SOFTWARE SYSTEM QUALITY

LECTURE # 15

SOFTWARE TESTING-II
(Dynamic)
BLACK BOX TESTING

chenbo@etao.net

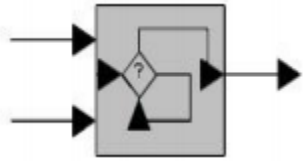
BLACK BOX TESTING TECHNIQUES

- Equivalence Class Partitioning Testing
- Boundary Value Testing
- Fuzzy Testing
- Omission Testing
- Comparison Testing
- End to End Testing
- Localization Testing
- Globalization Testing
- Error Handling Testing
- Integration Testing
- Sandwich Testing
- Security Testing
- Compatibility Testing
- Null Case Testing
- Volume Testing
- Load Testing
- Stress Testing
- Resource Testing
- Requirements/Specification Testing
- Configuration Testing
- Documentation Testing
- Smoke Testing
- Usability Testing
- Exploratory Testing
- Button Press Testing
- State Transition Testing
- Installation Testing

Dynamic Testing

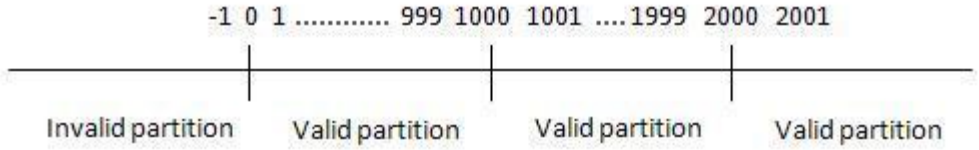
- **black-box/functional testing**
  - verifies the correct handling of the external functions provided by the software or whether the observed behavior conforms to user expectations or product specifications
  - The emphasis is on reducing the chances of encountering functional problems by target customers.

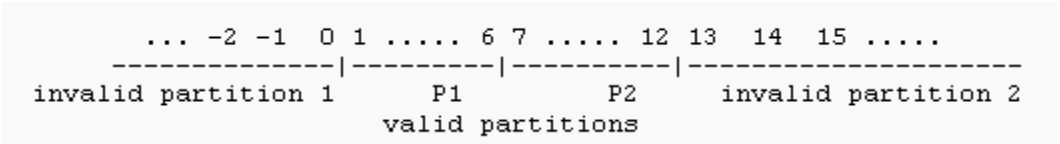
- **white/clear-box/structural testing**
  - verifies the correct implementation of internal units, structures and relations among them
  - When white box testing is performed ,failures related to internal implementations can be observed, leading to corresponding faults being detected and removed.



Equivalence Class Partitioning Testing

- Equivalence Partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived
- An ideal test case single handedly uncovers a class of errors e.g incorrect processing of all character data that might otherwise require many cases to be executed before the general error is observed.
- Equivalence Partitioning strives to define the test case that uncovers classes of errors, there by reducing the total number of test cases that must be developed.
- An equivalence class represents a set of valid or invalid states for input conditions.





Equivalence Class Partitioning Testing

- Equivalence classes can be defined according to the following guidelines;
  - **If an input condition specifies a range one valid and two invalid equivalence classes are defined.**
  - **If an input condition specifies a specific value one valid and two invalid equivalence classes are defined.**
  - **If an input condition specifies a member of a set one valid and one invalid equivalence class are defined.**
  - **If an input condition is Boolean, one valid and one invalid class are defined.**

Tasks | Test Data Table

Action	Type	SameData		DiffAmount	
		In	Expected	In	Expected
objMoney = new Money(amount, currency)					
amount	int	500		-1000	
currency	java.lang.String	"USD"		"EUR"	
objMoney	junit.samples.money.Mo...				
<expected exception>	Throwable				
objMoney1 = new Money(amount, currency)					
amount	int	objMoney.amount()		objMoney.amount()+1	
currency	java.lang.String	objMoney.currency()		objMoney.currency()	
objMoney1	junit.samples.money.Mo...				
<expected exception>	Throwable				
retValue = objMoney.equals(anObject)					
anObject	java.lang.Object	objMoney1		objMoney1	
retValue	boolean		true		false
<expected exception>	Throwable				

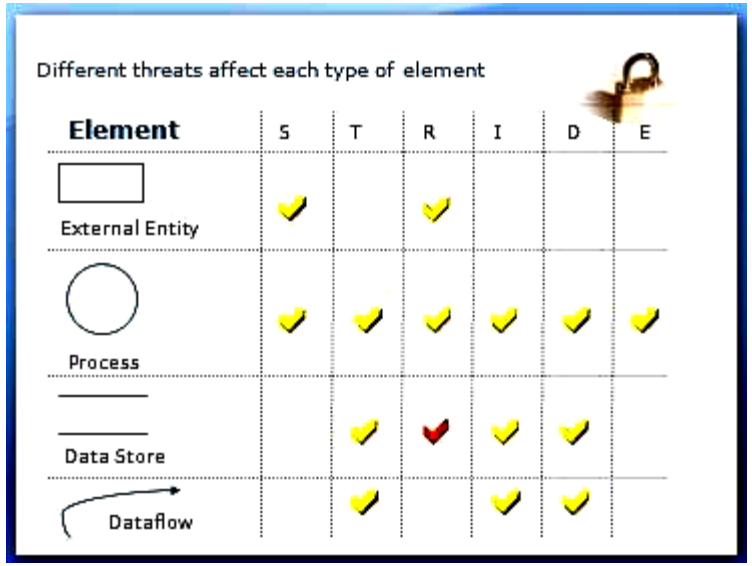
Boundary Value Testing

(Specific case of Equivalence Class Partitioning Testing)

- Boundary value analysis leads to a selection of test cases that exercise bounding values. This technique is developed because a great number of errors tend to occur at the boundary of input domain rather than at the center.
- Tests program response to extreme input or output values in each equivalence class.
- Guideline for BVA are following;
  - If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and below a and b.

Fuzzy Testing

- Fuzzy testing or fuzzing is a software testing technique, often automated or semi-automated, that involves providing invalid or unexpected data to the inputs of a computer program. The program is then monitored for exceptions such as crashes or failing built-in code assertions or for finding potential memory leaks.
- The term first originates from a class project at the University of Wisconsin 1988 although similar techniques have been used in the field of quality assurance, where they are referred to as robustness testing or negative testing.



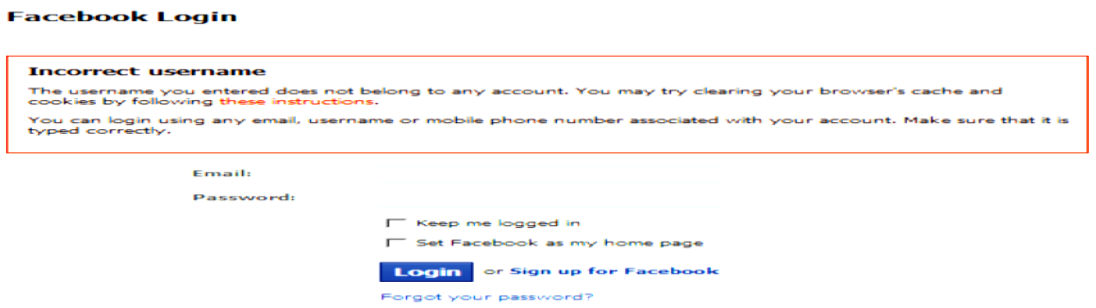
Omission Testing

- Omission Testing (also called Missing Case Testing):
- Exposes defects caused inputting cases (scenarios) the developer forgot to handle or did not anticipate
  - A study by Sherman on a released Microsoft product reported that 30% of client reported defects were caused by missing cases.
  - Other studies show that an average of 22 to 54% of all client reported defects are caused by missing cases.

Null Case Testing

Null Testing: (a specific case of Omission Testing, but triggers defects extremely often)

- Exposes defects triggered by no data or missing data.
- Often triggers defects because developers create programs to act upon data, they don't think of the case where the project may not contain specific data types
  - **Example: X,Y coordinate missing for drawing various shapes in Graphics editor.**
  - **Example: Blank file names**



Comparison Testing

- There are some situations in which the reliability of software is absolutely critical. In such applications redundant hardware and software are often used to minimize the possibility of error
- When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specification
- In such situations each version can be tested with the same test data to ensure that all provide identical output
- Then all versions are executed in parallel with real time comparison of results to ensure consistency
- These independent versions form the basis of black box testing technique called **comparison testing or back-to-back testing**

Comparison Testing

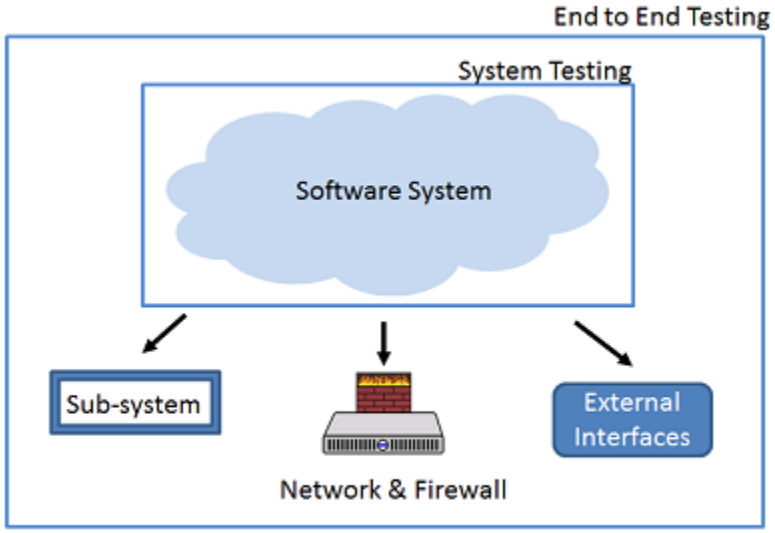
- If the output from each version is the same, it is assumed that all implementations are correct
- If the output is different, each of the application is investigated to determine if the defect in one or more versions is responsible for the difference
- Comparison testing is not fool proof , if the specification from which all versions have been developed is in error, all versions will likely reflect the error
- In addition if each of the independent versions produces identical but incorrect results, comparison testing will fail to detect the error

End to End Testing

- End-to-end testing is a methodology used to test whether the flow of an application is performing as designed from start to finish.
- End-to-end testing involves ensuring that that integrated components of an application function as expected.
- This is basically exercising an entire “workflow”. Although System Testing is similar, in System Testing you do not have to complete the entire workflow but may exercise individual interfaces one at a time.
- For example, a simplified end-to-end testing of an email application might involve logging in to the application, getting into the inbox, opening and closing the mail box, composing, forwarding or replying to email, checking in the sent items and logging out of the application.

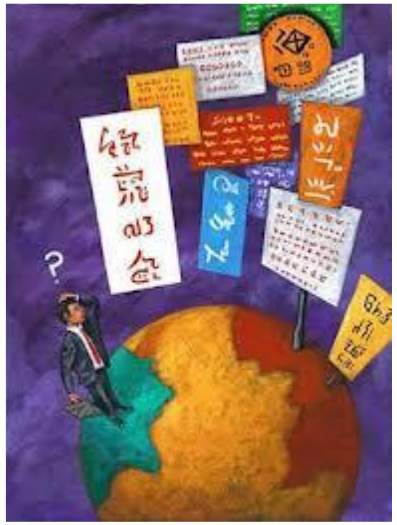
SQA Team

- Unlike System Testing, End-to-End Testing not only validates the software system under test but also checks it's integration with external interfaces. Hence, the name "**End-to-End**".
- End to End Testing is usually executed after functional and system testing. It uses actual production like data and test environment to simulate real-time settings. End-to-End testing is also called **Chain Testing**



Globalization Testing

- **Globalization Testing** is the software testing process for checking if the software can work properly in various culture/locale settings using every type of international input.



Localizing Testing

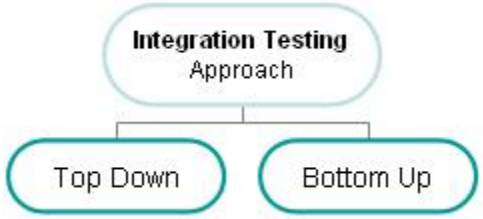
- Localization is the process of customizing a software application that was originally designed for a domestic market so that it can be released in foreign markets.
- This process involves translating all native language strings to the target language and customizing the GUI so that it is appropriate for the target market
- Localization testing checks how well the build has been translated into a particular target language (e.g., Japanese product for Japanese user).
- We should invite the local staff to help our localization testing by checking the quality of translation as well.
  - **Common bugs found from this testing**
  - **Cannot display the correct format**
  - **Functionality is broken**

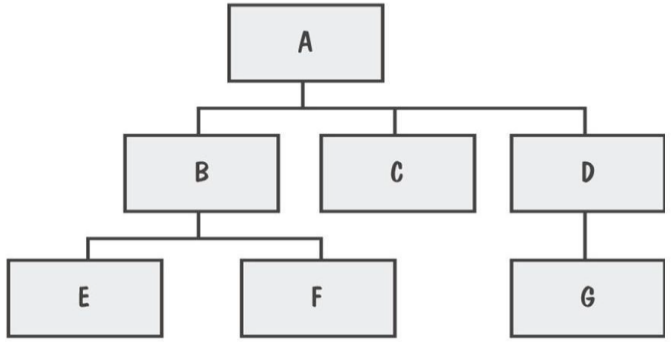
Error Handling Testing

- Error-handling testing determines the ability of the application system to properly process incorrect transactions.
- Specific objectives of error-handling testing include:
  - Determine that all reasonably expected error conditions are recognizable by the application system.
  - Determine that the accountability for processing errors has been assigned
  - Determine that that the procedures provide a high probability that the error will be properly corrected.
- Error-handling Test examples:
  - Produce a representative set of transactions containing errors and enter them into the system to determine whether the application can identify the problems.
  - Enter improper master data, such as prices or employee pay rates, to determine if errors that will occur repetitively are subjected to greater scrutiny(inspection/analysis) than those causing single error results.

Integration Testing

- Integration testing (sometimes called Integration and Testing, abbreviated "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 and validation 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.

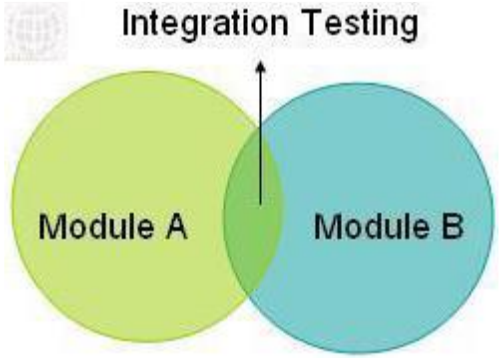




Integration Testing Types

- Big Bang Integration

- Incremental Integration
  - Top Down Integration
  - Bottom Up Integration
  - Sandwich Testing

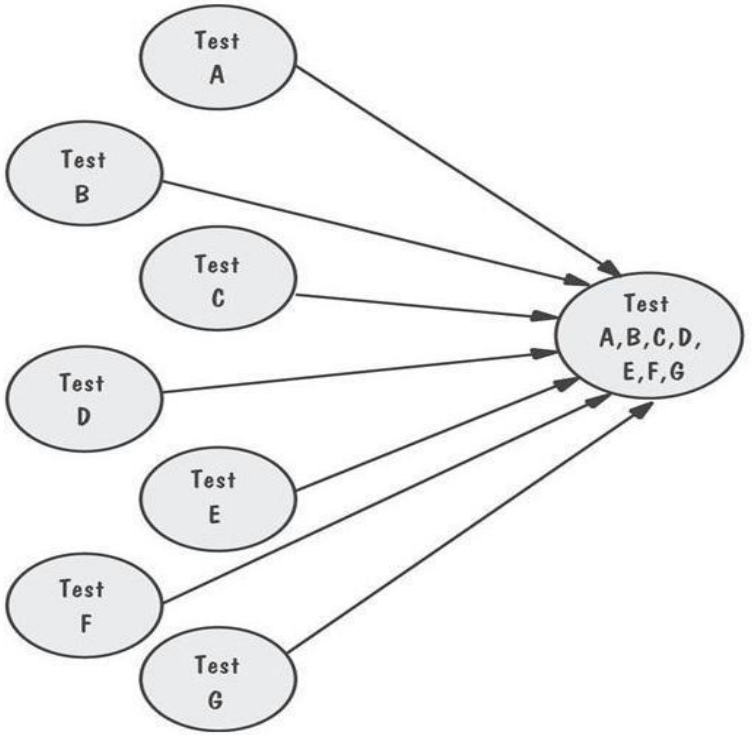


Big Bang Integration

- In this approach, all or most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing.
- The Big Bang method is very effective for saving time in the integration testing process. However, the major drawback of Big Bang Integration is to find the actual location of error.

Big Bang Integration

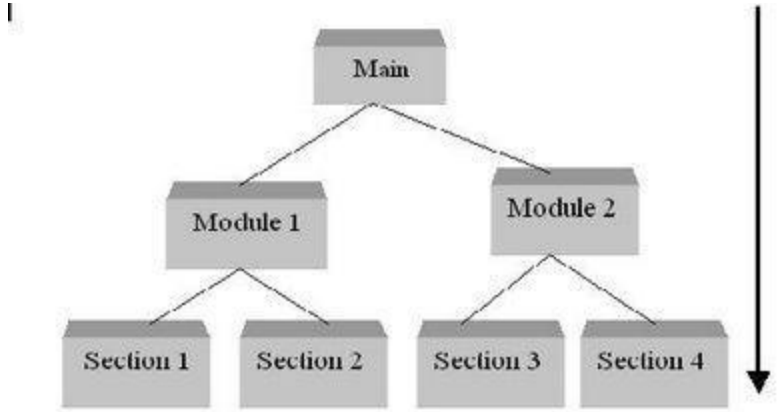
- Requires both Stubs and Drivers to test the independent components
- Need Many Drivers
- Need Many Stubs
- Hard to isolate faults

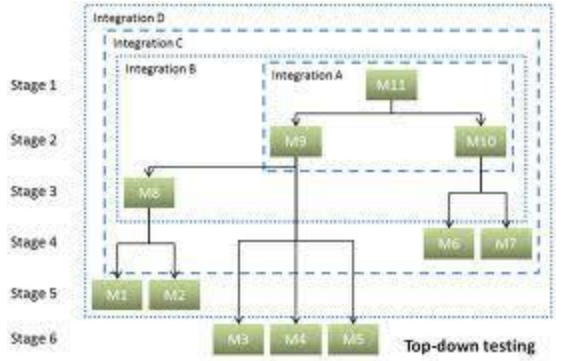


Incremental Integration

TOP DOWN INTEGRATION

- Top Down Testing is an approach to integrated testing where the top integrated modules are tested and the branch of the module is tested step by step until the end of the related module.



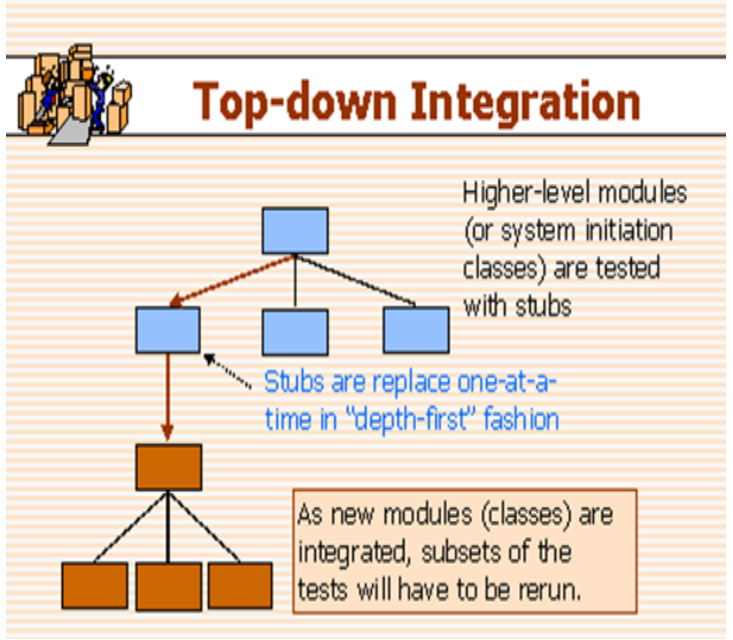


Incremental Integration

TOP DOWN INTEGRATION

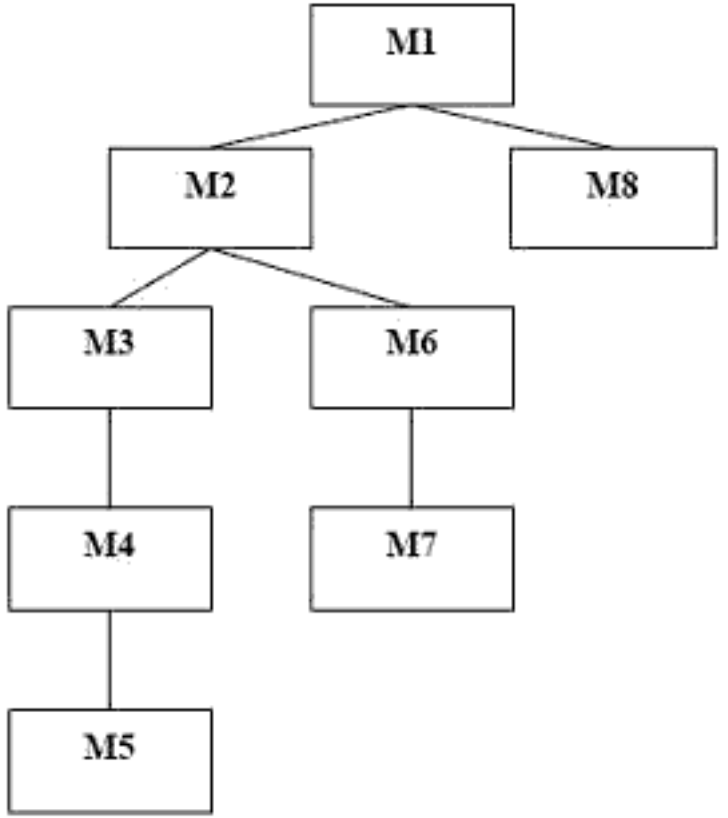
Top down integration is performed in a series of steps:

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



Incremental Integration

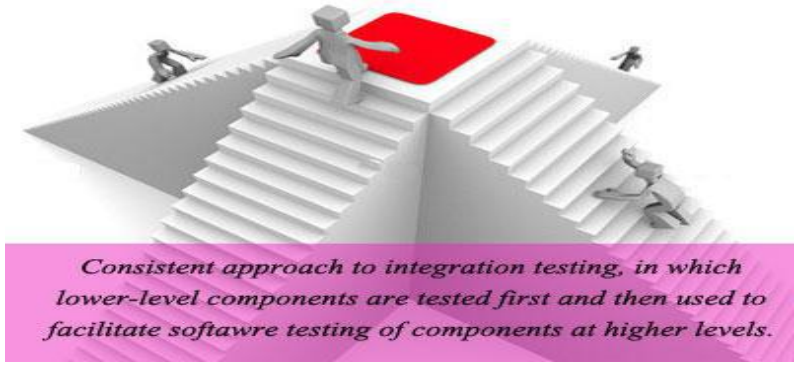
- In depth first approach all modules on a control path are integrated first. See the fig. on the right. Here sequence of integration would be (M1, M2, M3), M4, M5, M6, M7, and M8.
- In breadth first all modules directly subordinate at each level are integrated together. Using breadth first for this fig. the sequence of integration would be (M1, M2, M8), (M3, M6), M4, M7, andM5.



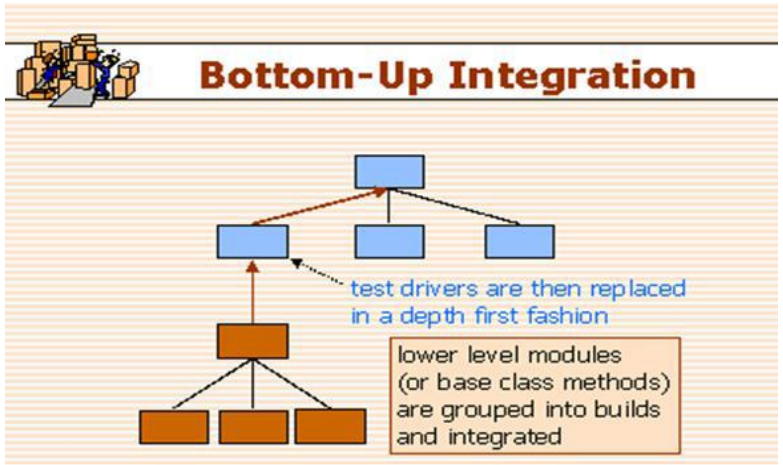
Incremental Integration

BOTTOM UP INTEGRATION

- Bottom Up Testing is an approach to integrated testing where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested.
- All the bottom or low-level modules, procedures or functions are integrated and then tested. After the integration testing of lower level integrated modules, the next level of modules will be formed and can be used for integration testing. This approach is helpful only when all or most of the modules of the same development level are ready.



Consistent approach to integration testing, in which lower-level components are tested first and then used to facilitate softawre testing of components at higher levels.

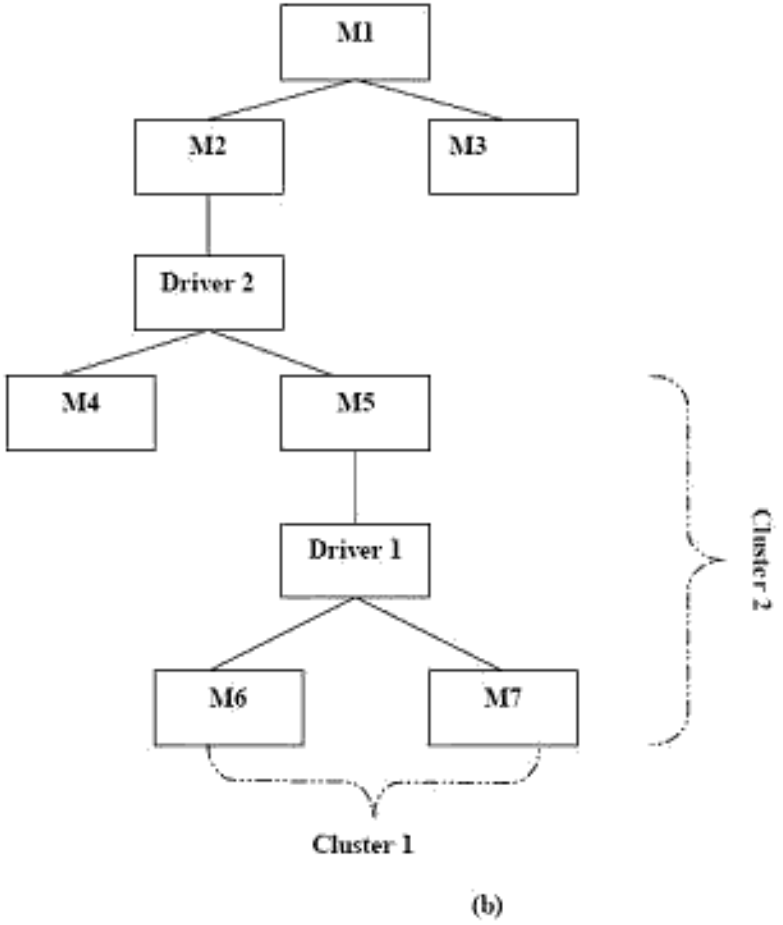


Incremental Integration

BOTTOM UP INTEGRATION

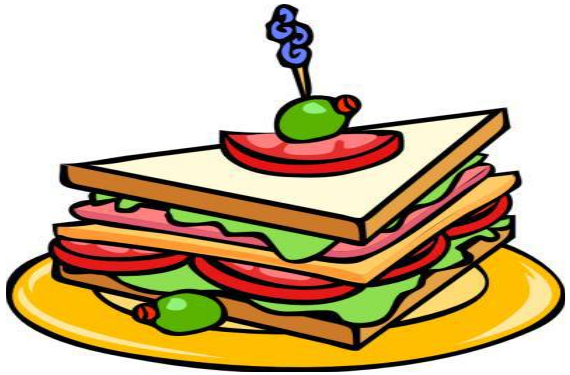
- Bottom up integration is performed in a series of steps:

1. Low level components are combined into clusters.
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.

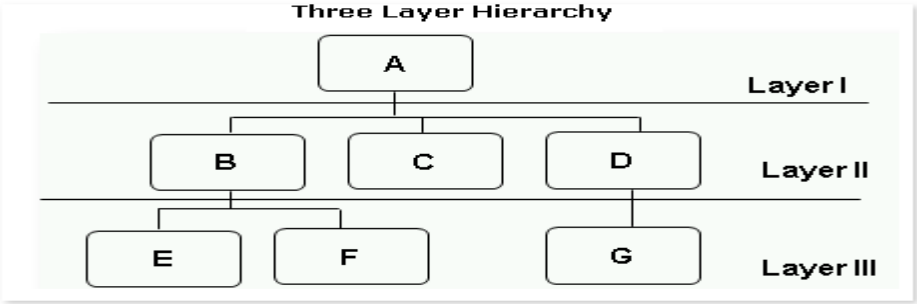


Sandwich Testing

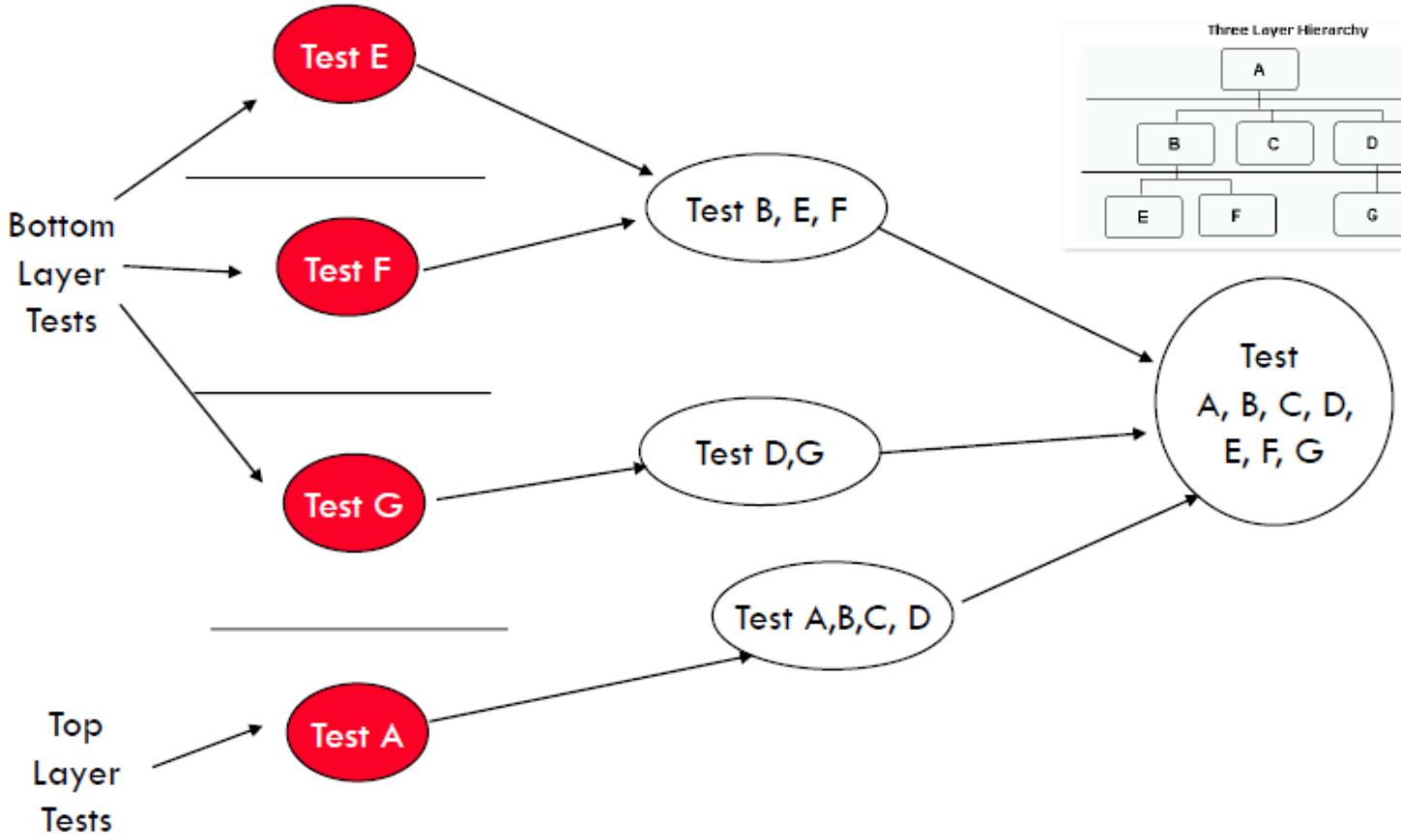
- Sandwich Testing is an approach to combine top down testing with bottom up testing.
- *The system is viewed as having three layers*
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
  - Testing converges at the target layer



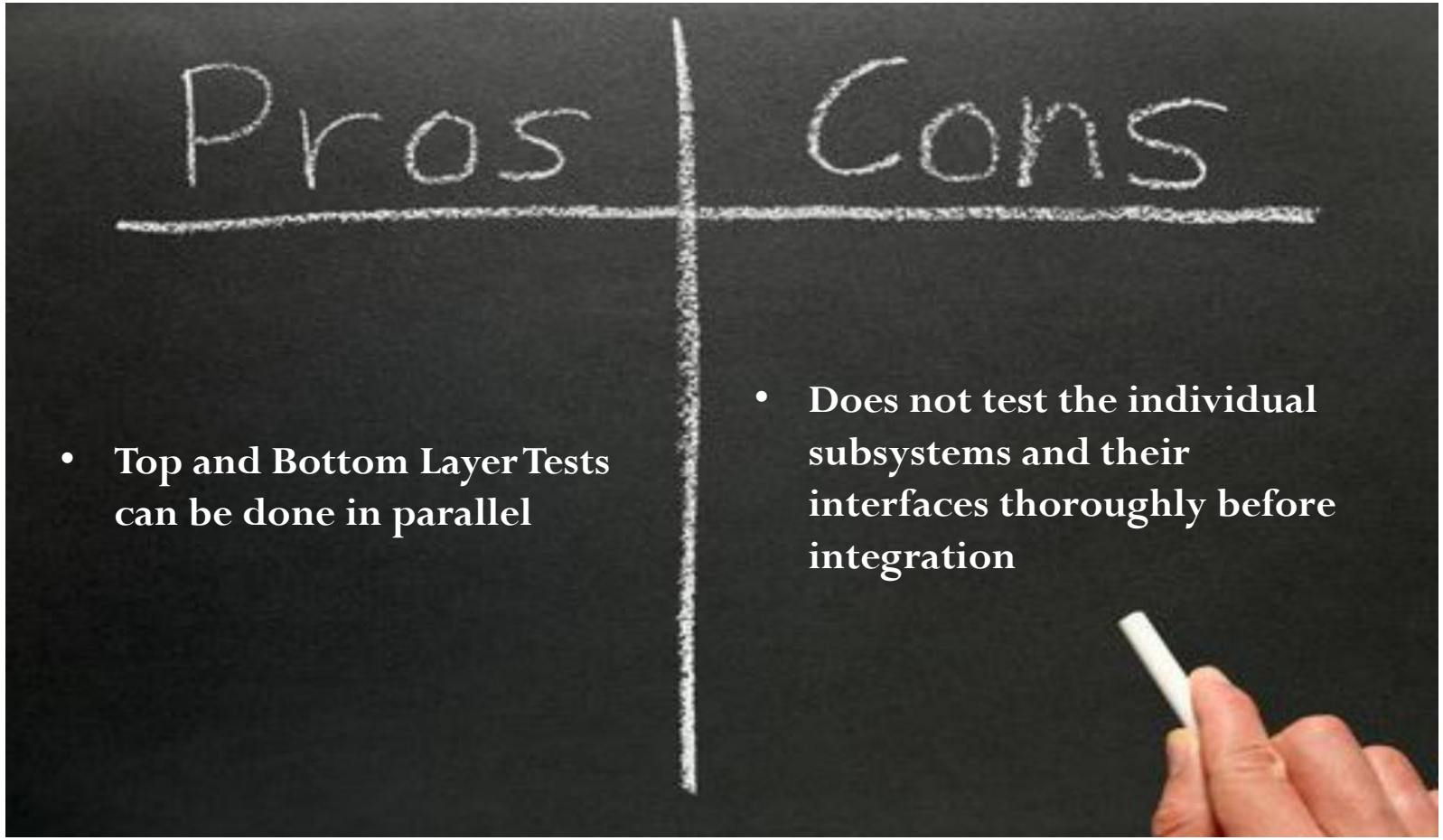
- How do you select the target layer if there are more than 3 layers?



Sandwich Testing



Sandwich Testing



SOLUTION: Modified Sandwich Testing

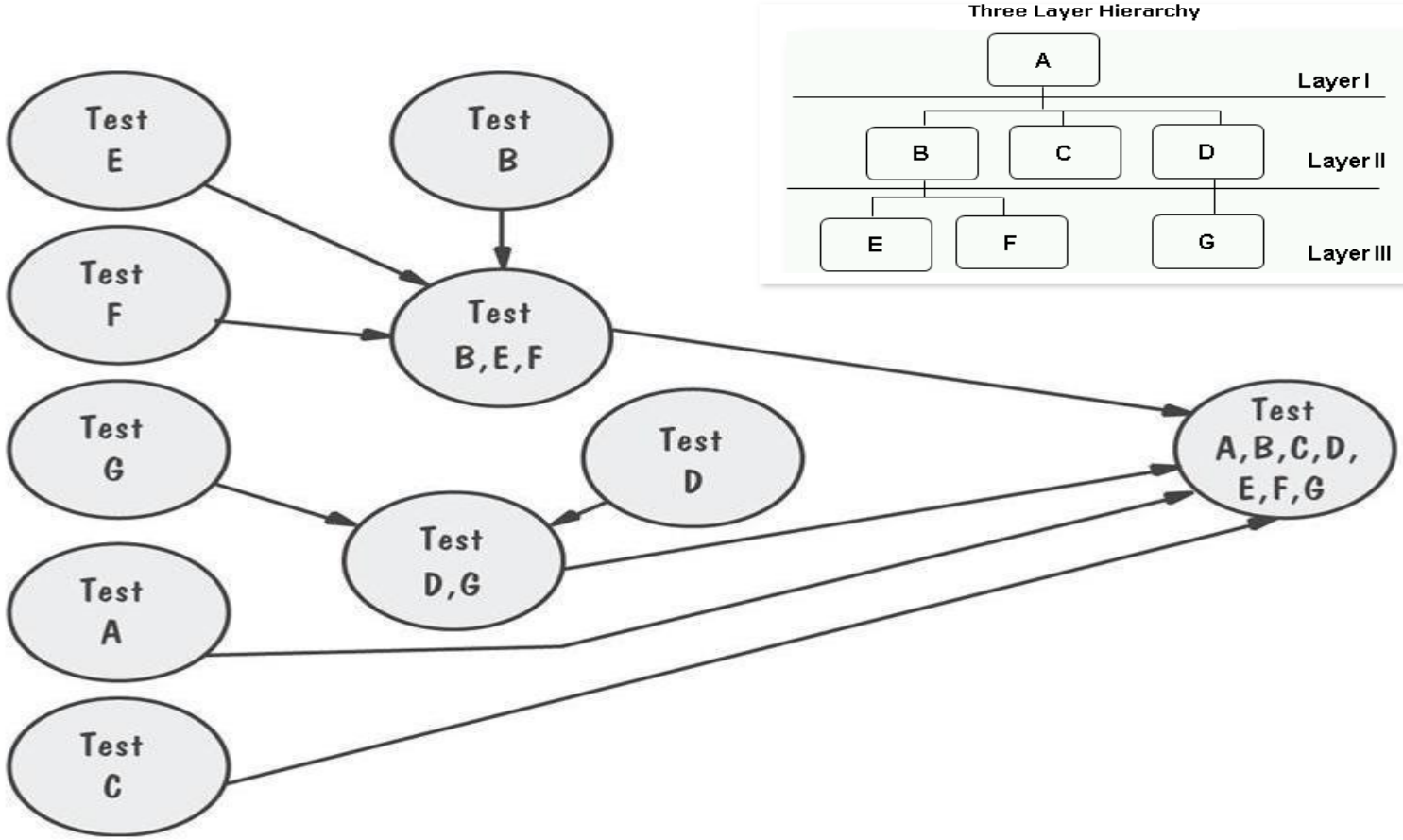
Modified Sandwich Testing

- Test in parallel:
  - Middle layer with drivers and stubs
  - Top layer with stubs
  - Bottom layer with drivers

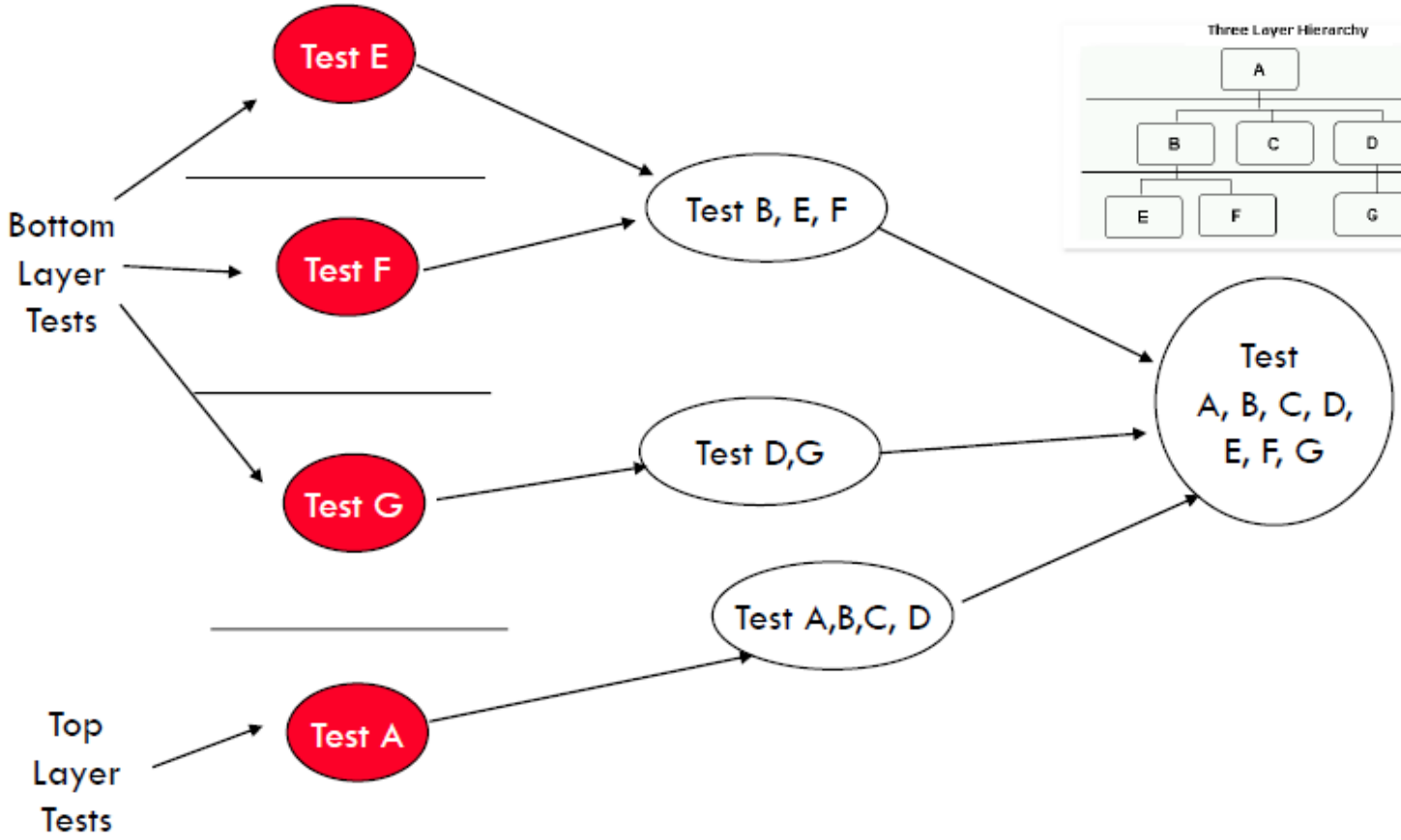
- Test in parallel:
  - Top layer accessing middle layer (top layer replaces drivers)
  - Bottom accessed by middle layer (bottom layer replaces stubs).

Modified Sandwich Testing

- Allows upper-level components to be tested before merging them with others



Sandwich Testing



Assignment

- Check out Reference No [14] at the end of lecture slides guru99.com site where around 105 testing types are mentioned. Study and understand those testing types. Submit a report on various Black box Testing Methods.

References

1. http://www.faqs.org/faqs/software-eng/testing-faq/section-14.html
2. http://en.wikipedia.org/wiki/White-box_testing
3. http://agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf
4. http://blogs.ebusinessware.com/2009/06/26/unit-testing-vs-module-testing/
5. http://www.softwaretestinghelp.com/black-box-testing/
6. Roger Pressman, Software Engineering, 7th Edition
7. http://www.techopedia.com/definition/7035/end-to-end-test
8. http://testing-a-software.blogspot.com/2011/11/common-issue-found-from-globalization.html
9. www.onestoptesting.com
10. http://www.guru99.com/end-to-end-testing.html
11. http://www.asi-test.com/ASI/system-vs-endtoend-testing/
12. http://www.guru99.com/types-of-software-testing.html