Lecture 19: Testing (3)

Integration Testing

Integration Testing

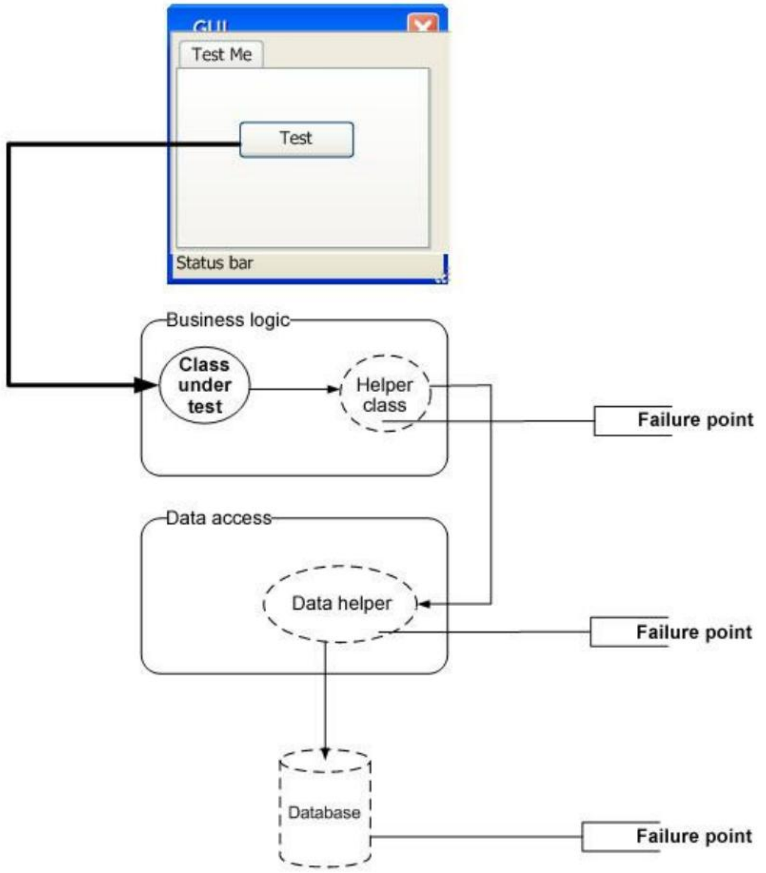
- Integration testing ("I&T") is the phase in software testing in which individual software modules are combined and tested as a group.

- It occurs **after** unit testing and **before** system testing.

- Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

Integration Testing (2)

- Integration testing: Verifying software quality by testing two or more dependent software modules as a group.
- Challenges:
  - Combined units can fail in more places and in more complicated ways.
  - How to test a partial system where not all parts exist?
  - How to "rig" the behavior of unit A so as to produce a given behavior from unit B?



Why do you need integration testing?

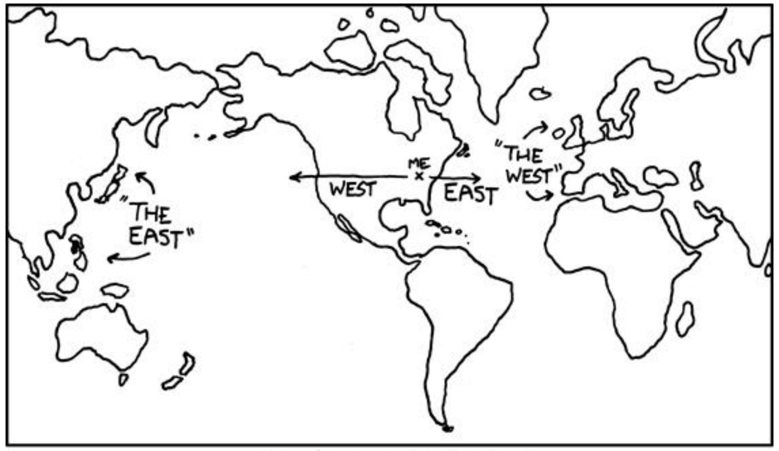
- To make sure that your components satisfy the fallowing requirements:
  - Functional.
  - Performance.
  - Reliability.

Terminology

- Stub – the dummy modules that simulates the low level modules.
- Driver – the dummy modules that simulate the high level modules.
- Top-down approach (also known as step-wise design) - the breaking down of a system to gain insight into its compositional sub-systems.
- A bottom-up approach is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system.

Terminology (2)

- Regression testing - any type of software testing that seeks to uncover new errors, or *regressions*, in existing functionality after changes have been made to the software, such as functional enhancements, patches or configuration changes.



THIS ALWAYS BUGGED ME.

What is Big Bang Testing?

- In Big Bang Integration testing, individual modules of the programs are not integrated until every thing is ready. It is called 'Run it and see' approach.
- In this approach, the program is integrated without any formal integration testing, and then run to ensures that all the components are working properly.

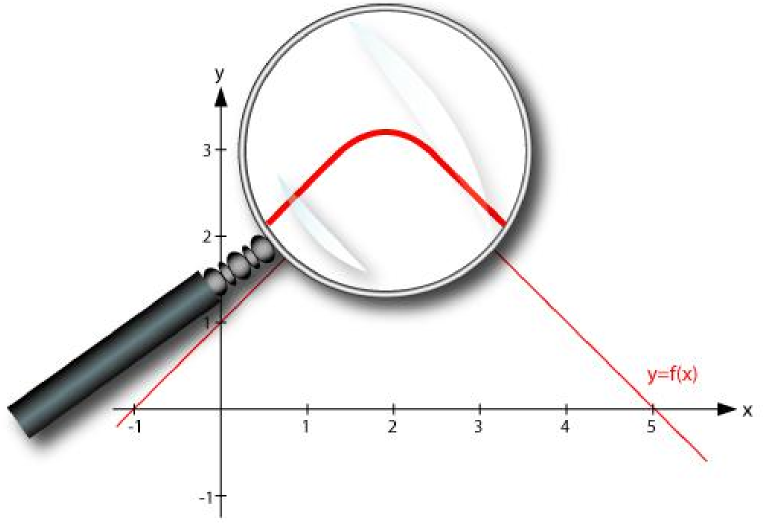
Disadvantages of Big Bang

- Defects present at the interfaces of components are identified at very late stage.
- It is very difficult to isolate the defects found, as it is very difficult to tell whether defect is in component or interface.
- There is high probability of missing some critical defects which might surfaced in production.
- It is very difficult to make sure that all the cases for integration testing are covered.

Big Bang Testing: Conclusion

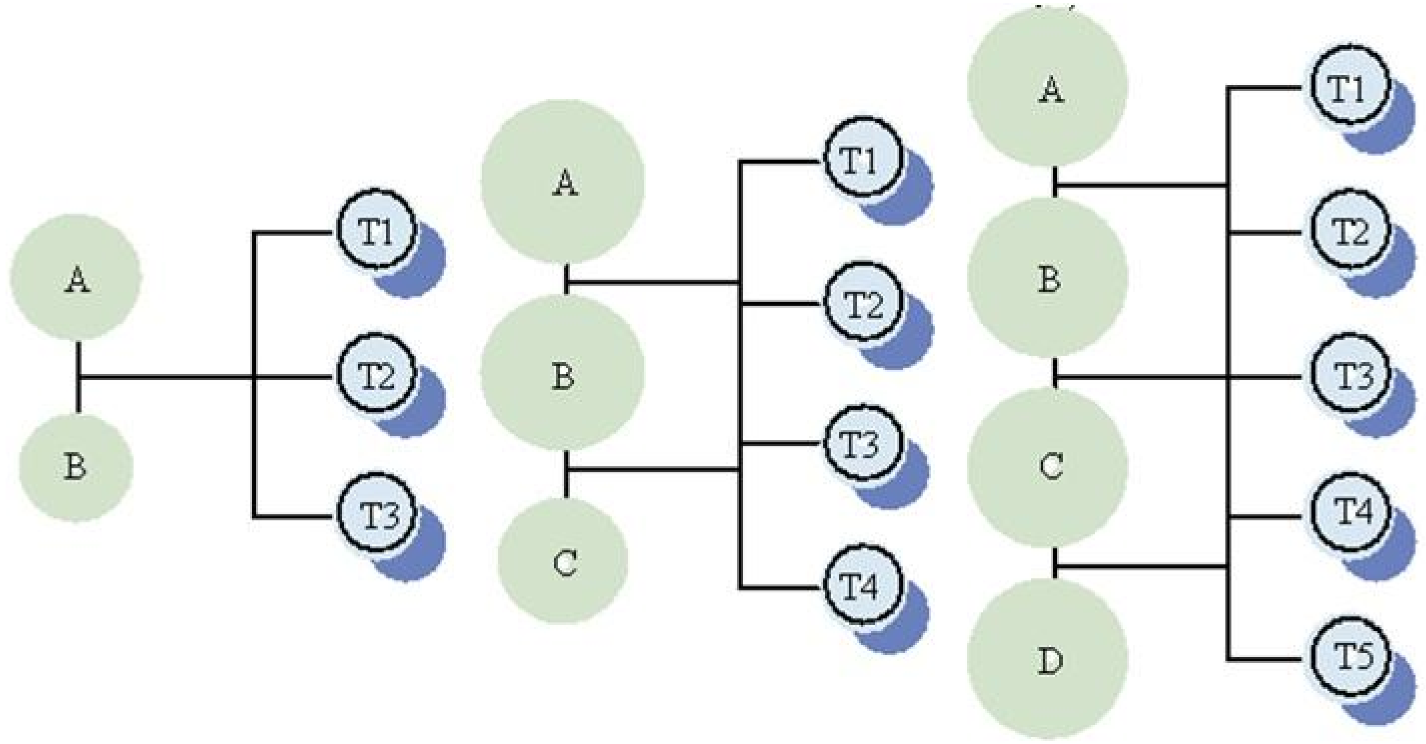
- This is not the way you should integrate and test software!

- But it might be good with this assumption applied:
  - for small systems, but not for enterprise level applications.



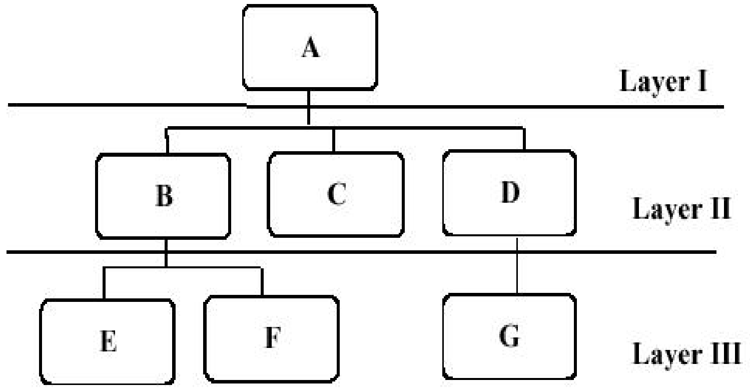
How to integrate?

- If Big Bang integration is bad, then how to integrate?
- The answer: incremental integration.



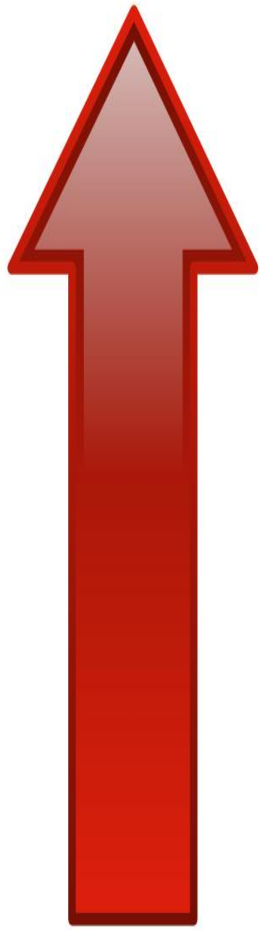
Bottom-up Integration Testing

- In bottom up integration testing, module at the lowest level are developed first and other modules which go towards the 'main' program are integrated and tested one at a time.
- Bottom up integration also uses test drivers to drive and pass appropriate data to the lower level modules.

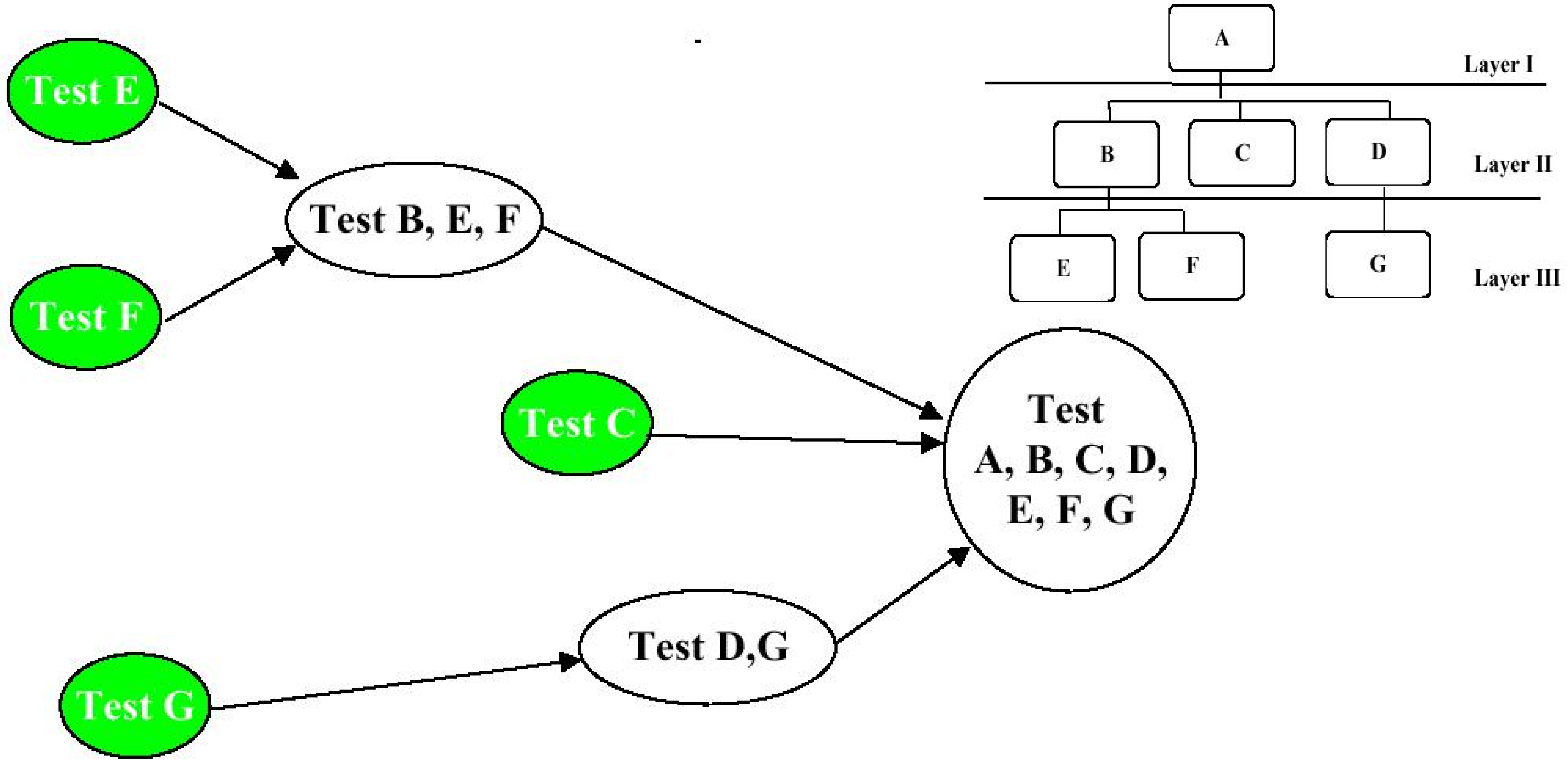


Bottom-up Integration Testing (2)

- As and when code for other module gets ready, these drivers are replaced with the actual module.
- In this approach, lower level modules are tested extensively thus make sure that highest used module is tested properly.

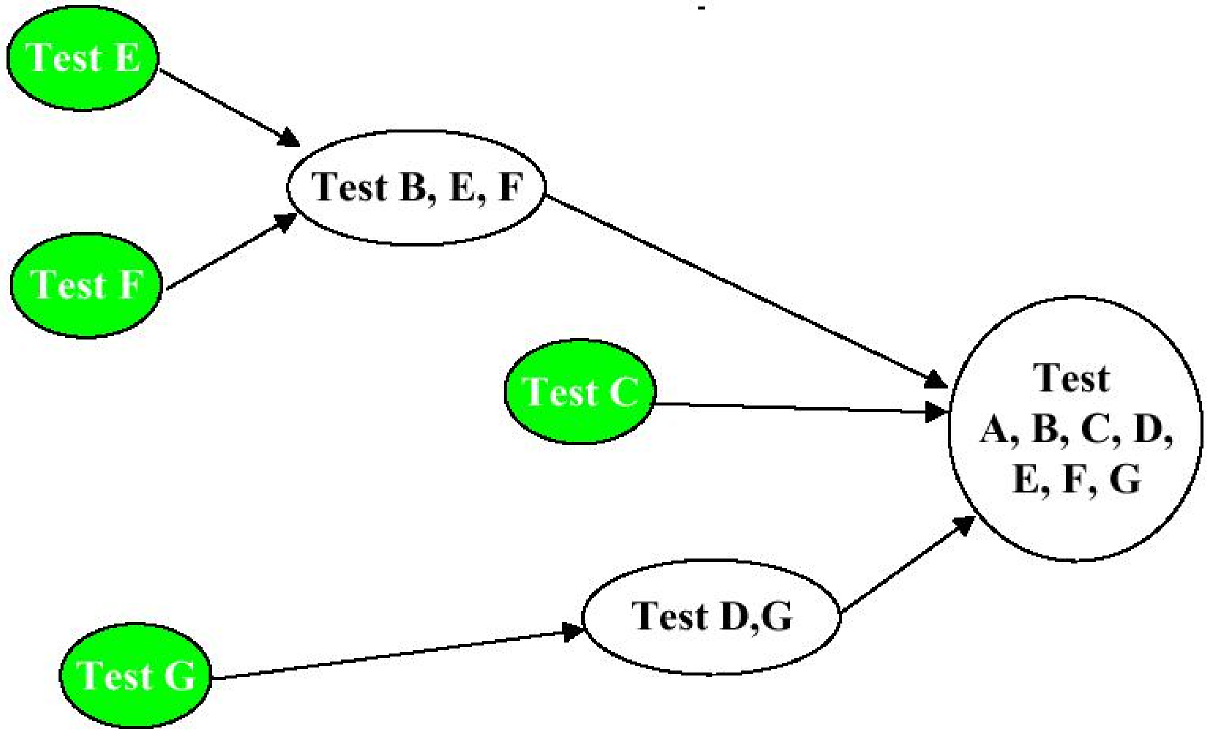


Bottom-up Testing Graphical Representation



Comments on Graphical Representation

- Modules E and F are tested. Then modules B, E, F are tested.
- Module G is tested. Then modules D and G are tested.
- Module C is tested.
- Finally – modules A, B, C, D, E, F, G are tested.



Advantages of Bottom-up Testing

- Behavior of the interaction points are crystal clear, as components are added in the controlled manner and tested repetitively.

- Appropriate for applications where bottom up design methodology is used.

Disadvantages of Bottom-up Testing

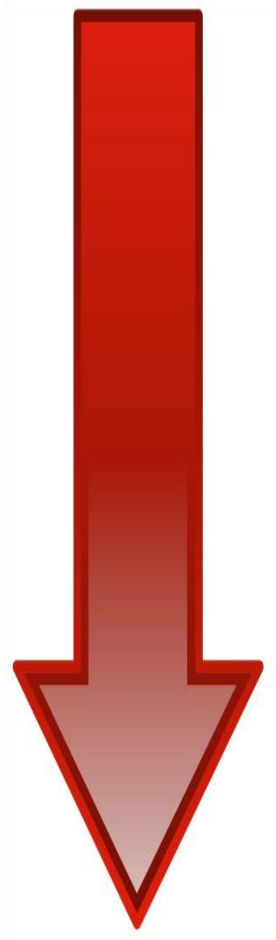
- Writing and maintaining test drivers is more difficult than writing stubs.

  *Stub – the dummy modules that simulates the low level modules.

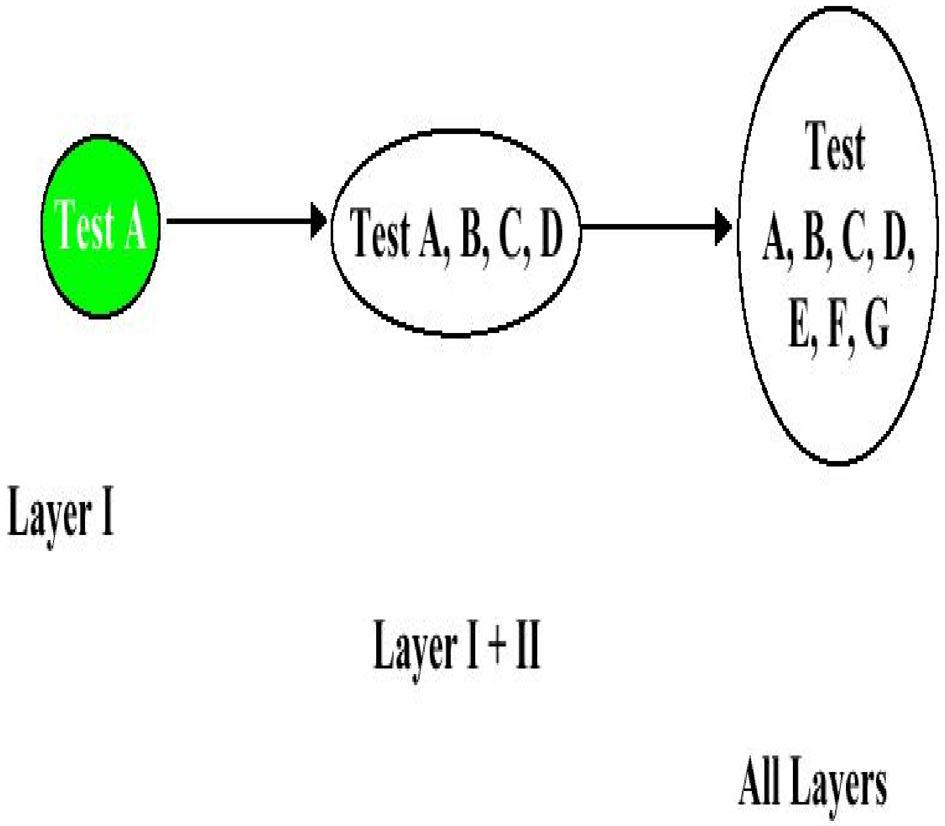
- This approach is not suitable for the software development using top-down approach.

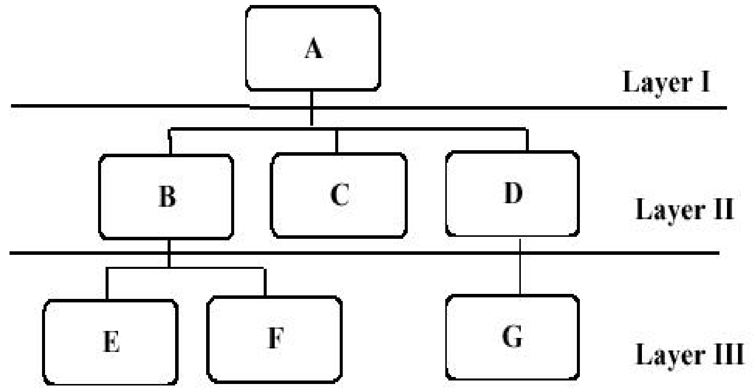
Top-down Testing

- Top down integration testing is an incremental integration testing technique which begins by testing the top level module and and progressively adds in lower level module one by one.
- Lower level modules are normally simulated by stubs which mimic functionality of lower level modules.
- As you add lower level code, you will replace stubs with the actual components.



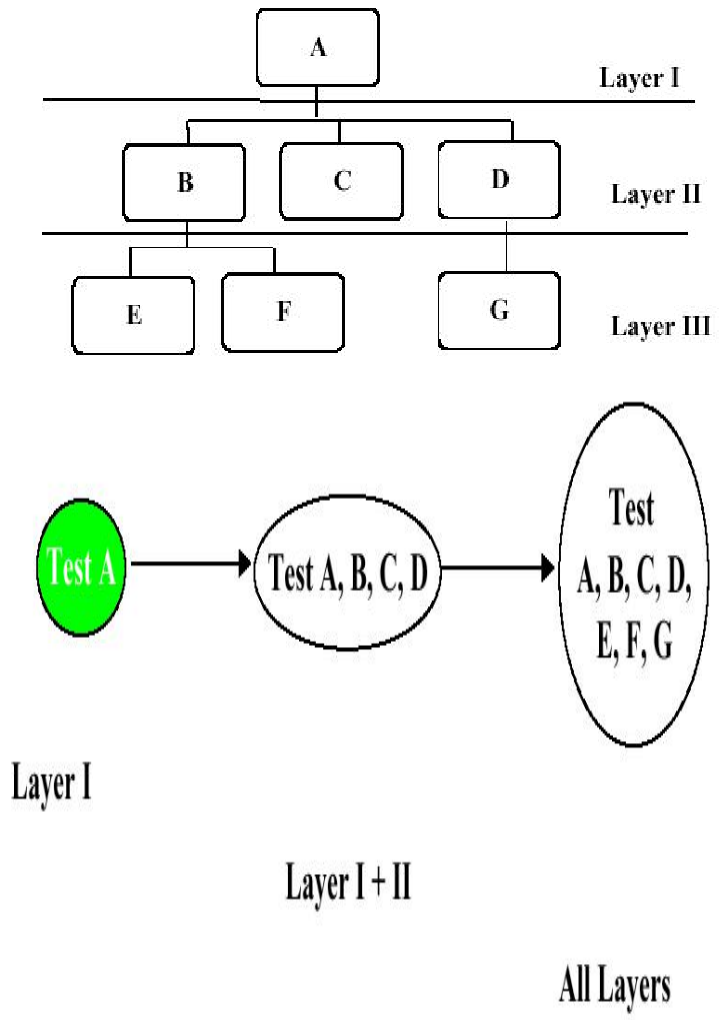
Top-down Testing (2)





Advantages of Top-down Testing

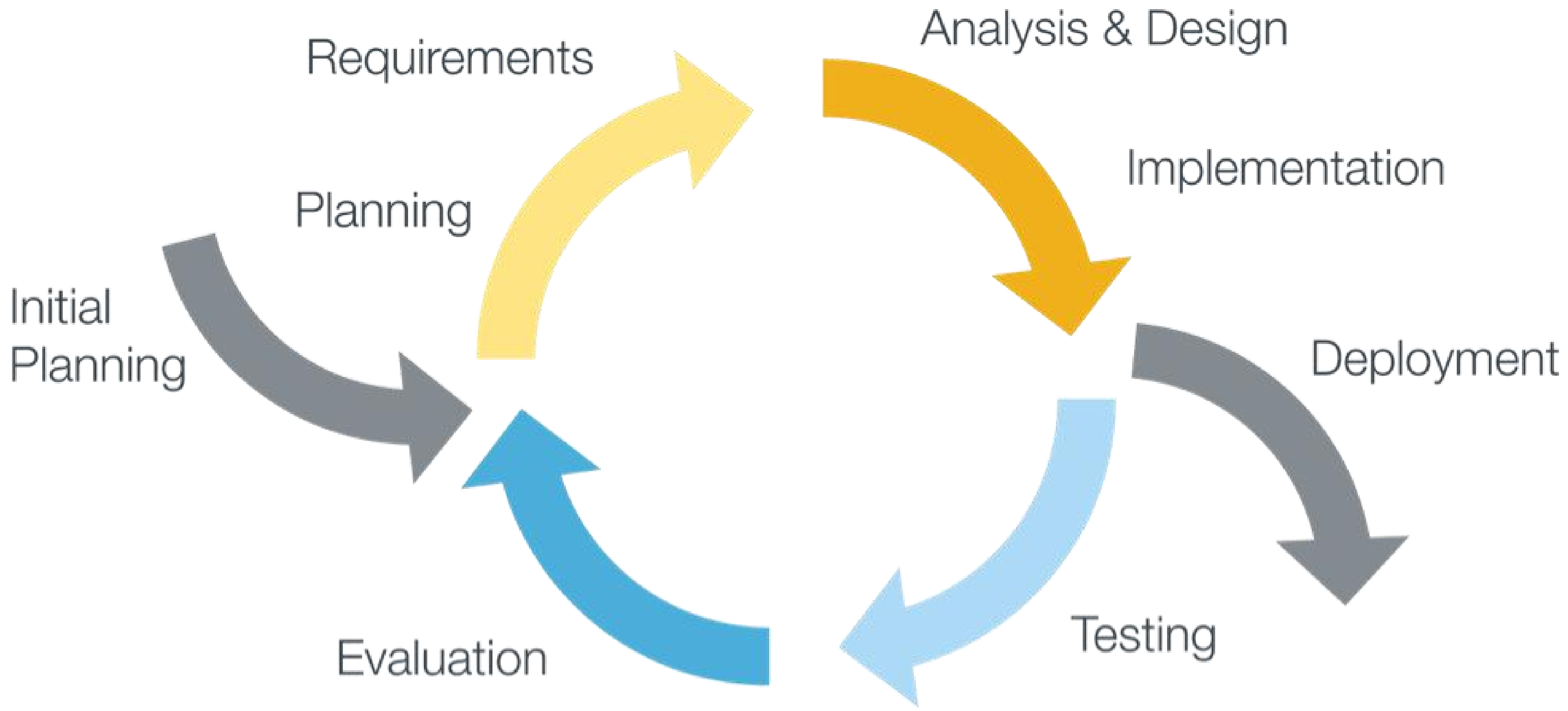
- Driver do not have to be written when top down testing is used.
- It provides early working module of the program and so design defects can be found and corrected early.



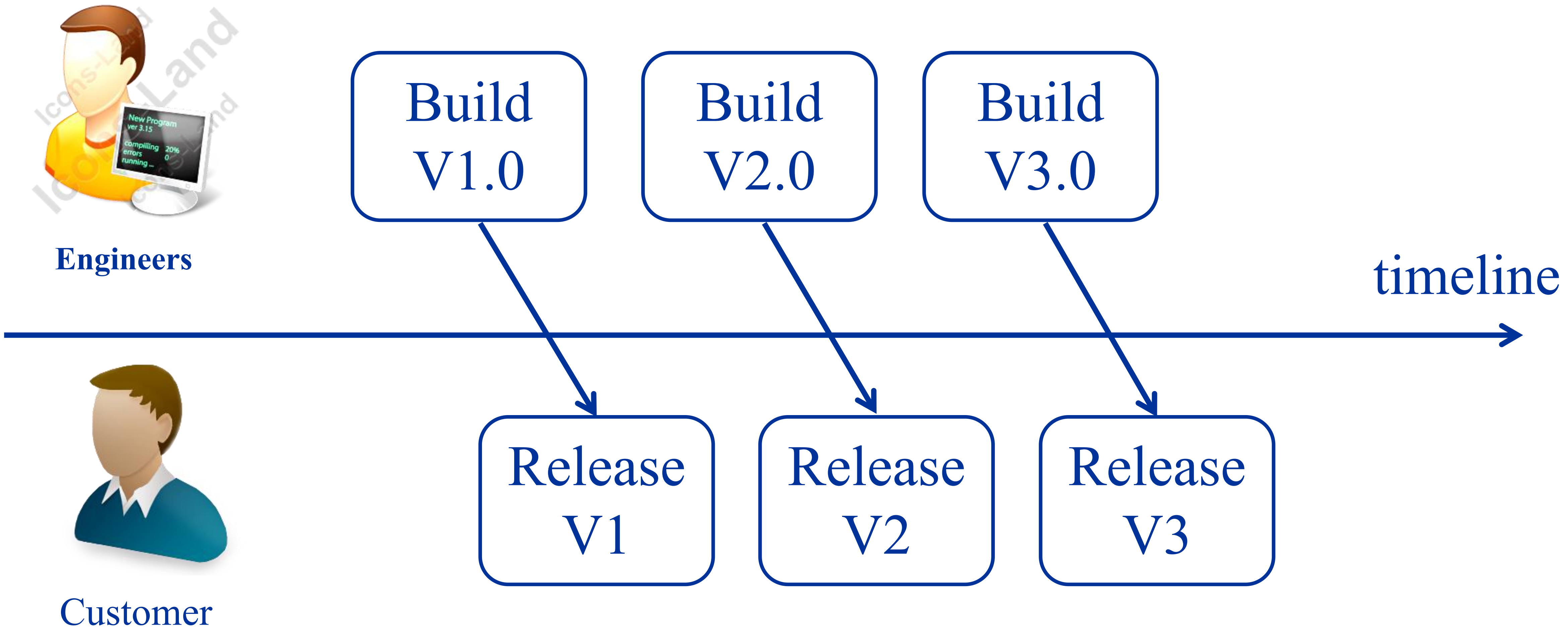
Disadvantages of Top-down Testing

- Stubs have to be written with utmost care as they will simulate setting of output parameters.
- It is difficult to have other people or third parties to perform this testing, mostly developers will have to spend time on this.

Iterative Development



Iterative Development (2)



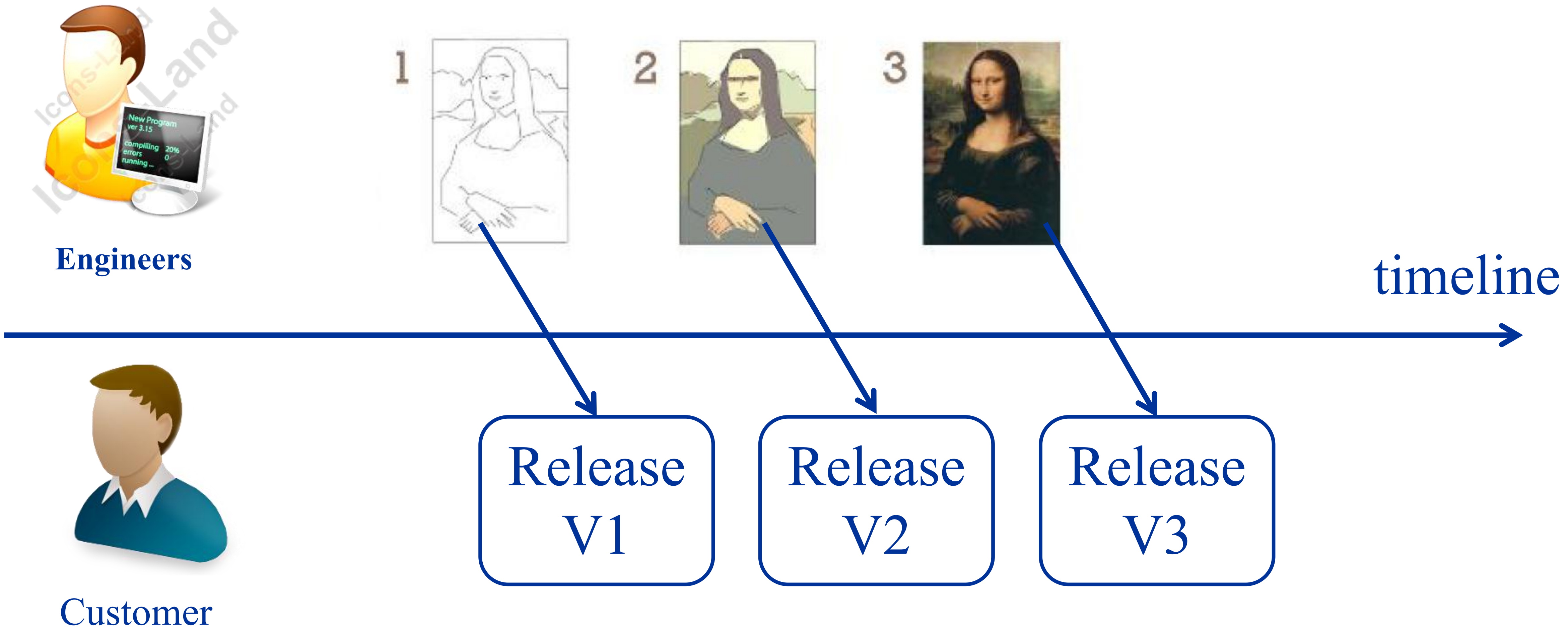
Iterative Model

Iterative Model

woman
in pastoral
setting
1
2
3

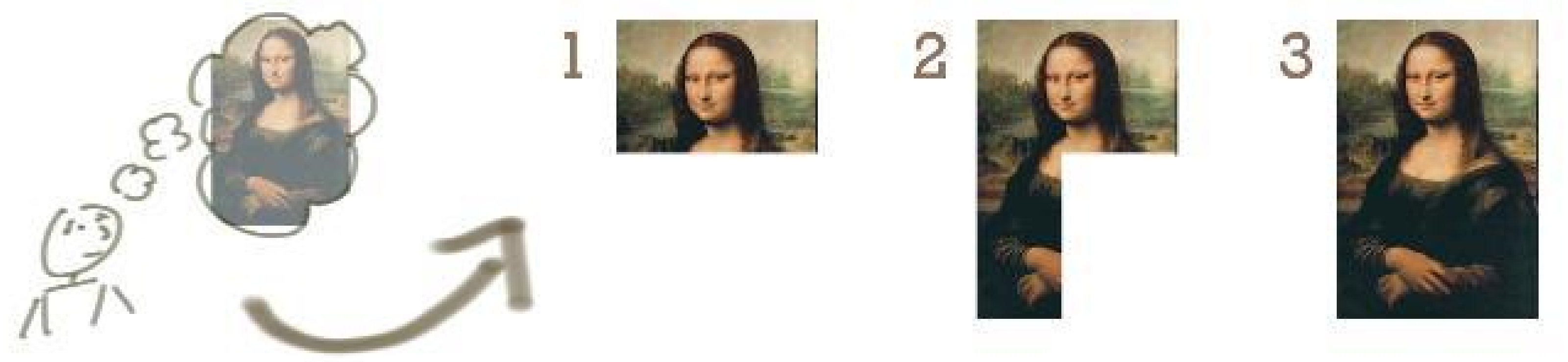
Iterative: The controlled reworking of part of a system to remove mistakes or make improvements.

Iterative Model (2)



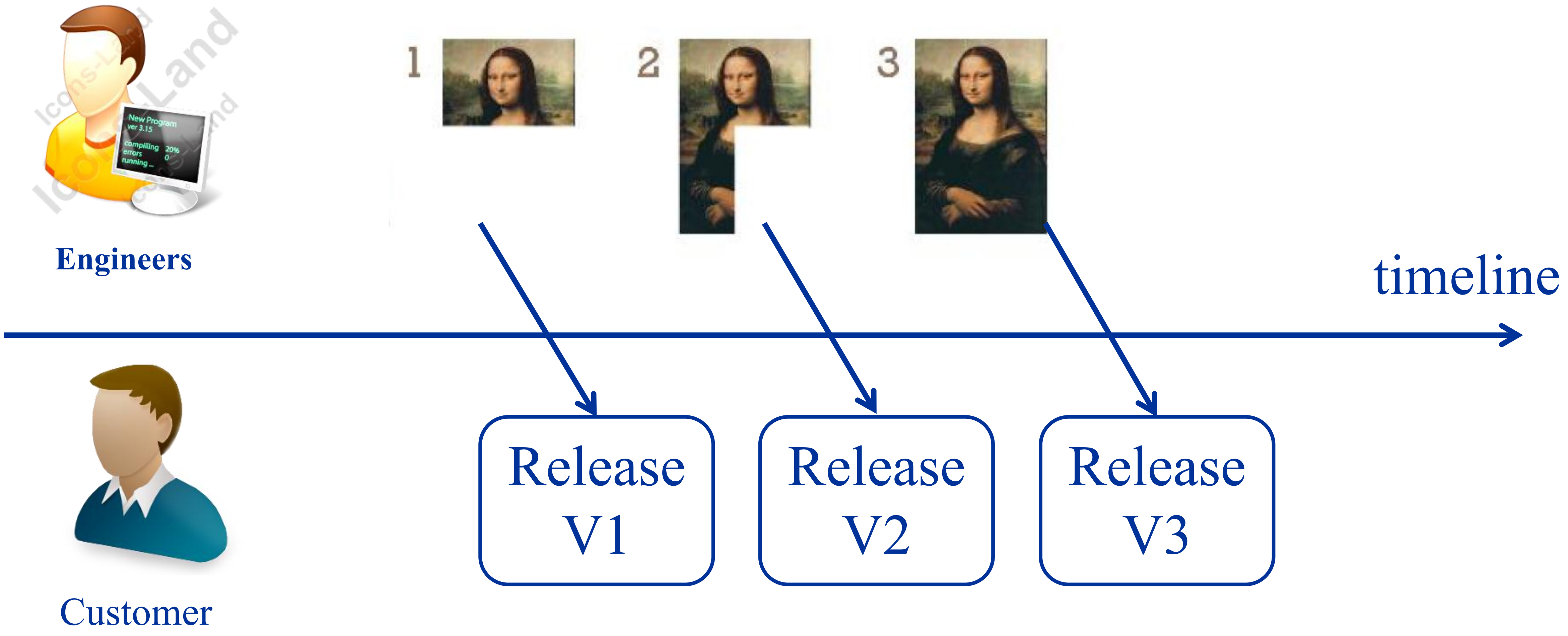
Incremental Model

Incremental Model



Incremental: Making progress in small steps to get early tangible results.

Incremental Model (2)



Top-down And Button-up Testing: Conclusion

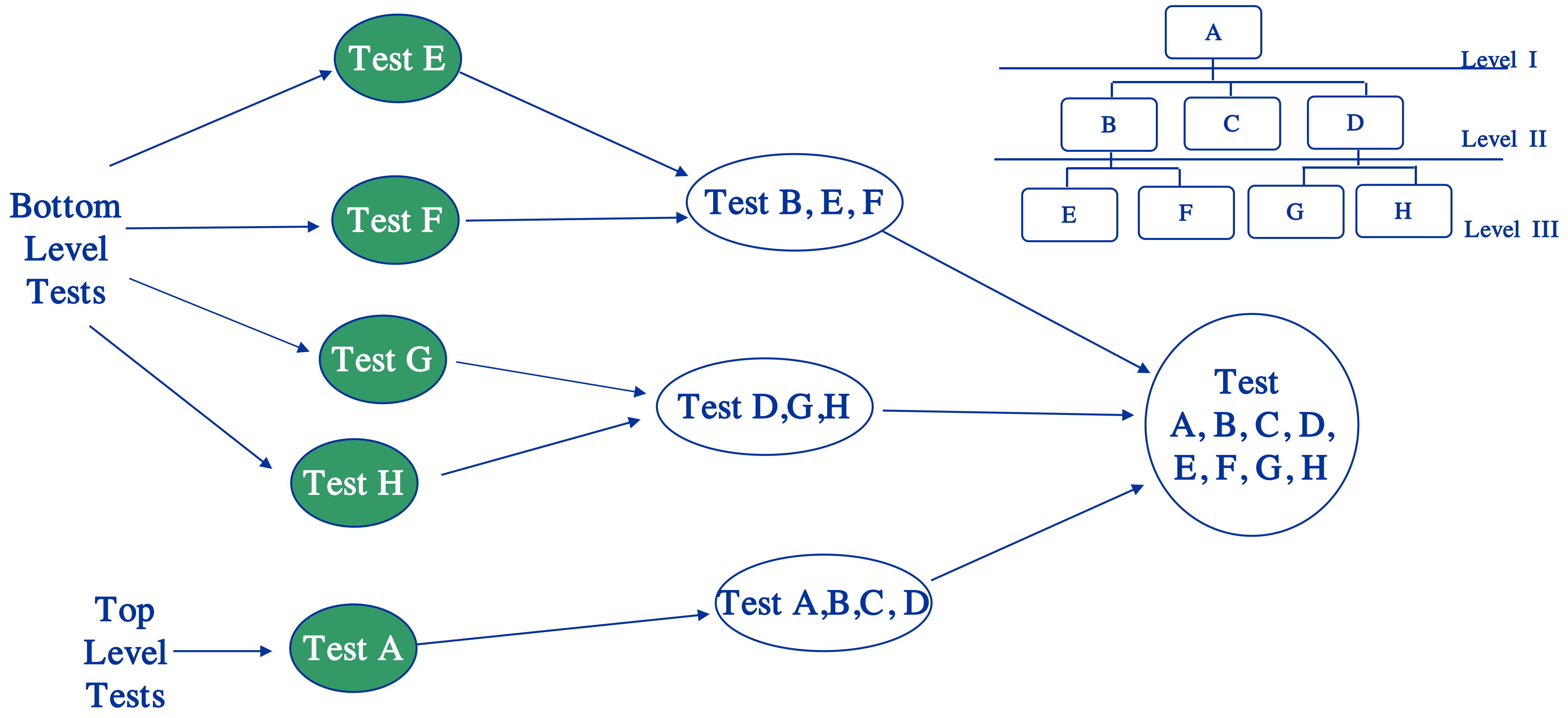
- You'll probably use a combination of these two techniques.

- It's called Sandwich testing strategy.

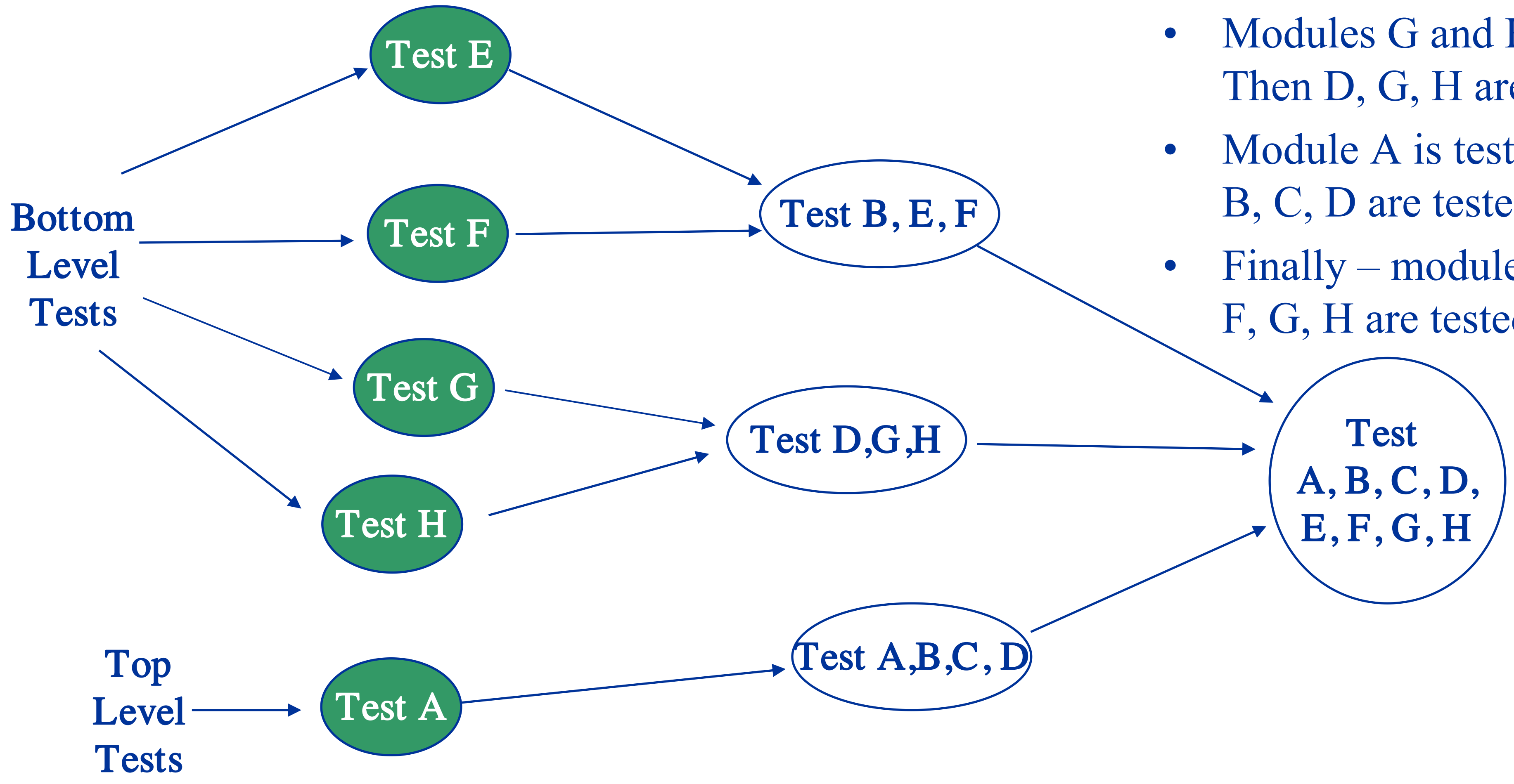
Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- The system is view as having three layers:
  - A target layer in the middle;
  - A layer above the target;
  - A layer belove the target;
  - Testing converges as the target layer;
- How do you select the target layer if there are more than 3 layer?
  - Heuristic: Try to minimize the number of stubs and drivers.

Graphical Representation of Sandwich Testing Strategy



Graphical Representation of Sandwich Testing Strategy



- Modules E and F are tested. Then B, E, F are tested.
- Modules G and H are tested. Then D, G, H are tested.
- Module A is tested. Then A, B, C, D are tested.
- Finally – modules A, B, C, D, F, G, H are tested.

Comments on Graphical Representation

- Modules E and F are tested. Then B, E, F are tested.
- Modules G and H are tested. Then D, G, H are tested.
- Module A is tested. Then A, B, C, D are tested.
- Finally – modules A, B, C, D, F, G, H are tested.

Sandwich Testing Strategy: Conclusions

- Top and Bottom Layer Tests can be done in parallel.
- Does not test the individual subsystems thoroughly before integration.

Sandwich Testing Strategy

- How do you select the target layer if there are more than 3 layer?

Continuous Integration

- Pioneered by Martin Fowler; part of Extreme Programming
- Ten principles:
  - maintain a single source repository
  - automate the build
  - make your build self-testing
  - everyone commits to mainline every day
  - every commit should build mainline on an integration machine
  - keep the build fast
  - test in a clone of the production environment
  - make it easy for anyone to get the latest executable
  - everyone can see what's happening
  - automate deployment

CI - Daily builds

"Automate the build."

- **daily build**: Compile working executable on a daily basis
  - allows you to test the quality of your integration so far
  - helps morale; product "works every day"; visible progress
  - best done *automated* or through an easy script
  - quickly catches/exposes any bug that breaks the build

- **Continuous Integration (CI) server**: An external machine that automatically pulls down your latest repo code and fully builds all resources.
  - If anything fails, contacts your team (e.g. by email).
  - Ensures that the build is never broken for long.

Build from command line

- An Android project needs a `build.xml` to be used by Ant.
  - This file allows your project to be compiled from the command line, making automated builds possible.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MainActivity" default="help">
   <property file="ant.properties" />
   <import file="${sdk.dir}/tools/ant/build.xml" />
   <taskdef name="findbugs" classname="edu.umd.cs.findbugs.anttask.FindBugsTask"/>
   <target name="findbugs">
      <findbugs home="${findbugs.home}" output="xml" outputFile="findbugs.xml" excludeFilter="findbugs-exclude.xml">
         <auxClasspath path="${android.jar}" />
         <auxClasspath path="${rt.jar}" />
         <auxClasspath path="libs\android-support-v4.jar" />
         <class location="${out.dir}" />
      </findbugs>
   </target>

   <taskdef resource="checkstyletask.properties" classpath="${basedir}/libs/checkstyle-5.6-all.jar"/>
   <checkstyle config="sun_checks.xml" failonviolation="false">
      <fileset dir="src" includes="**/*.java"/>
      <formatter type="plain"/>
      <formatter type="xml" toFile="checkstyle-result.xml"/>
   </checkstyle>
</project>
```

CI - Automated tests

"Make your build self-testing."

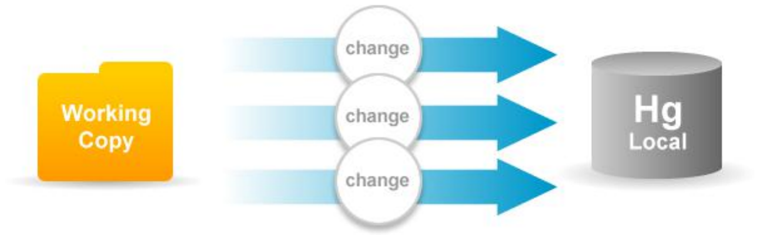
- **automated tests**: e.g. Tests that can be run from the command line on your project code at any time.
  - can be unit tests, coverage, static analysis / style checking, ...

- **smoke test**: A quick set of tests run on the daily build.
  - NOT exhaustive; just sees whether code "smokes" (breaks)
  - used (along with compilation) to make sure daily build runs

CI – Daily commits

"Everyone commits to the mainline every day."

- **daily commit**: Submit work to main repo at end of each day.
  - Idea: Reduce merge conflicts; avoid later integration issues.
  - This is the key to "continuous integration" of new code.



  - *Caution:* Don't check in faulty code (does not compile, does not pass tests) just to maintain the daily commit practice.
  - If your code is not ready to submit at end of day, either submit a coherent subset or be flexible about commit schedule.

Steps of Integration Testing

- Select component to test and unit test the classes of the component.
- Put selected component into system. Do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs).
- Define test cases that exercise all uses cases with the selected component

Steps of Integration Testing (2)

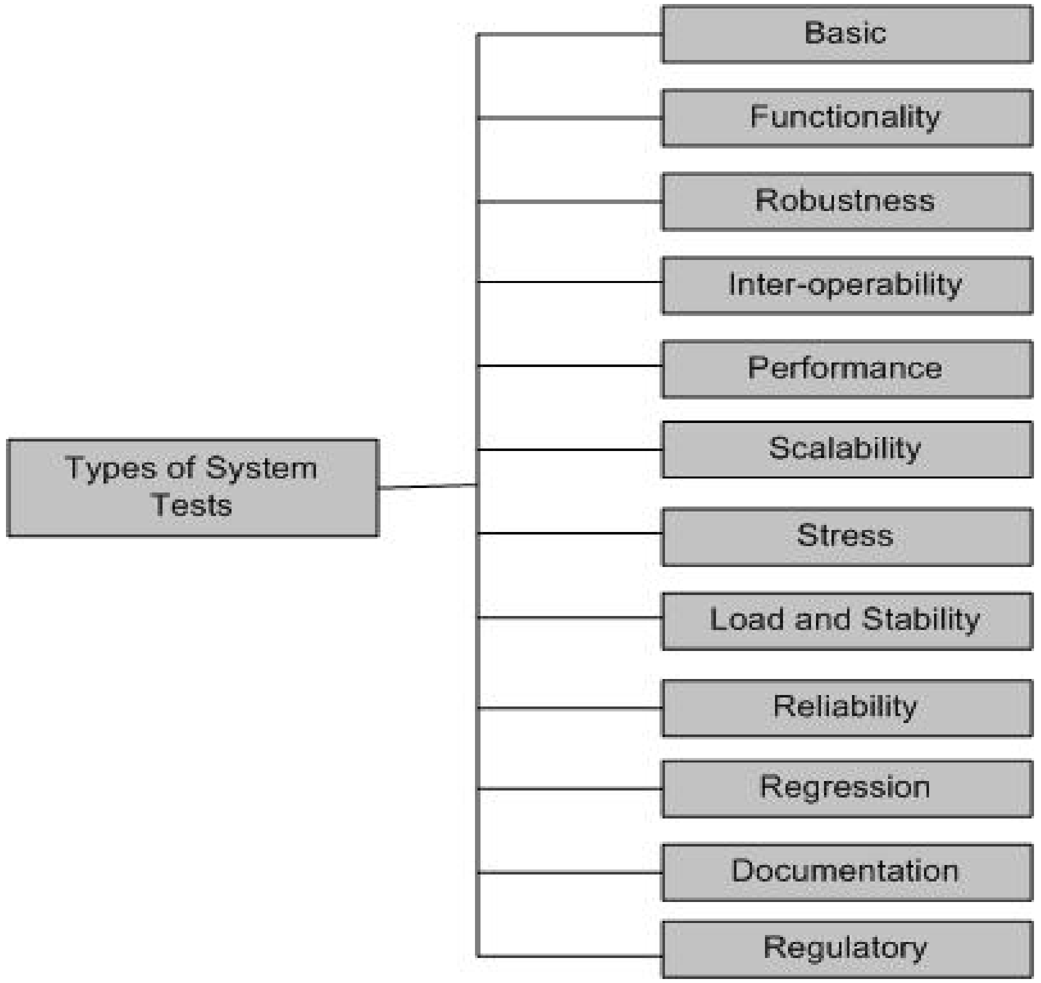
- Define test cases that exercise the selected component
- Execute test cases and once again with another component…
- The primary goal of integration testing is to identify errors in the interfaces, connections, the (current) component configuration.

Main Point: Incremental Integration

- All the units of a system must **be integrated consecutively and integrated in step by step process** by incrementing the levels of testing at one end to other end.

System Testing

Taxonomy of System Tests



Taxonomy of System Tests (2)

- **Basic tests** provide an evidence that the system can be installed, configured and be brought to an operational state
- **Functionality tests** provide comprehensive testing over the full range of the requirements, within the capabilities of the system
- **Robustness tests** determine how well the system recovers from various input errors and other failure situations

Taxonomy of System Tests (3)

- **Inter-operability tests** determine whether the system can inter-operate with other third party products

- **Performance tests** measure the performance characteristics of the system, e.g., throughput and response time, under various conditions

- **Scalability tests** determine the scaling limits of the system, in terms of user scaling, geographic scaling, and resource scaling

Taxonomy of System Tests (4)

- **Stress tests** put a system under stress in order to determine the limitations of a system and, when it fails, to determine the manner in which the failure occurs

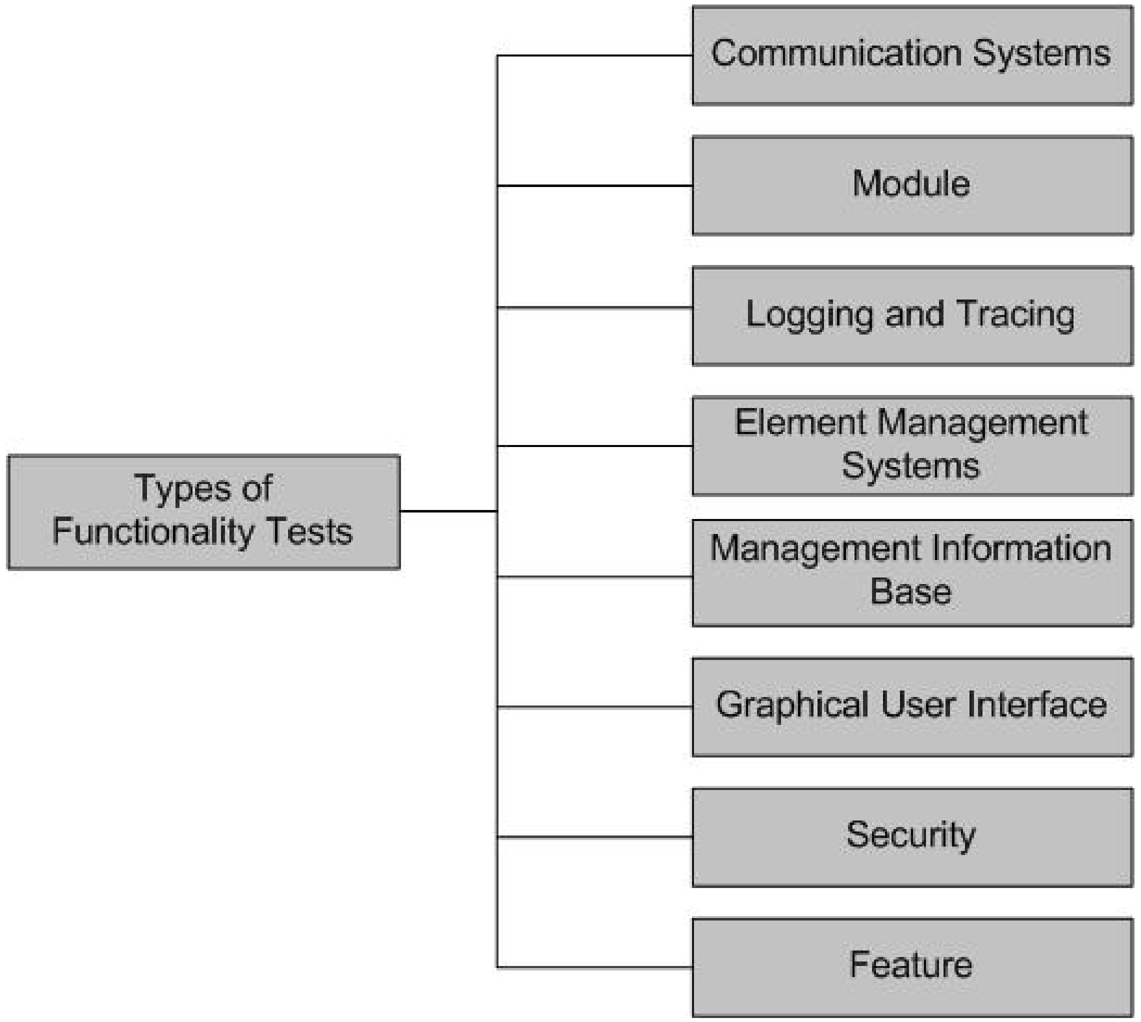
- **Reliability tests** measure the ability of the system to keep operating for a long time without developing failures

Taxonomy of System Tests (5)

- **Regression tests** determine that the system remains stable as it cycles through the integration of other subsystems and through maintenance tasks

- **Documentation tests** ensure that the system's user guides are accurate and usable

Functionality Tests



Functionality Tests (2)

- Communication System Tests
  - These tests are designed to verify the implementation of the communication systems as specified in the requirements specification;

Functionality Tests (3)

- Module Tests
  - Module Tests are designed to verify that all the modules function individually as desired within the systems
  - The idea here is to ensure that individual modules function correctly within the whole system.
    - For example, an Internet router contains modules such as line cards, system controller, power supply, and fan tray. Tests are designed to verify each of the functionalities

Functionality Tests (4)

- Logging and Tracing Tests
  - Logging and Tracing Tests are designed to verify the configurations and operations of logging and tracing
  - This also includes verification of “flight data recorder: non-volatile Flash memory” logs when the system crashes
- Tests may be designed to calculate the impact on system performance when all the logs are enable.

Functionality Tests (5)

- Graphical User Interface Tests
  - Tests are designed to look-and-feel the interface to the users of an application system
  - Tests are designed to verify different components such as icons, menu bars, dialog boxes, scroll bars, list boxes, and radio buttons
  - The GUI can be utilized to test the functionality behind the interface, such as accurate response to database queries
  - Tests the usefulness of the on-line help, error messages, tutorials, and user manuals

Functionality Tests (6)

- The usability characteristics of the GUI is tested, which includes the following
  - ***Accessibility:*** Can users enter, navigate, and exit with relative ease?
  - ***Responsiveness:*** Can users do what they want and when they want in a way that is clear?
  - ***Efficiency:*** Can users do what they want to with minimum number of steps and time?
  - ***Comprehensibility:*** Do users understand the product structure with a minimum amount of effort?

Functionality Tests (7)

- Security Tests
  - Security tests are designed to verify that the system meets the security requirements
    - Confidentiality
      - It is the requirement that data and the processes be protected from unauthorized disclosure
    - Integrity
      - It is the requirement that data and process be protected from unauthorized modification
    - Availability
      - It is the requirement that data and processes be protected form the denial of service to authorized users
  - Security test scenarios should include negative scenarios such as misuse and abuse of the software system

Functionality Tests: Security Tests (2)

- Useful types of security tests includes the following:
  - Verify that only authorized accesses to the system are permitted
  - Verify the correctness of both encryption and decryption algorithms for systems where data/messages are encoded.
  - Verify that illegal reading of files, to which the perpetrator is not authorized, is not allowed
  - Ensure that virus checkers prevent or curtail entry of viruses into the system
  - Ensure that the system is available to authorized users when a zero-day attack occurs
  - Try to identify any “backdoors” in the system usually left open by the software developers

Functionality Tests (8): Feature Tests

- Feature Tests
  - These tests are designed to verify any additional functionalities which are defined in requirement specification but not covered in the functional category discussed