them separately.

III. But, if you code is easy to unit test does not necessarily indicate its good quality.

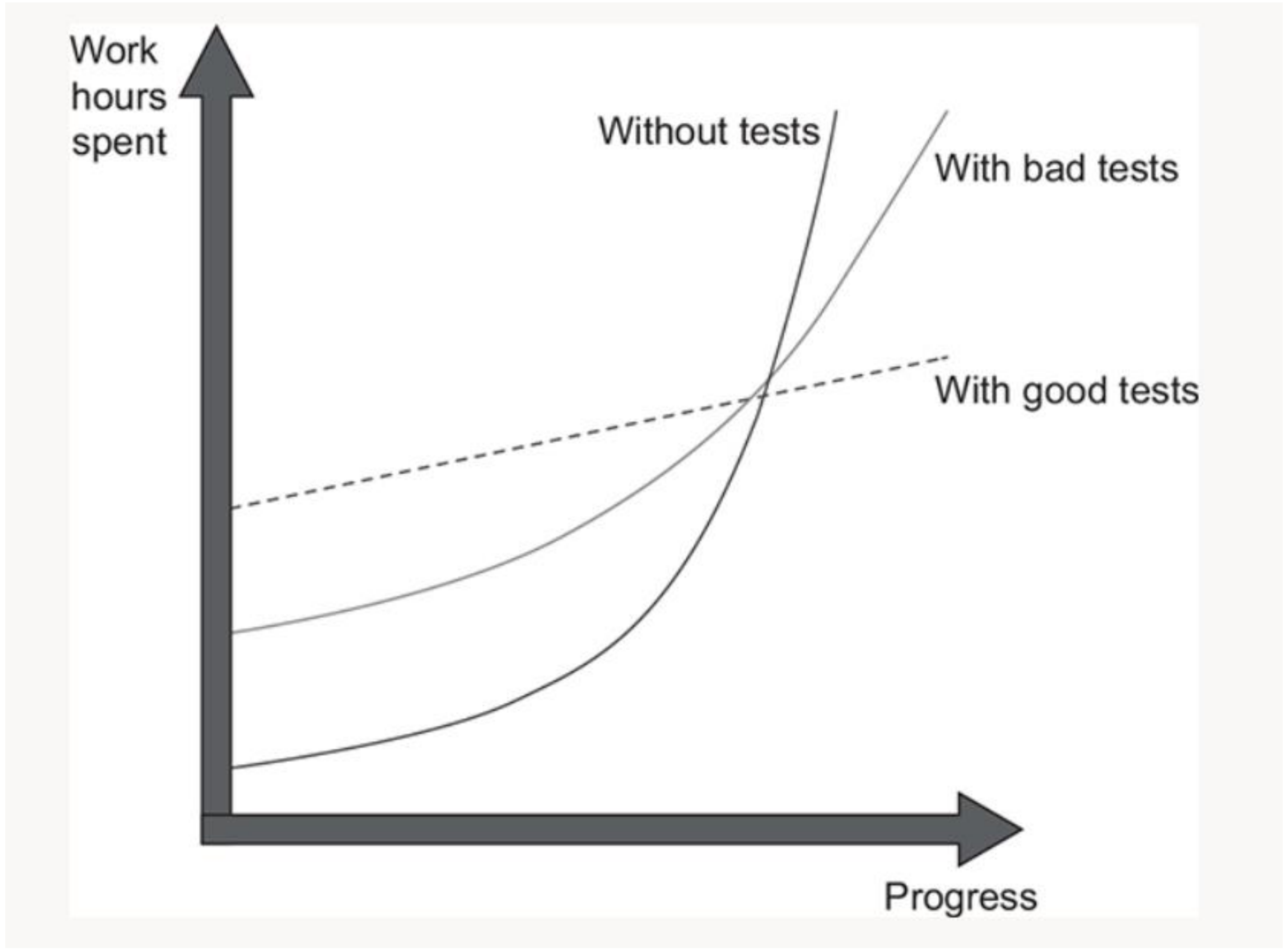
IV. Thus, the goal of unit test is to ***enable sustainable growth of the software project***.

V. It is quite easy to grow a project, especially when you start from scratch.

VI. It’s much harder to sustain this growth over time.

What makes a good or bad test?

- It is not enough to just write tests.
- Badly written tests still result in the same picture.
- Bad tests do help to slow down code deterioration at the beginning. But nothing really changes in the grand scheme of things.
- It might take longer for such a project to enter the stagnation phase, but stagnation is still inevitable.
- Not all tests are created equal.
- Bad tests can raise false alarms, don’t help you catch regression errors, and are slow and difficult to maintain.
- It’s easy to fall into the trap of writing unit tests for the sake of unit testing without a clear picture of whether it helps the project.
- Things to remember : test’s value and its upkeep cost.



The difference in growth dynamics between projects with good and bad tests. A project with badly written tests exhibits the properties of a project with good tests at the beginning, but it eventually falls into the stagnation phase.

Integration Testing

I. Have you ever been in a situation where all the unit tests pass but the application still doesn’t work?

II. You can never be sure your system works as a whole if you rely on unit tests exclusively.

III. Validating software components in isolation from each other is important, but it’s equally important to check how those components work in integration with external systems.

IV. This is where integration testing comes into play.

V. You have to validate how different parts of a system integrate with each other and external systems: the database, the message bus, and so on.

Contd…

I. Integration testing is done by combining the modules and testing them together.

II. It is gradual.

III. The integration plan is guided by the module dependency graph of the structure chart.

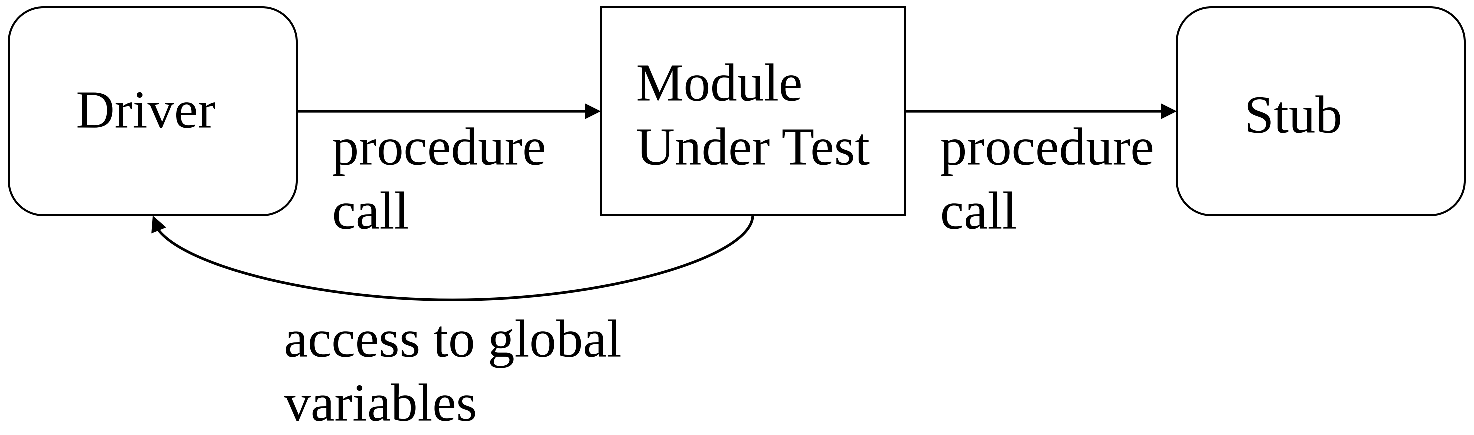
IV. Modules are typically integrated in a top-down, incremental order.

V. You continue this procedure until the entire program has been tested as a unit.

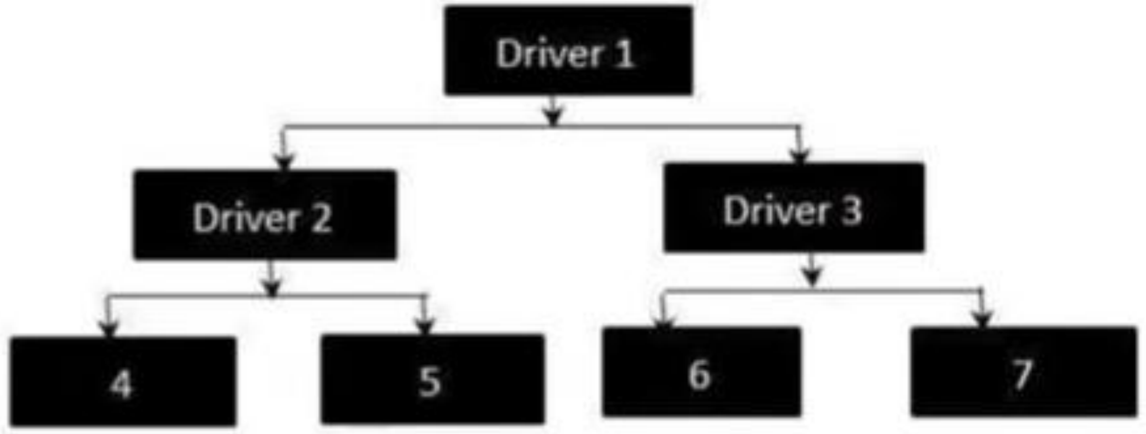
Drivers and Stubs

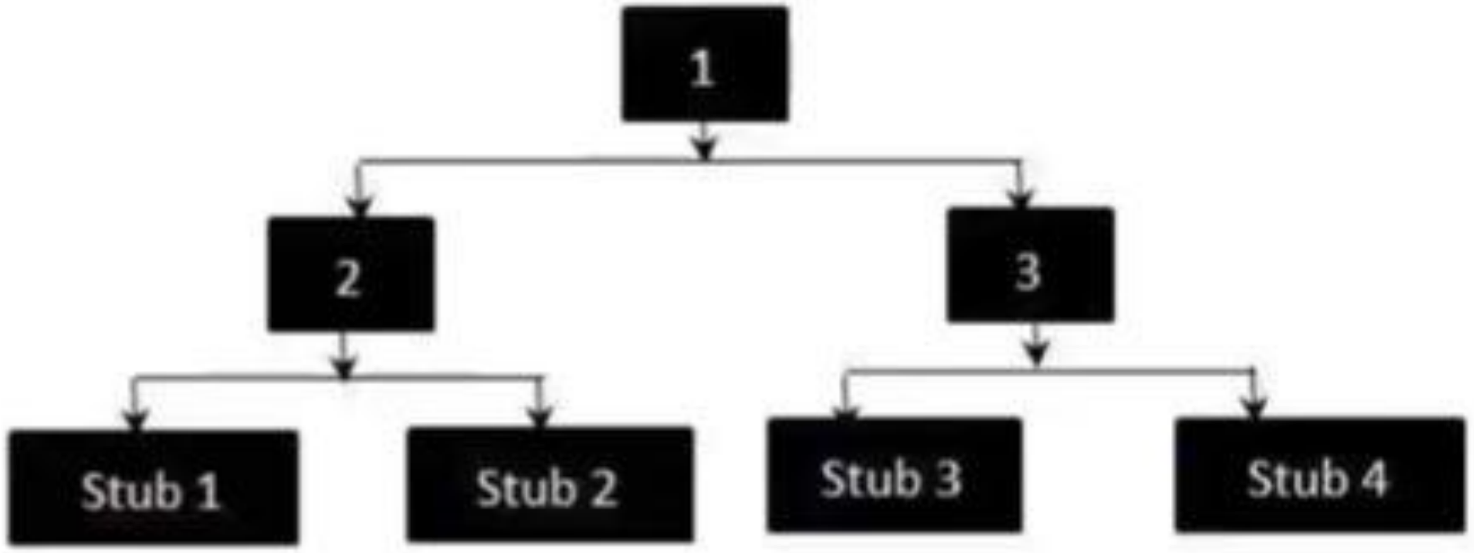
- It is almost always a good idea to develop and test software in “pieces”.
- At first glance this may seem impossible because it is hard to imagine how one can test one “piece” if the other “pieces” that it uses have not yet been developed.
- The way around this is to use stubs and drivers.
- Very simply:
- **STUB** A piece of code that simulates the activity of missing components.
- **DRIVER** A piece of code that passes test cases to another piece of code.

Drivers and Stubs



- Driver and Stub should have the same interface as the modules they replace
- Driver and Stub should be simpler than the modules they replace





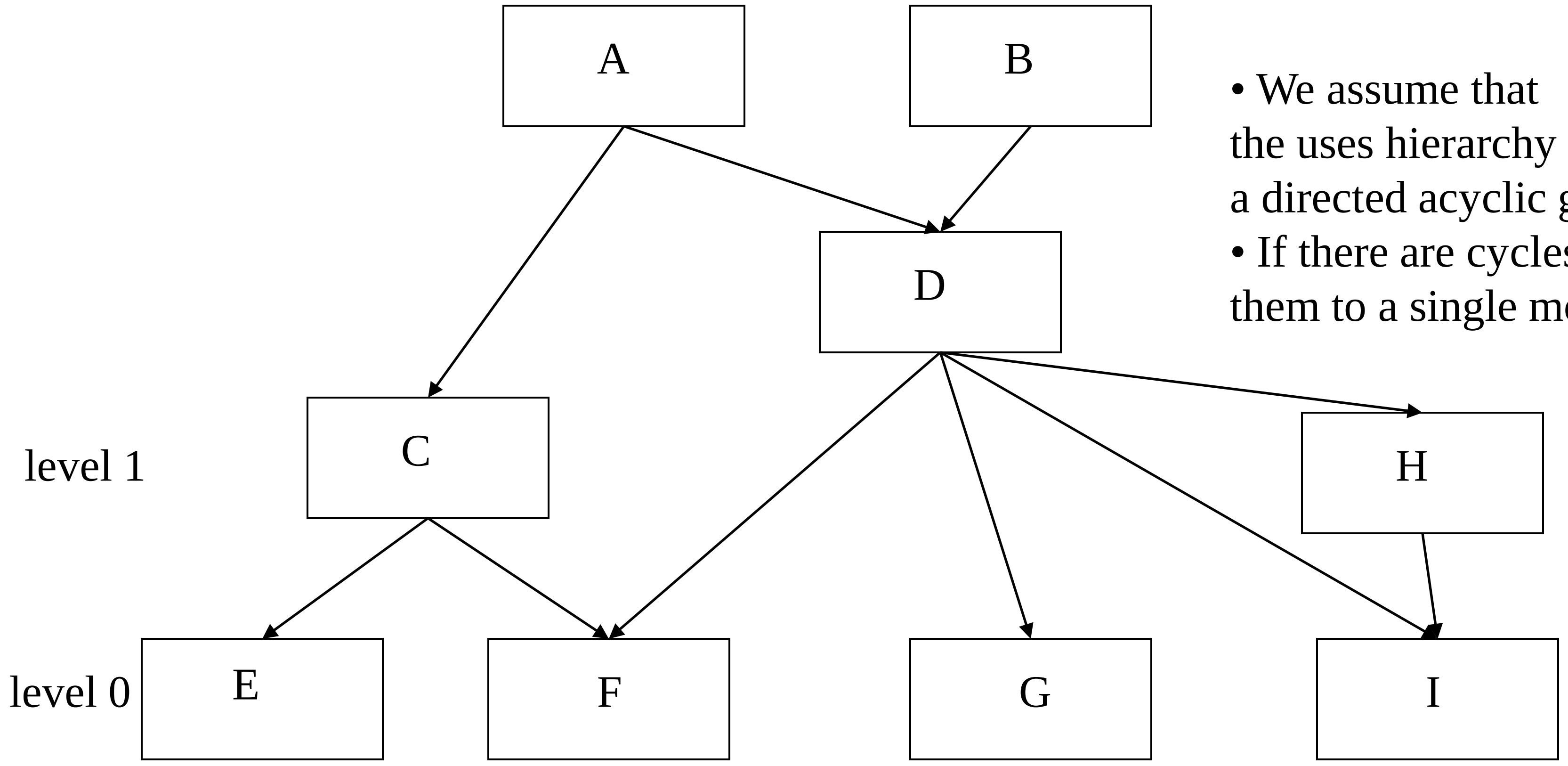
Video

Example

- We need to develop a “calculator” that converts between different units (e.g. feet to inches)
- We “stub out” the **`UnitConverter`** class.
- This “stubbed-out” class contains comments that describe the class and all of the methods that we expect it to have.

- In summary:
- **Stubs** are created during top-down integration. i.e Allow the testing of the upper level module when the lower level module is not yet developed.
- **Drivers** are created during bottom-top integration. Ie. Allow testing of the lower level modules when upper level modules are not yet developed.

Module Structure



• We assume that
the uses hierarchy is
a directed acyclic graph.
• If there are cycles merge
them to a single module

• A uses C and D; B uses D; C uses E and F; D uses F, G, H and I; H uses I
• Modules A and B are at level 3; Module D is at level 2
Modules C and H are at level 1; Modules E, F, G, I are at level 0
• level 0 components do not use any other components
• level *i* components use at least one component on level *i*-1 and no component at a level higher than *i*-1

Top-down integration approach

I. Integration starts with the root module and one or two sub modules followed by testing.

II. After the top-level modules are tested, the immediate module is combined and become ready for testing.

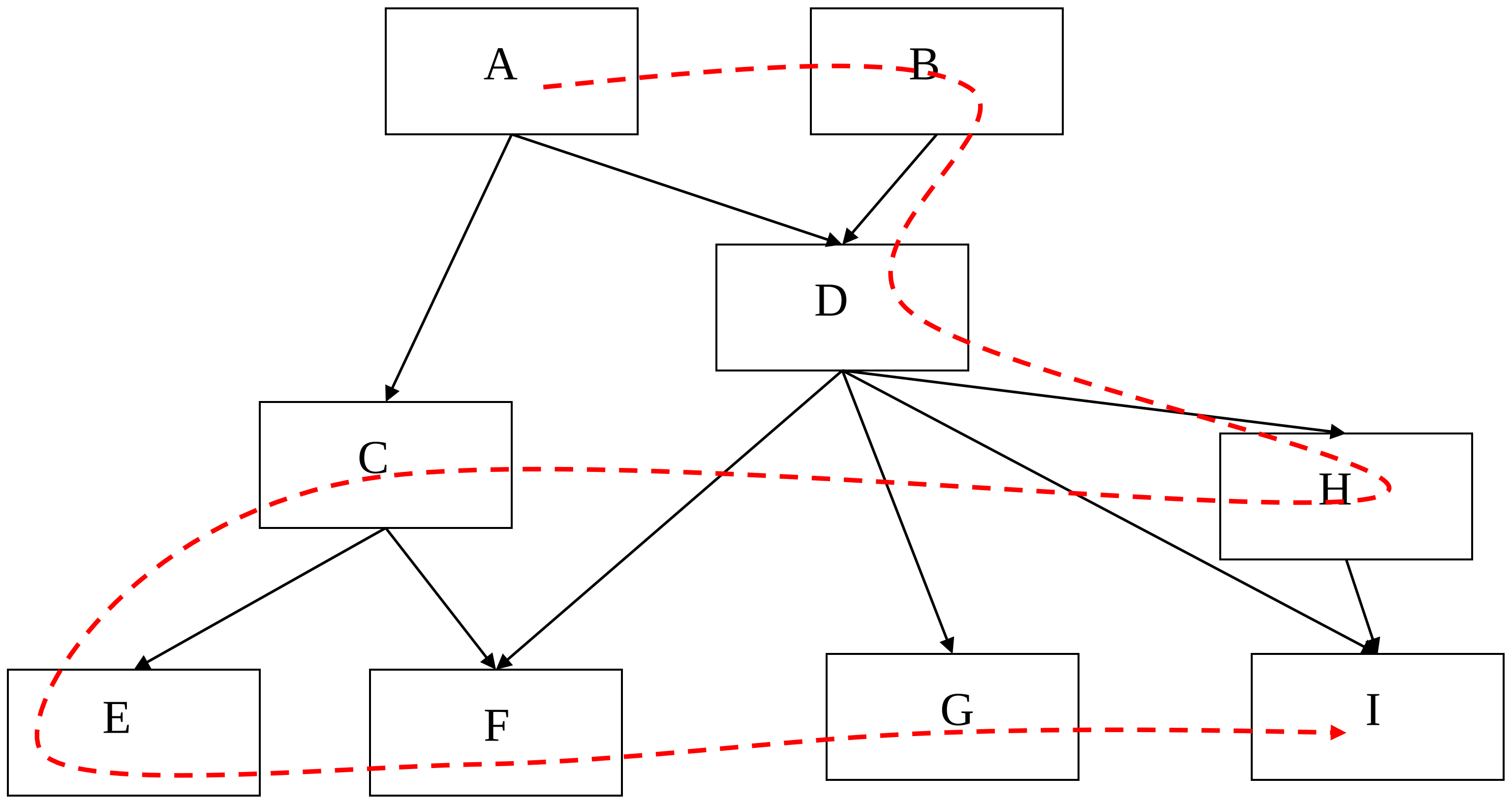
III. It is suitable for small systems.

IV. Advantage : Depth-first strategy make the parts of system available for early demo.

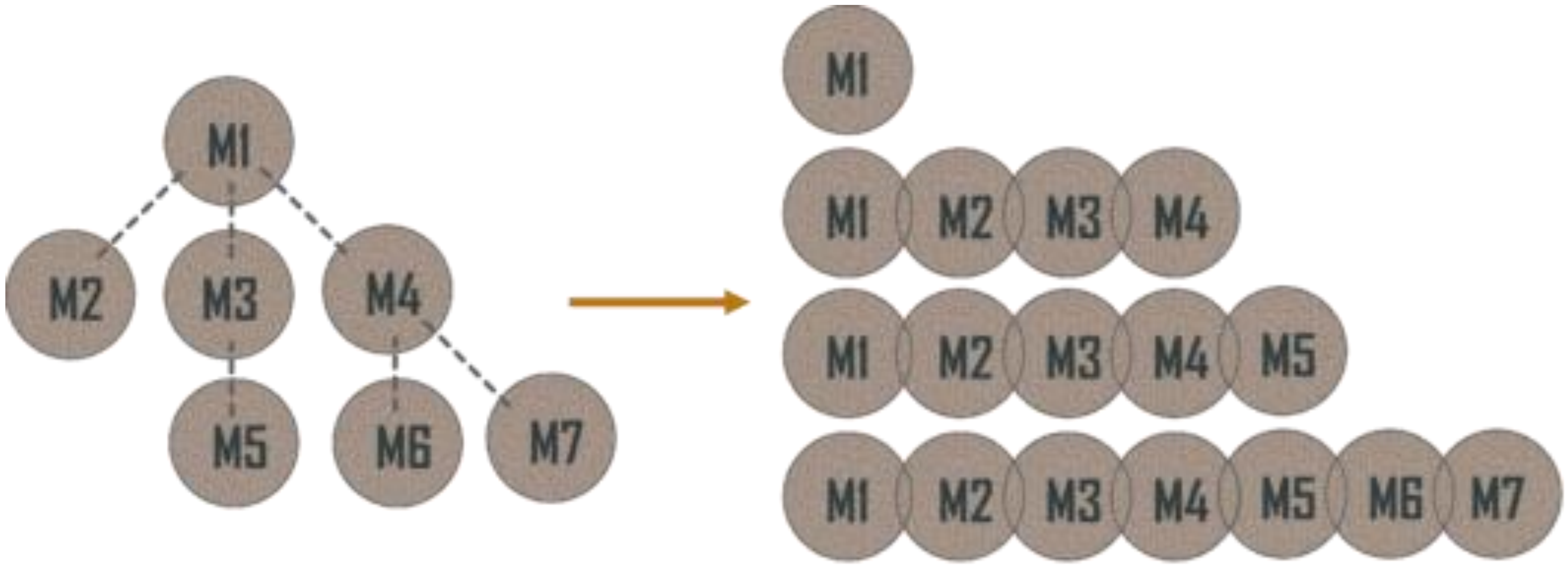
V. Building stubs adequate to test upper levels may require a lot of effort.

VI. A lot of effort (later discarded) is put into designing and implementing drivers and stubs.

Top-down Integration



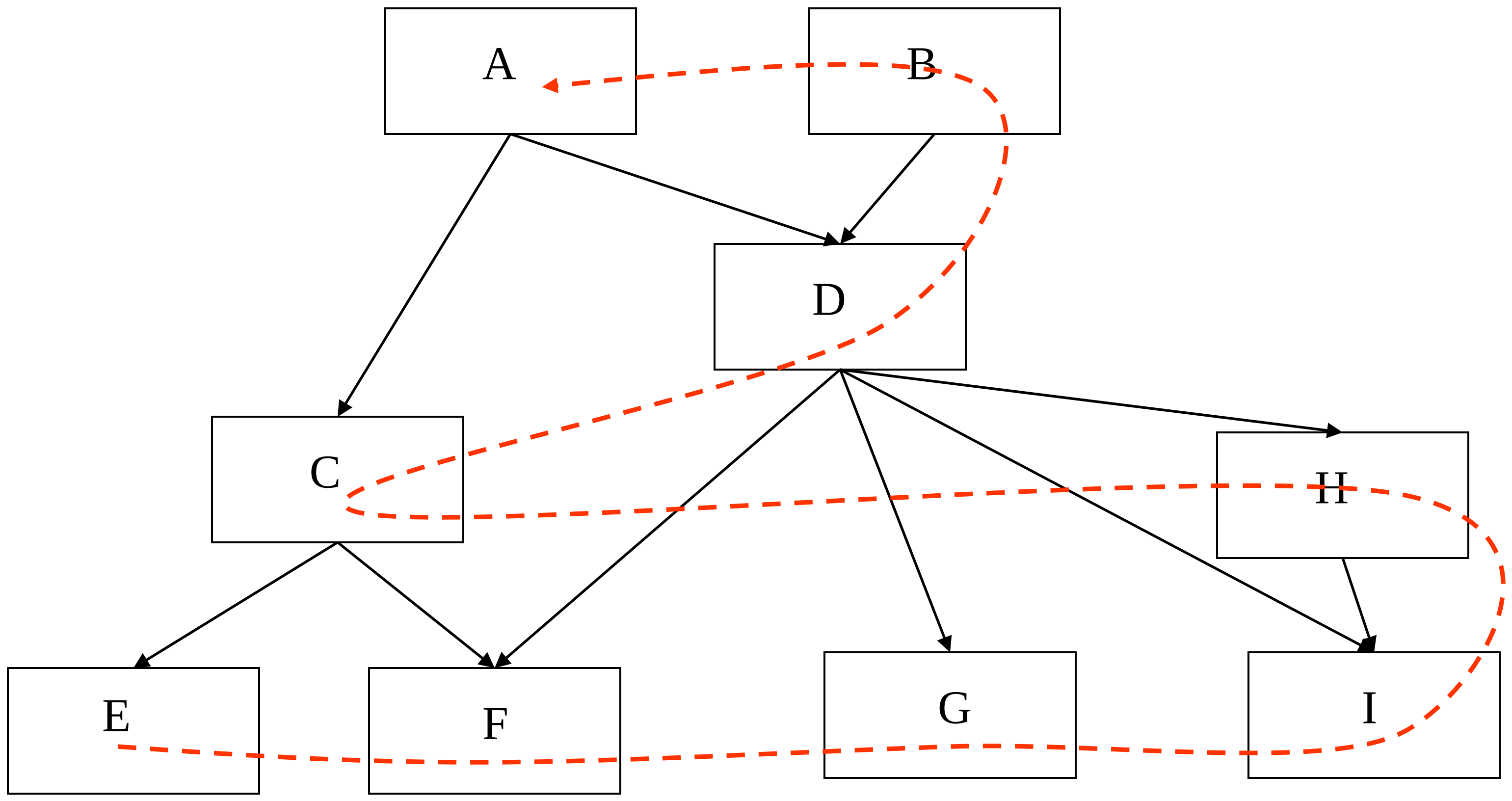
Top down integration



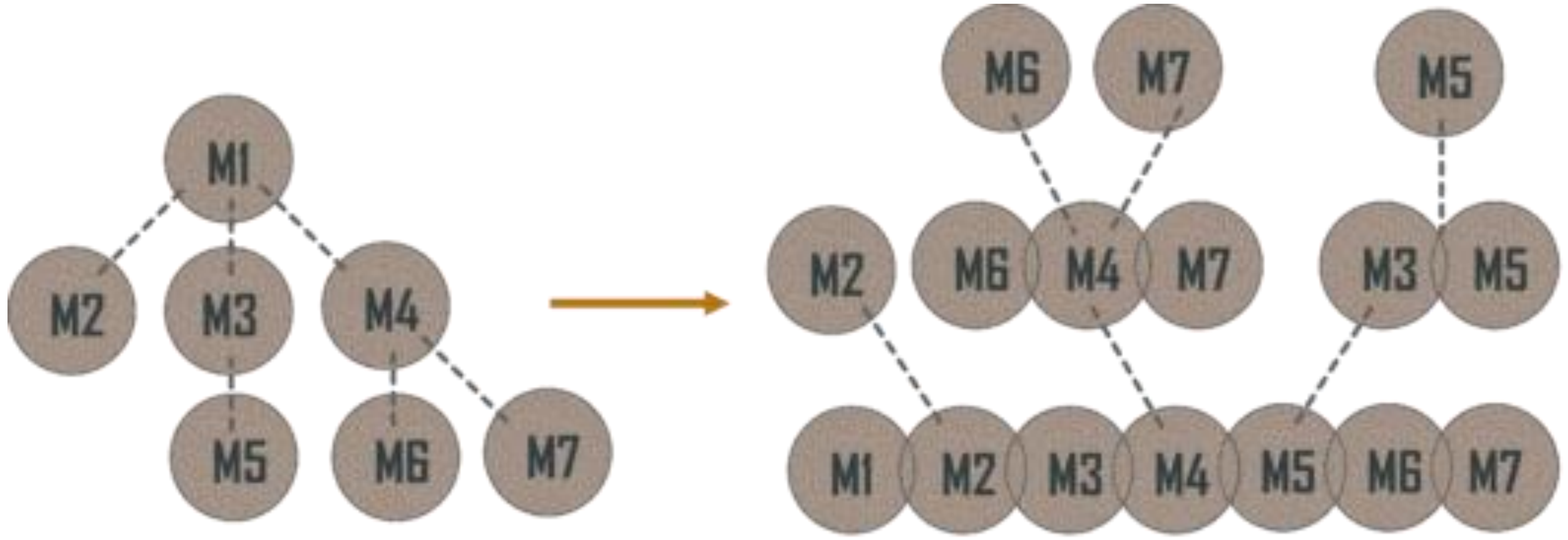
Bottom-top integration

- Up - Start with atomic (leaf) modules.
- Use drivers to tests clusters of modules.
- Gradually add calling modules and move testing upward.
- The purpose of testing each sub module is to test the interfaces among them.
- In this situation control and data interfaces between modules are tested.
- The advantage is, many sub systems, having no dependency, can be tested simultaneously.
- The disadvantage is that during testing a number of sub systems makes the situation complex.
- This extreme condition calls for **big bang approach**.

Bottom-up Integration

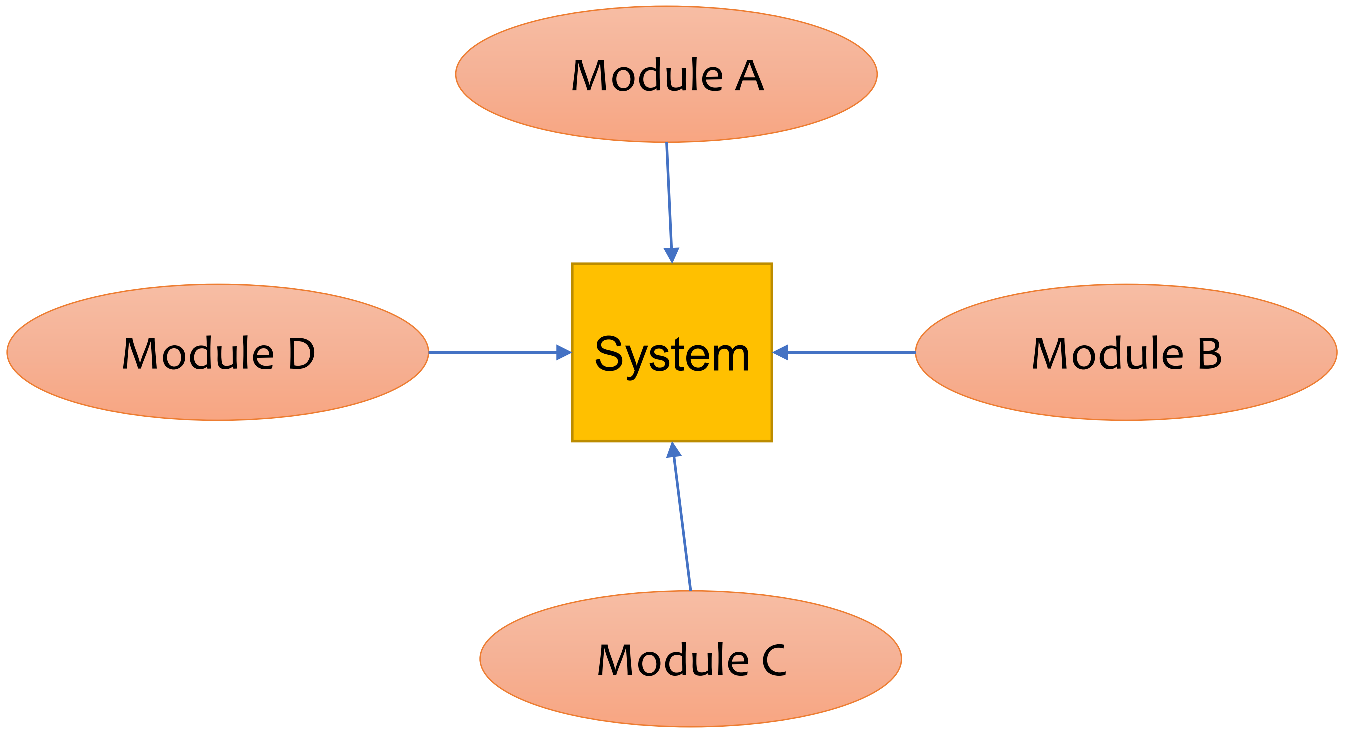


Bottom up integration



Big Bang Approach

- All the sub modules are integrated in a single step and tested together.
- A form of non-incremental integration testing.
- This approach is convenient for a small system where on-go-the errors are fixed.
- ***Fault Localization Issues***: The disadvantage is to spot the error found in the exact module during integration testing.
- To remove the error during integration testing is costly and time consuming.
- Generally followed by those developers who follow "Run and See" approach.
- Chances of having critical failures are more because of integrating all the components together at the same time.

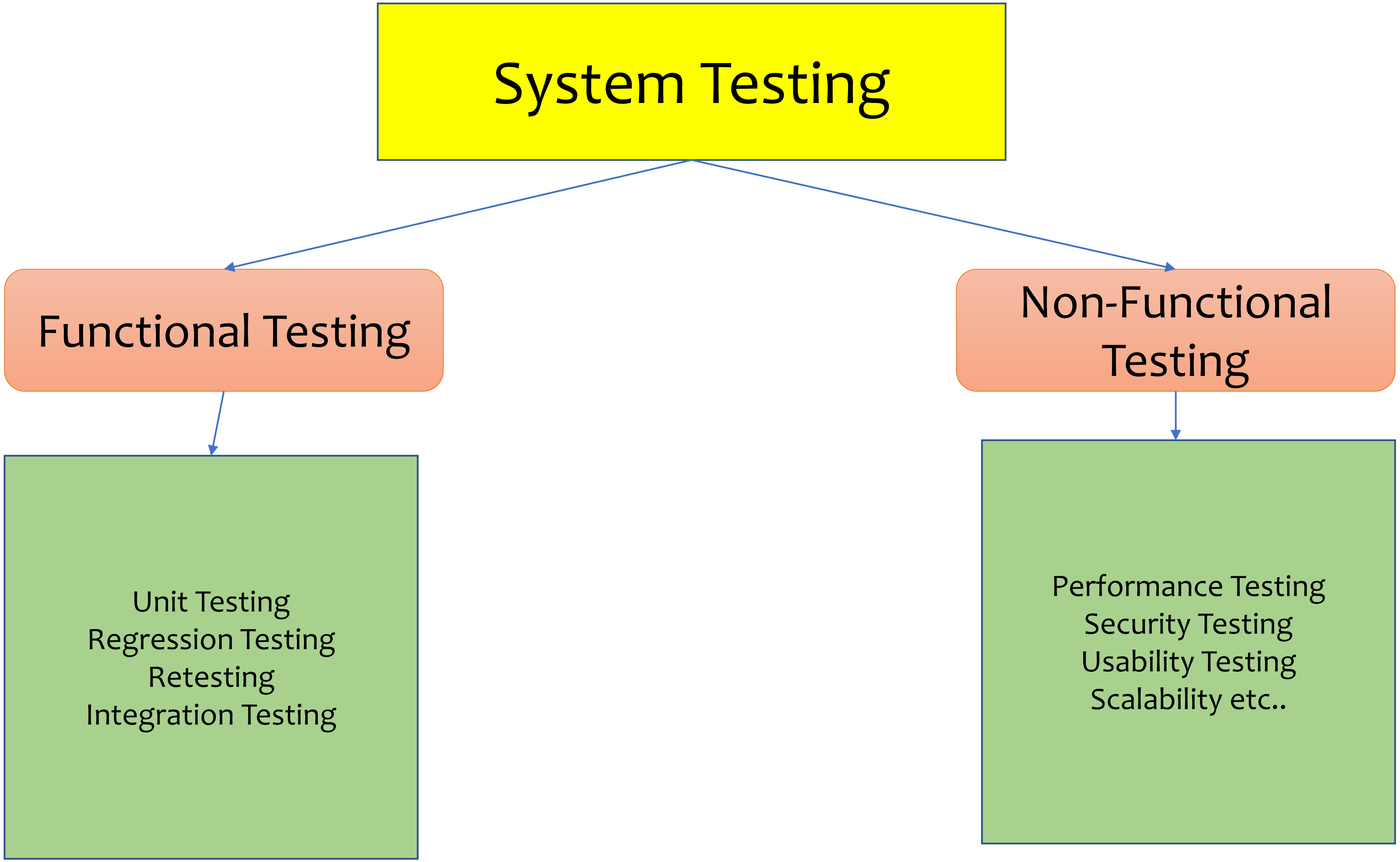


Mixed Integration Approach

- Is a combination of both top-down and bottom-up approaches.
- Also known as ***sandwiched integration, hybrid integration testing.***
- The disadvantages of both the approaches are reduced to a great extent.
- The testing is possible from both top and bottom level, wherever the modules are available.
- This is common approach for integration testing .
- In top-down integration, integration testing can start only once the top-level modules are coded and unit tested while in bottom-up approach integration testing can be started only when bottom level modules are coded and unit tested.
- Mixed integration testing approach overcomes those limitations by making sure integration testing of the modules can be started as and when they are available after unit testing.
- Top-level modules are tested with lower level modules and lower level modules are tested with high-level modules simultaneously.
- Mixed integration testing requires to create both the stubs and the drivers.
- It is a complicated method.

System Testing

- Is similar to integration testing, but instead of integrating modules into programs for testing, you integrate programs into systems.
- Tests are performed to know if a complete build aligns with the functional and non-functional requirements made for it.
- System testing follows the same incremental logic that does the integration testing.
- Programs are typically integrated in top-down incremental fashion.
- Under both integration and system testing, the individual modules and programs get tested many times as well as interfaces between modules and programs also get tested.
- Considering the test location and data : two types of testing are conducted- Alpha testing and beta testing.


System Testing
Functional Testing
Non-Functional Testing
Unit Testing
Regression Testing
Retesting
Integration Testing
Performance Testing
Security Testing
Usability Testing
Scalability etc..

When is system testing done?

- During the development of each software version.
- During the application launch, through alpha and beta testing.
- When unit and integration testing are complete.
- This test level checks the software’s conformity to user demands, functional performance, and design.

Alpha testing and Beta Testing

- **Alpha Testing**
  - Tests performed at the developer’s site before the system is finally installed in the real working environment (user’s site).
  - Involves testing the system with live data supplied by the organization rather than by the test data used by the system designer.
- **Beta Testing**
  - The system is delivered to a number of potential users who agree to use that system and provide feedback to the designers.

Acceptance Testing

- The final stage in the testing process before the system is accepted for operational use by the client.
- System is tested with data supplied by the user than with simulated data.
- May reveal errors and omissions in the system requirement definition.
- May also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is not acceptable.

The Test Strategy

- Prepare test plan.
  - Outputs expected from the system. Criteria for evaluating outputs. A volume of test data. Procedure for using test data. Personnel and training requirements.
- Specify conditions for user acceptance testing.
  - Analyst and the user should agree on the conditions for the test.
- Prepare test data for program testing.
  - Test data should be prepared and documented to ensure that all aspects of the program are properly tested.
- Prepare test data for transaction path testing.
  - Develops the data required for testing every condition and transaction to be introduced into the system. The path of each transaction from origin to destination is carefully tested for reliable results.
- Plan user training.
- Compile/assemble programs.
- Prepare job performance aids.
  - Materials to be used by personnel to run the system are specified and scheduled. Eg. Posted instruction schedule to load the data/disk drive or to a help menu.
- Prepare operational documents.
  - Mainly related to setting some performance criteria to be used.
  - Turn around time, back up, file protection, human factor.

Guidelines for module testing

- To emphasize the importance and applicability to module testing the following guidelines may be followed:
  - Try to design module tests so that failure to pass one test case will not move to the next test case.
  - Under stressful load and conditions, test the modules at their performance limit and beyond.
  - Measure the performance characteristics like throughput accuracy, input and output capacities, and timings should be measured for each module.
  - Concurrently many problems should not be solved. They should be handled one after the other so that the source of error can be traced easily.
  - Test data as input should be valid or representative of the data to be provided by the user. If not, then the reliability of the output is suspected.
  - If using artificial data, make sure al combinations of values and formats are input to the system.

Testing Frameworks

- Following are the different testing frameworks:
- jUnit - for Java unit test.
- Selenium - is a suite of tools for automating web applications for software testing purposes, plugin for Firefox.
- HP QC - is the HP Web-based test management tool. It familiarizes with the process of defining releases, specifying requirements, planning tests, executing tests, tracking defects, alerting on changes, and analyzing results. It also shows how to customize project.
- IBM Rational - Rational software has a solution to support business sector for designing, implementing and testing software.

Conclusion

- Software Testing is done to ensure that it runs correctly, including hardware and other software linked to it.
- Planning, discipline, procedure control and documentations are vital for successful software testing.
- Planning for the test phase should begin early in the development life cycle and should be constant concern throughout the development cycle.