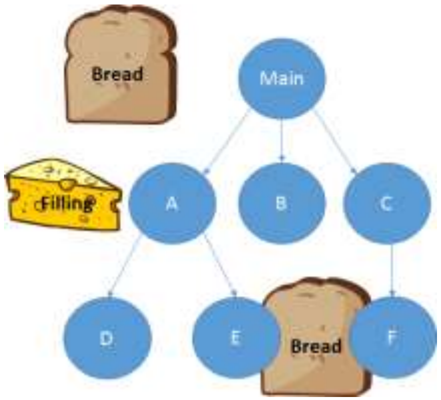
Software Engineering - 5CM505

Software Testing

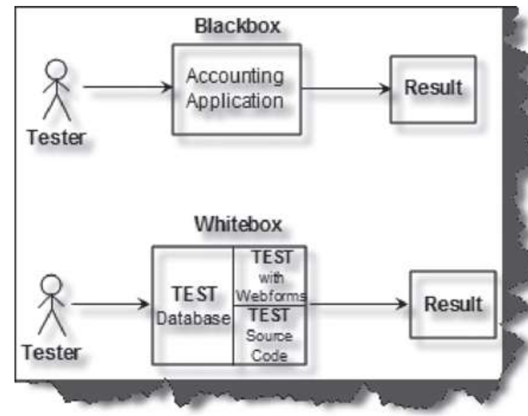
Ioannis Tsioulis (MC)

slides acknowledgements
Wajahat Ali Khan (UoD)

Content



- Introduction to Software Testing
- When to test?
- Validation and Verification
- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing
- Summary



Learning Outcomes

- By the end of this lecture you should be able to:
  - Explain why it is important to test software
  - Discuss the difficulties with testing software and bug fixing
  - Explain the differences between software validation and software verification
  - Discuss the different types of software tests (unit tests, integration tests, system tests, user acceptance tests)
  - Explain the differences between black box testing and white box testing
  - Explain the different integration testing strategies

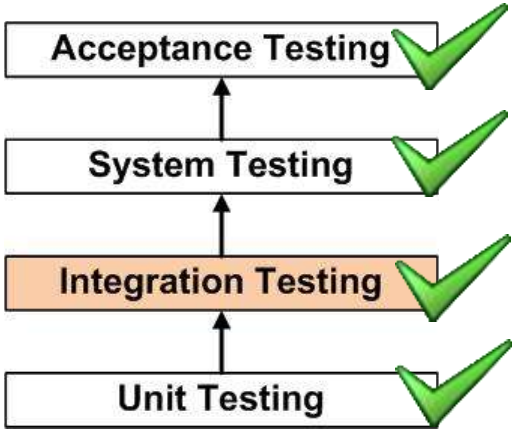
Introduction to Software Testing

Introduction

- Difficult to test your own code because …
- If you knew there was a bug you would have fixed it already!
- Programmers often assume their code is correct so they do not always test it as thoroughly as they should
- Even if a piece of code is thoroughly tested and contains no bugs there is no guarantee that it will work properly when integrated with other parts of the system
- To address these issues, different kinds of tests are performed.

Different Types of Test

- First developers test their own code
- Then others tests it
- If it seems to work properly it is integrated with other parts of the project to see if the new code “broke” anything
- When the complete system is assembled it is tested as a whole
- Finally, the system is tested to check that it meets the requirements
- Any time a test fails, programmers need to go back into the code to see what is wrong and how to fix it
- After changes are made:
  - The whole process starts again



Software Testing

- Why restart the whole process again?

Why Restart the Whole Process?

- Fixing a bug often creates a new bug
- Sometimes the bug fix is incorrect
- Sometimes the bug fix corrects some behaviour, but it breaks another part of the code because that part depended on the **original incorrect behaviour**
- Sometimes the bug fix changes some correct behaviour to a different correct behaviour, but it breaks another part of the code because that part depended on the **original correct behaviour**

Bugs

- You can never be certain that you've caught every bug
- If you run your tests and don't find anything wrong it doesn't mean there are no bugs, just that you haven't found them
- The best you can do is test and fix until no more bugs are encountered

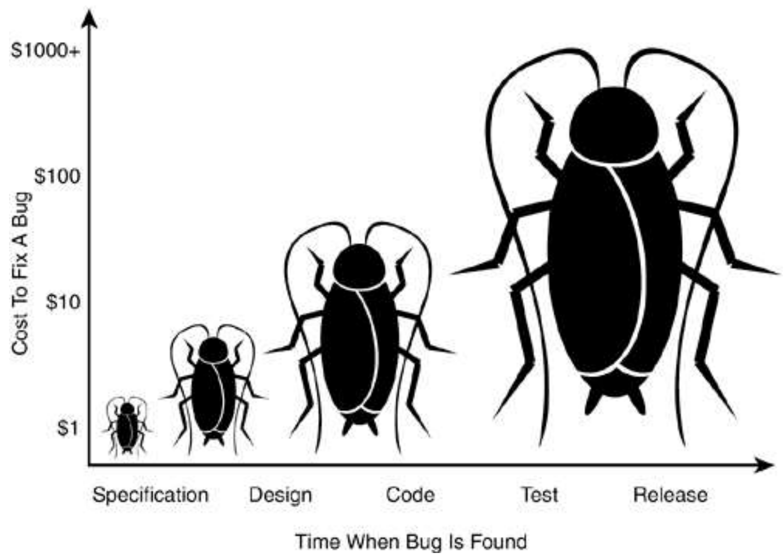
Testing shows the presence, not the absence of bugs

Edsger Dijkstra

When to Test?

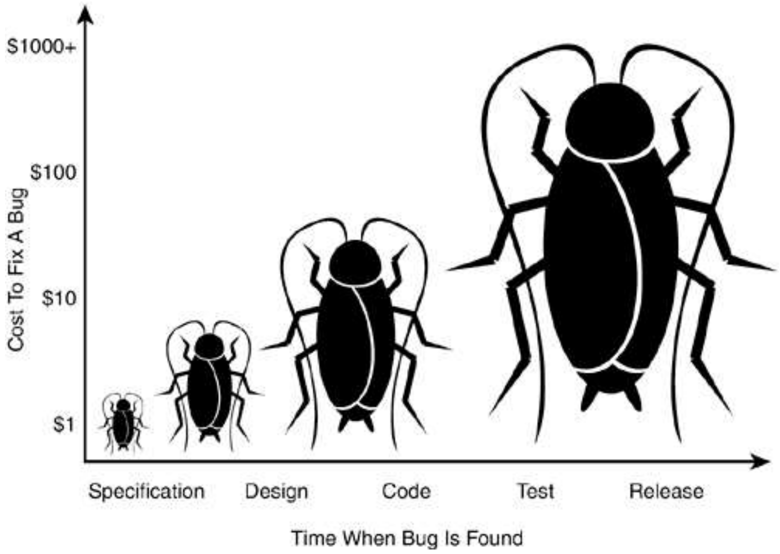
- When is the best time to test for bugs?

- Some people think of testing as something that is carried out on code after it is written, to verify that it is correct
- However, testing is critical at **every stage of development** to ensure that the resulting application is usable
- The earlier in the development process that an error is detected the better in terms of time and cost to fix it



When to Test?

- For example, if you detect an error during the requirements gathering phase, you need only fix that error.
- However, if the error goes unchecked, incorrect decisions could be made in the next step, leading to more errors
- In turn, decisions could be made based on those errors that produce even more errors in the next step etc.
- **<u>The longer a bug remains undetected, the harder it is to fix</u>**



Validation and Verification

- What is the difference?
- This question is often asked during job interviews for software development roles

Software Validation

- Validation is the process of checking whether the specification captures the customer's needs
- It can be defined as:

  "*The evaluation of software throughout the development process to ensure compliance with user requirements*"
  (Sanders and Curran, 1994)

- It is concerned with answering the question "**Am I building the correct product**?"
- Validation ensures that the product will be useful to the customer, i.e. that it has the functionality that the customer requested
- Does the product do what it is supposed to do? Does it meet the **operational and functional** requirements?

- Validation is largely a **subjective** process
- It involves making subjective assessments of how well the (proposed) system addresses a real-world need.

Software Verification

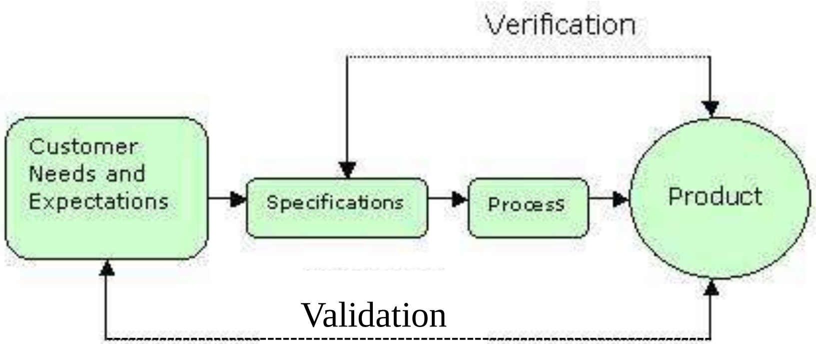
- Verification ensures the quality of a product
- It can be defined as:

  “*The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services or documents conform to specified requirements*”

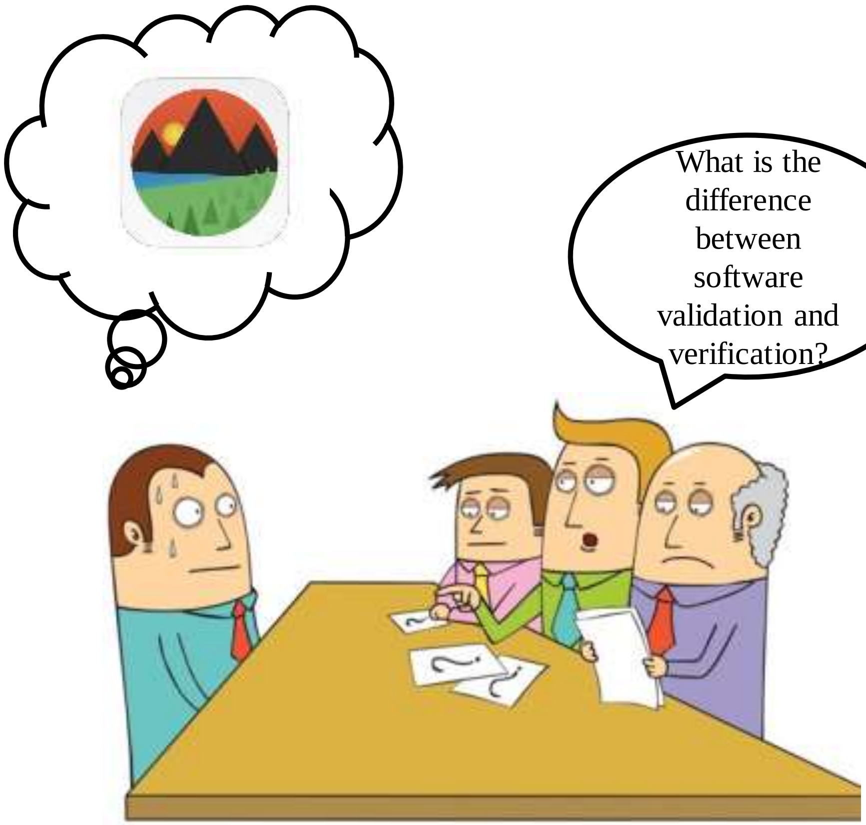
  (IEEE 729 standard)

- It is concerned with answering the question “**Am I building the product correctly**?”
- Verification ensures that the product specifications are met
- Largely concerned with the **non-functional requirements** (performance, system, implementation)

○ Verification is a relatively **objective** process, in that if the specifications are expressed precisely enough, no subjective judgements should be needed in order to verify software



Remembering the Difference



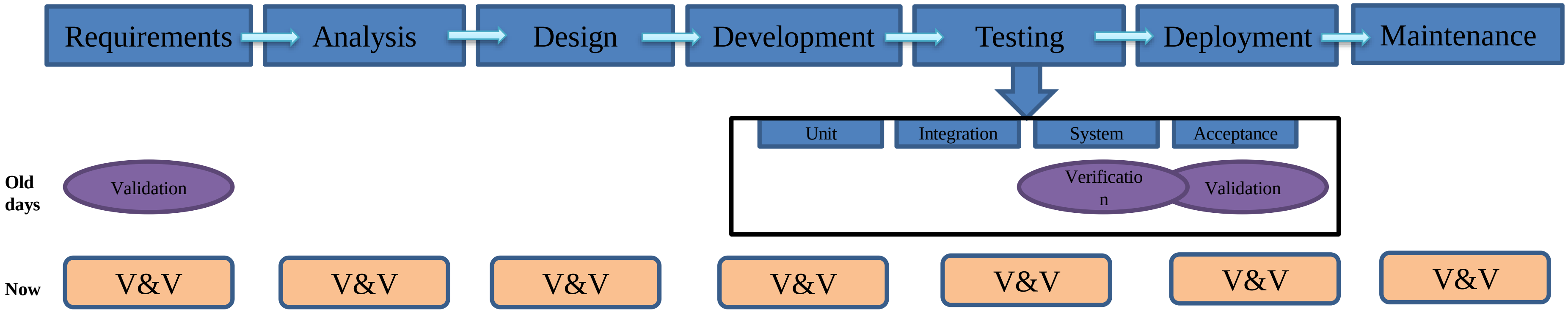
- Validate the solution - VALSOL
- Verify the quality - VERQUAL

Verification and Validation

- Validation is often relegated to the beginning and ending of the project: requirements analysis and acceptance testing.
- This view is common in many software engineering textbooks, but is misguided… why?
- It assumes that the customer’s requirements can be captured completely at the start of a project, and that those requirements will not change while the software is being developed.

Verification and Validation

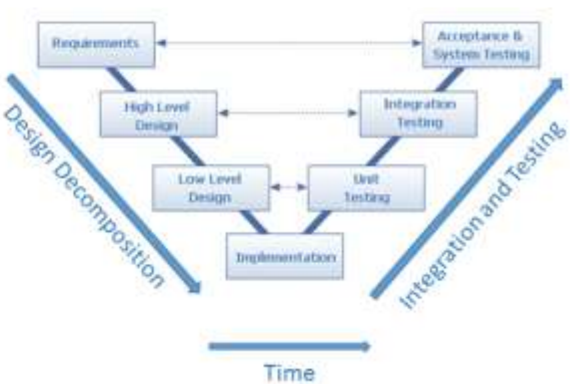
- In practice, the requirements change throughout a project, partly in reaction to the project itself: the development of new software makes new things possible. Therefore both validation and verification are needed throughout the lifecycle.
- Finally, V&V is now regarded as a coherent discipline.

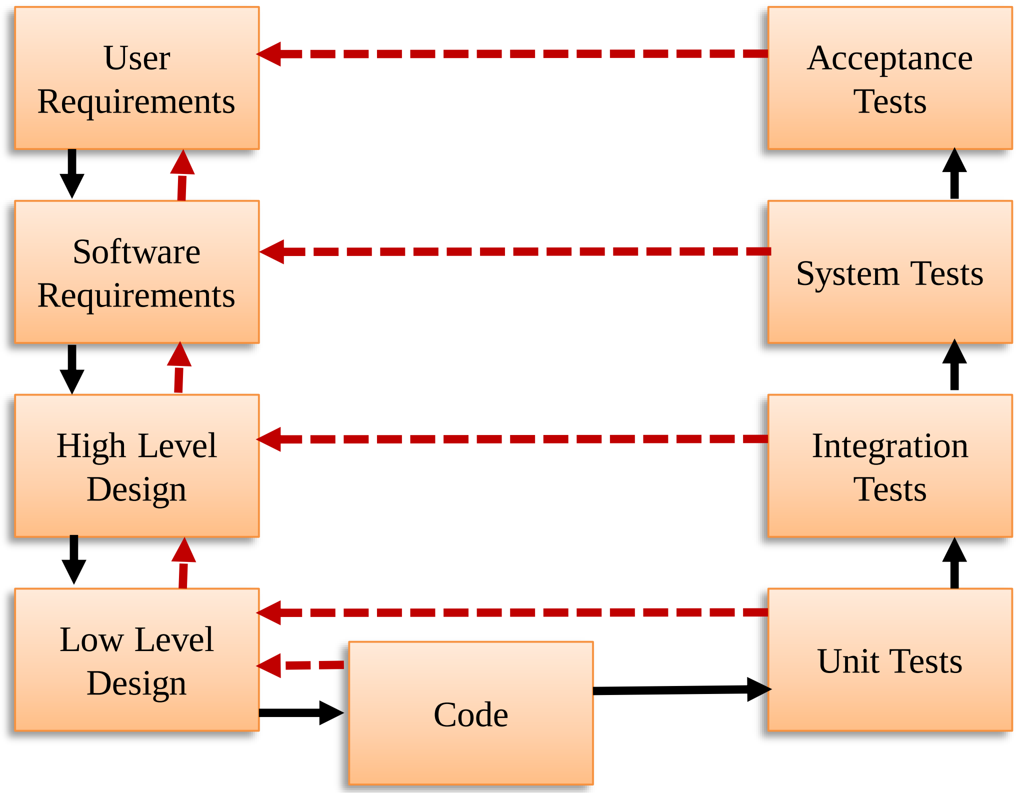


Sequence

Verify / Validate

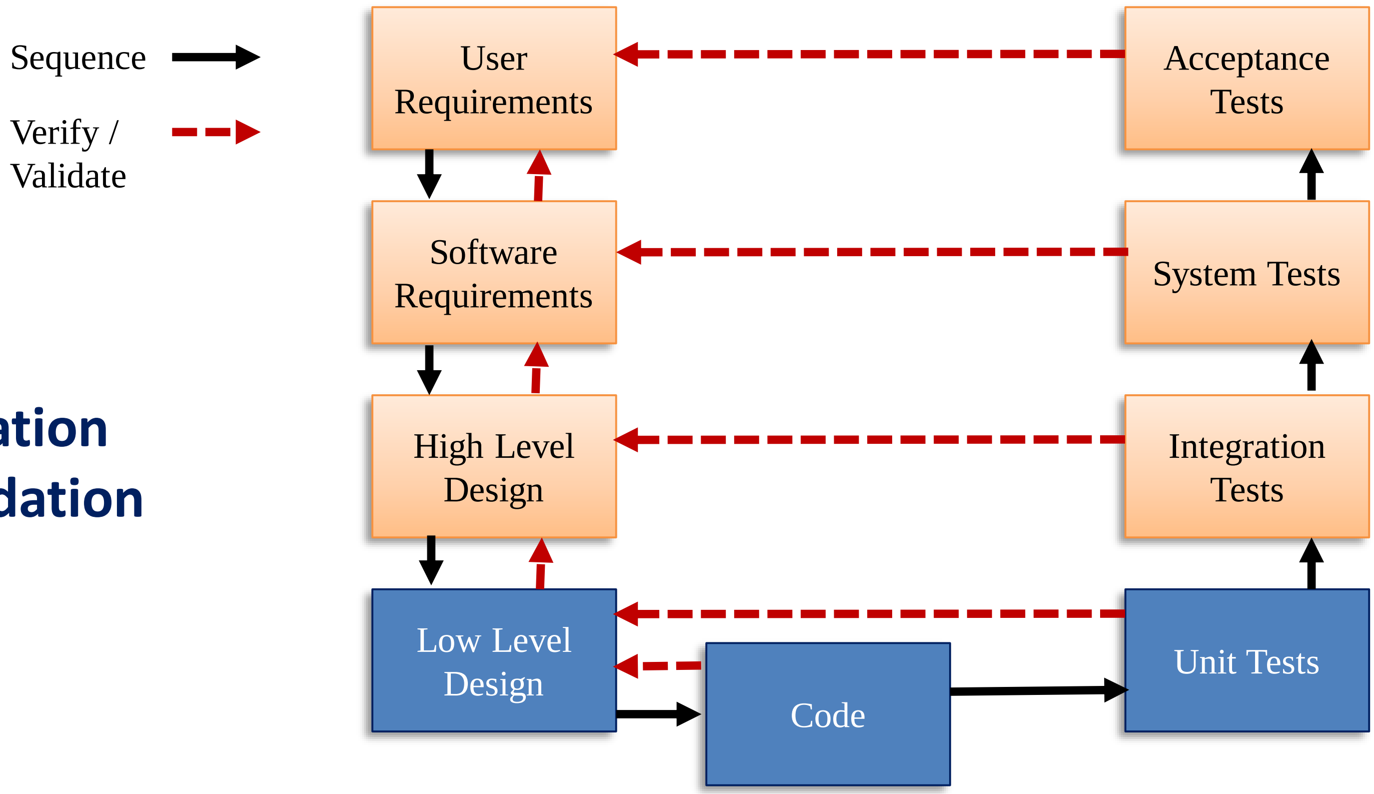
Remember the V Model?





Unit Testing

Verification and Validation

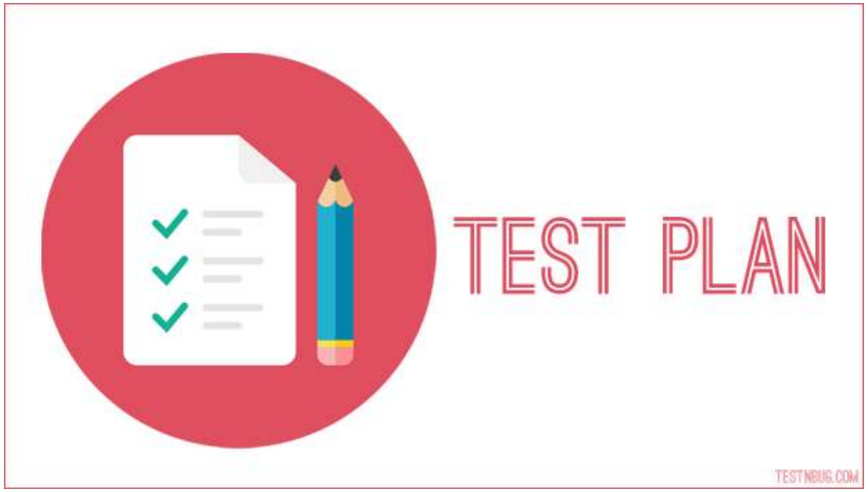


Unit Testing

- Definition – “A test that confirms the correctness of a specific piece of code” (Stephens, 2015)
- As soon as you finish writing a piece of code you should test it. Test as thoroughly as possible because it will get harder to fix later
- Usually, unit tests are applied at the **method** level but sometimes they are broken down into **parts of methods**.
- Unit tests represent the first opportunity to catch bugs so they are very important

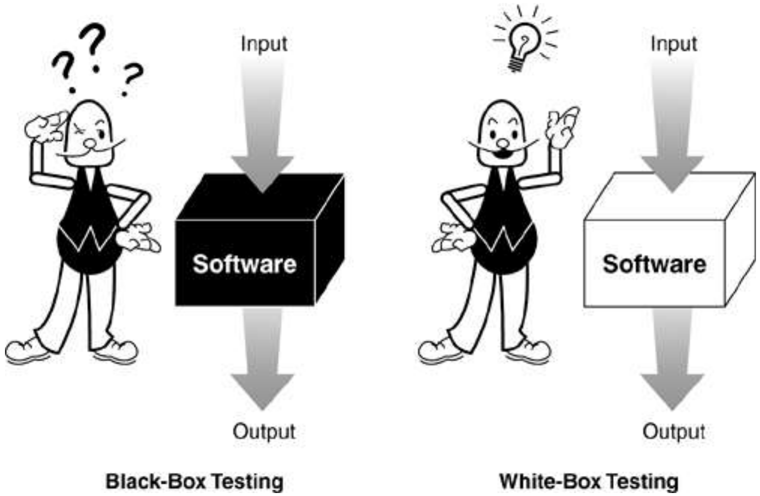
Unit Testing

- It is sometimes a good idea to create the test structure before actually writing the method.
- This way you do not know what assumptions the code makes, so you cannot make the same assumptions when you write the tests



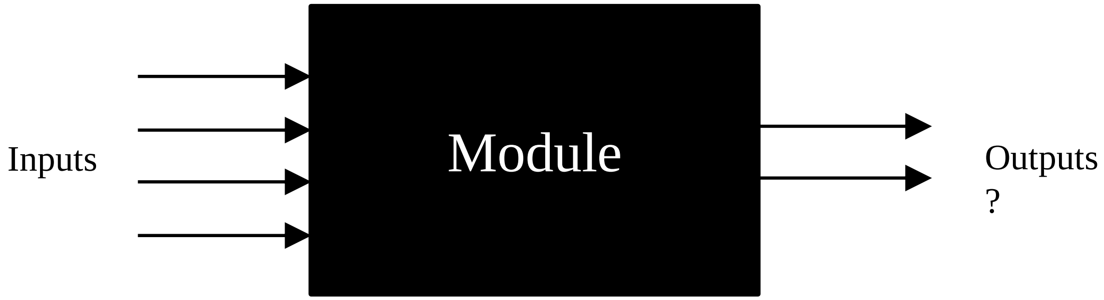
Black Box versus White Box Testing

- What is the difference?



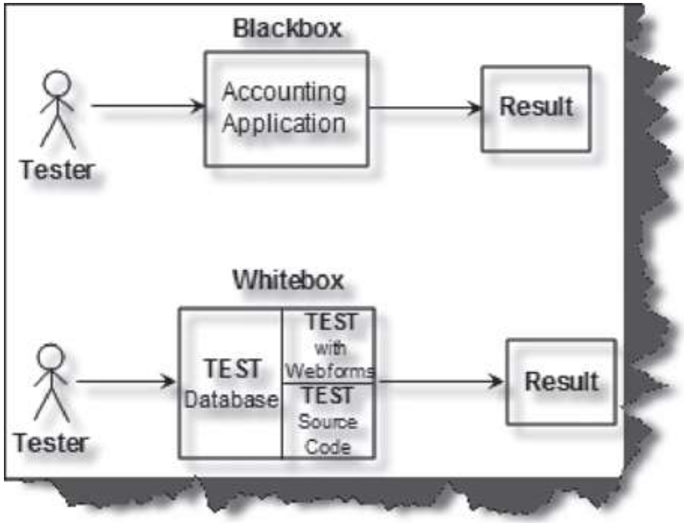
Black Box Testing

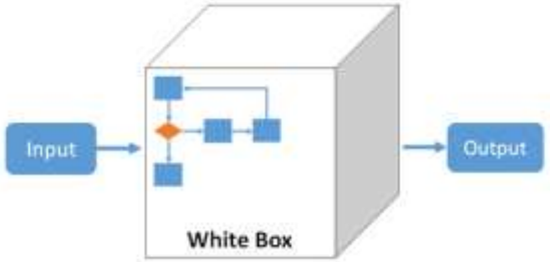
- Black Box Testing – Checks that for a given set of inputs the module produces the correct set of outputs. It **does not** check the method used to produce the results



White Box Testing

- White Box Testing – Checks that the correct paths through the code are executed under given circumstances.
- Should check that every statement in the module is executed at least once, and that all possible paths through the code are identified and tested.
- It is important to carry out **both** black box and white box testing on each module of the code to test whether it is performing as it should.

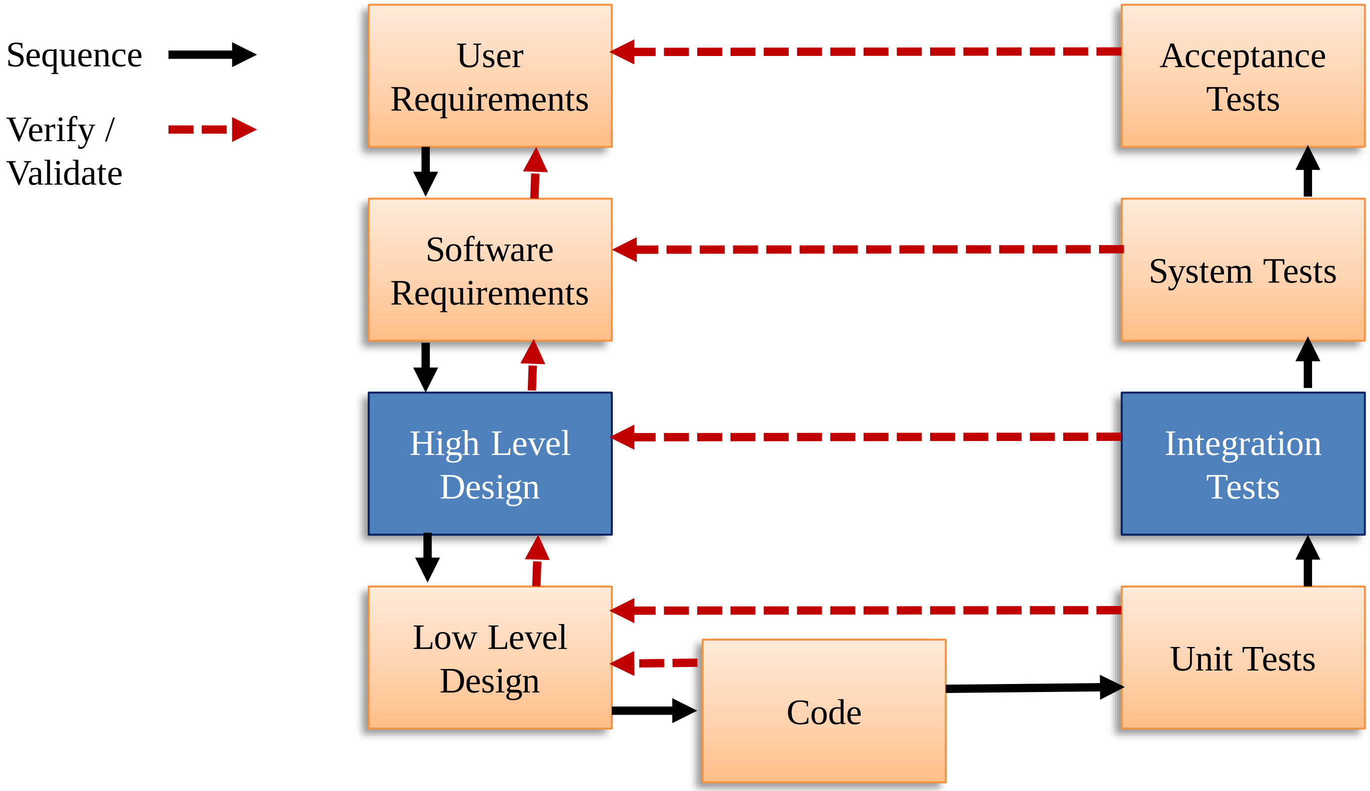




Integration Testing

Emergent Behaviour in Software

- Even if our sub systems have been tested and work well in isolation, we cannot necessarily predict what will happen when they are integrated together
- Maybe there are simple incompatibility issues
- Maybe there are errors that we cannot uncover until we try to integrate them
- Maybe unwanted emergent behaviour will prevent us obtaining the result we hoped for
- Whatever the case, we need to test the subsystems in combination with each other

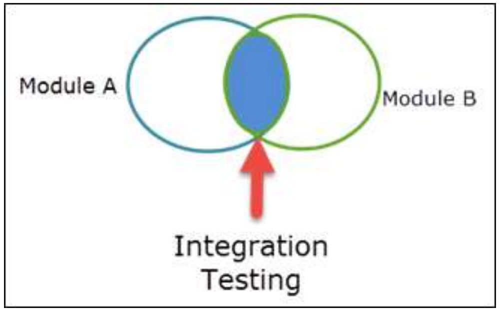

Sequence
Verify /
Validate
User
Requirements
Acceptance
Tests
Software
Requirements
System Tests
High Level
Design
Integration
Tests
Low Level
Design
Code
Unit Tests

Famous Failure

- In September 1999, the Mars Climate Orbiter failed after successfully travelling 416 million miles in 41 weeks. It disappeared just as it was to begin orbiting Mars
- Lockheed Martin Astronautics worked in pounds (Imperial measurement) when calculating acceleration data, whilst NASA's Jet Propulsion Laboratory worked in Newtons (metric).
- NASA spent a further $50,000 investigating how the fault occurred
- The fault could have been avoided by using integration testing

Integration Testing

- Integration testing is the phase in software testing in which individual software modules are combined and tested as a group.
- It takes modules that have been unit tested as input, groups them into larger aggregates, and applies tests (defined in an integration test plan), both black box and white box.

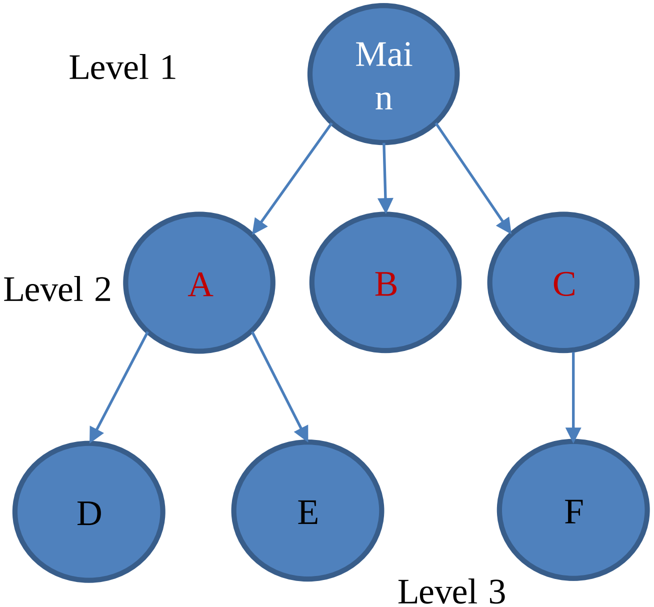


Integration Strategies

- Four approaches to integration
  - Big bang
  - Top-down
  - Bottom-up
  - Sandwich
- Each assumes that the individual units have been separately tested
- Last three describe the order in which units of code are to be integrated

Big Bang

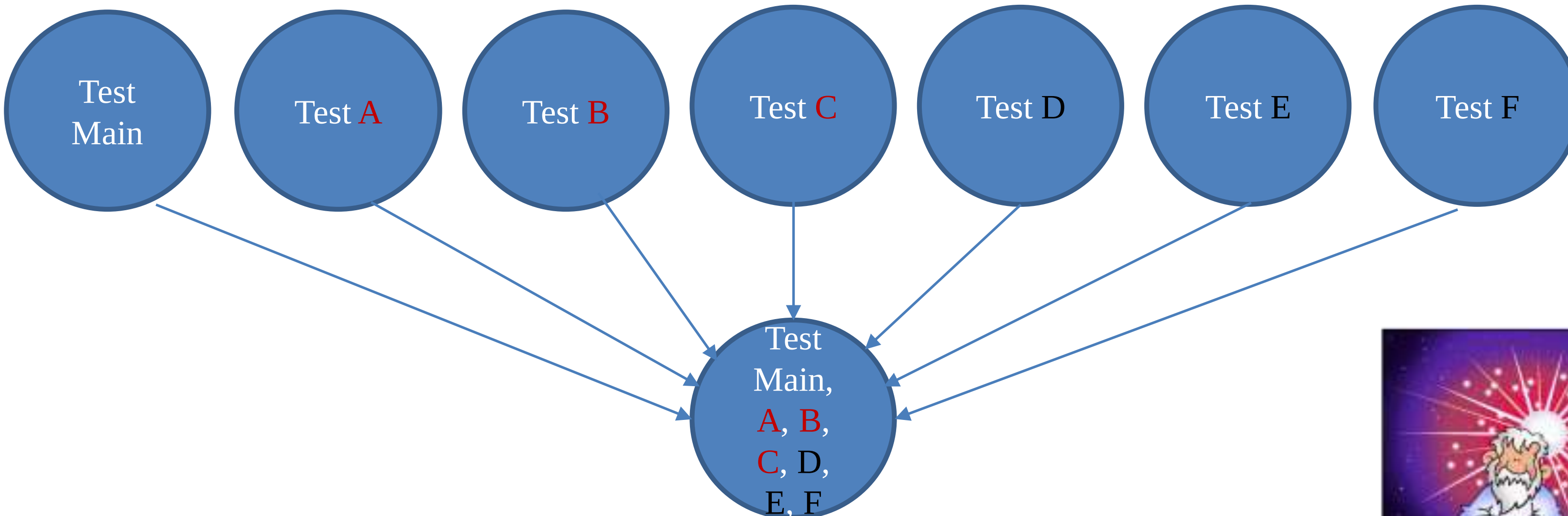
- Test all units separately and then integrate them all at once
- The disadvantage is that …
  - It is difficult to isolate faults
- The advantages are
  - We do not need to create stubs or drivers (see later slides)
  - Less tests need to be carried out



Example abstract representation of code

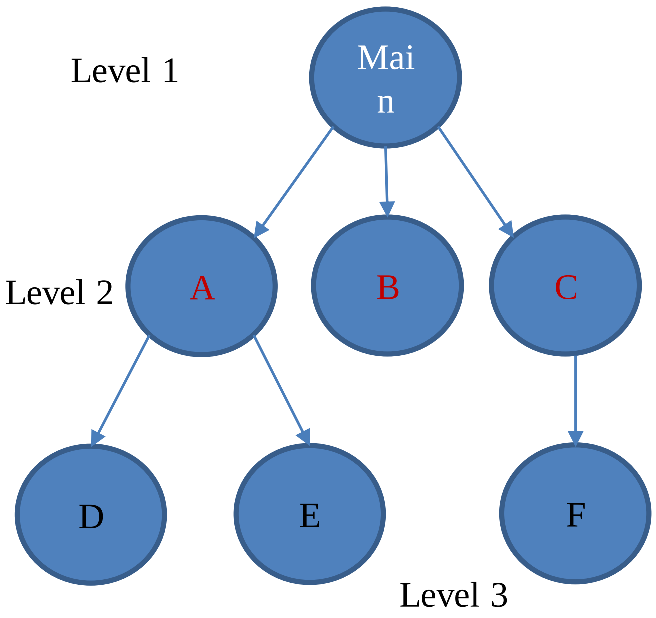
A call to a lower level method or function

Big Bang

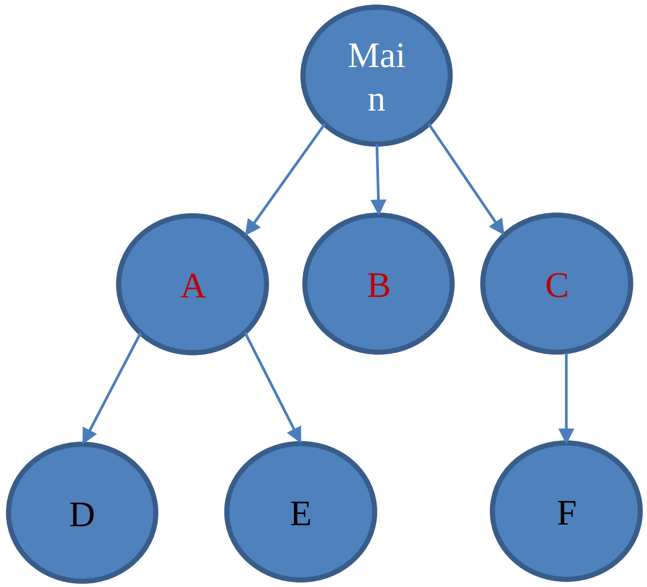

Test Main
Test A
Test B
Test C
Test D
Test E
Test F
Test Main, A, B, C, D, E, F

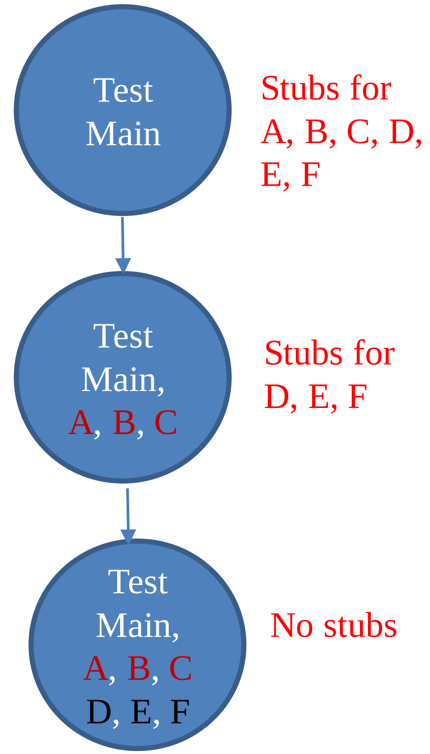
Top Down

- Begins with testing of the Main program
- At this point any lower level unit called by the Main program is replaced by a “stub”
- A stub is a piece of code that emulates a particular unit
- When we are convinced that the Main program logic is correct we gradually replace stubs with the actual code, level by level
- Two approaches to Top Down - replace all stubs on each level at once or replace individually

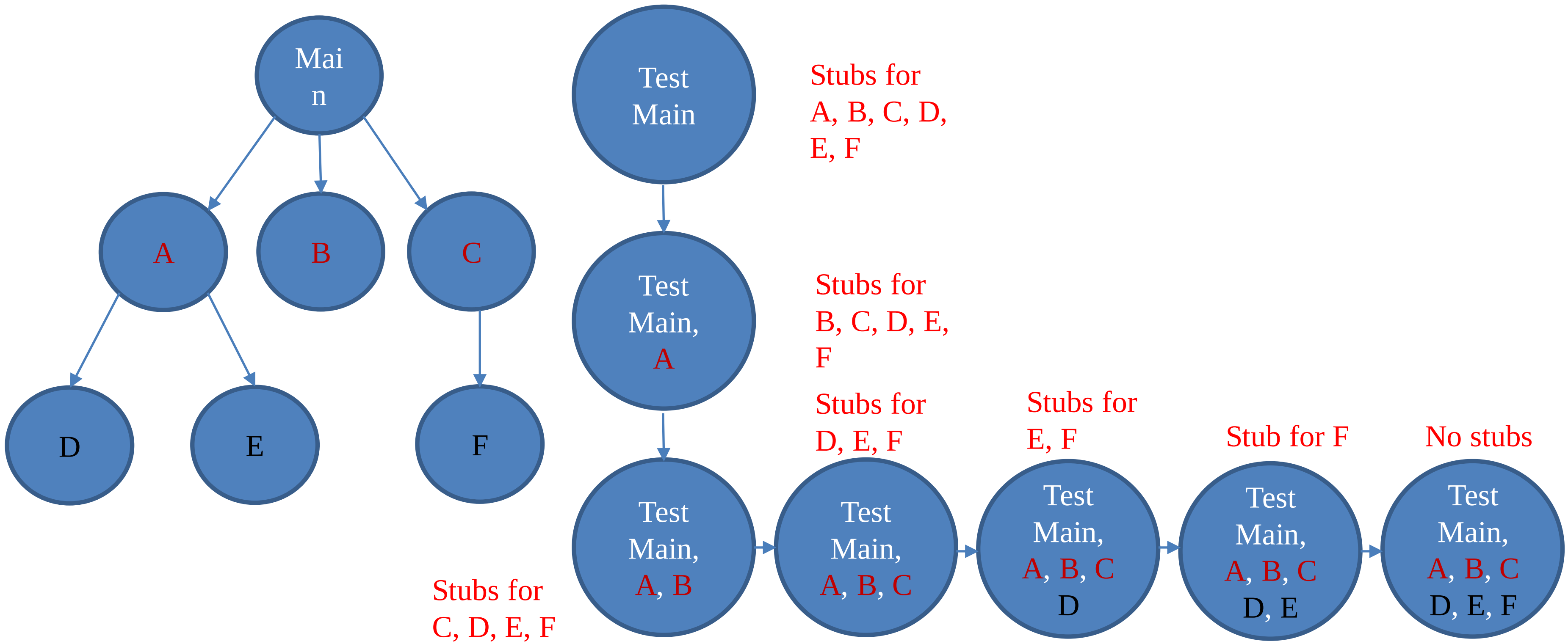


Top Down – Replace All Same-level Stubs at Once





Top Down – Replace One Stub at a Time

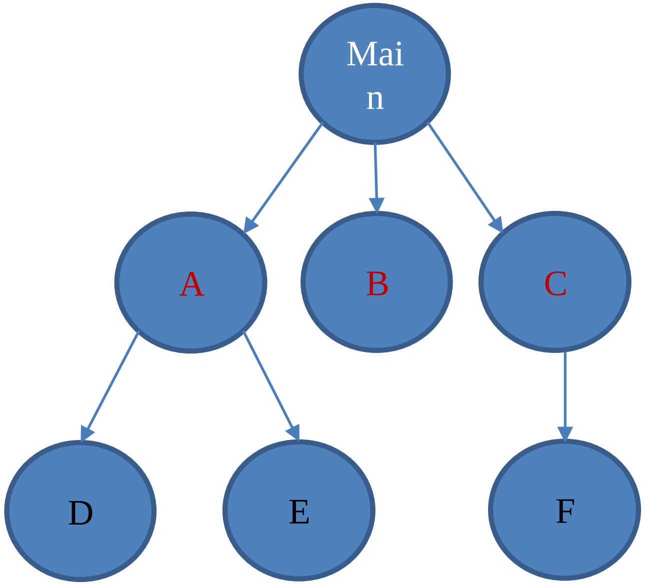


Top Down Comparison

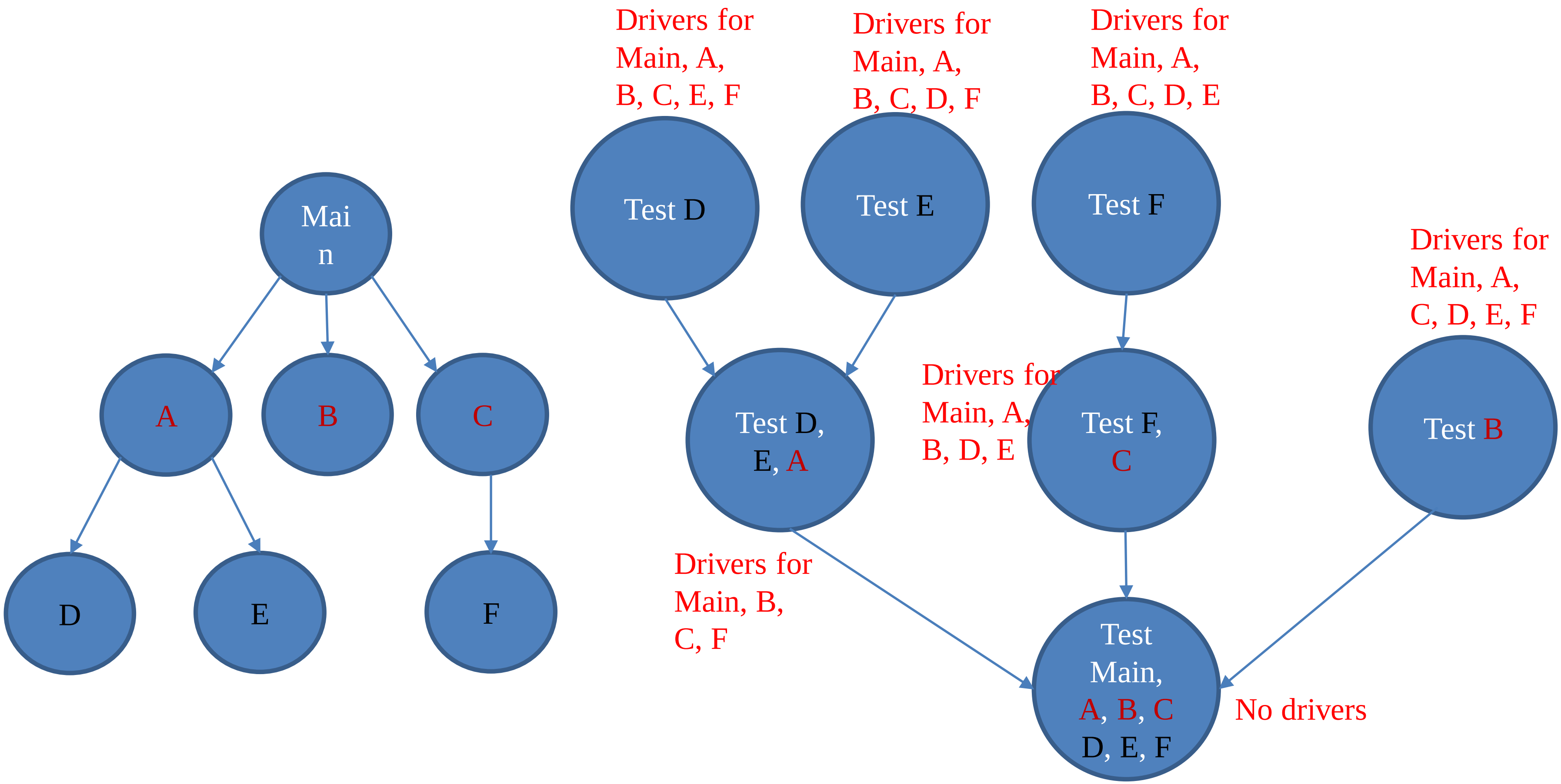
All same level stubs at once	One stub at a time
In the example, we would test the Main program three times • Disadvantage – If a fault is found how do we know which sub-program is causing it? • Advantage – Less tests so it could be faster	In the example, we would test the Main program seven times • Advantage – Easier to deduce which sub-program is causing the fault • Disadvantage – Testing more times, so it might take longer

Bottom Up

- Mirror image of the top-down order
- The difference is that stubs are replaced by driver modules that emulate the units at the next level up
- Start with the leaves of the decomposition tree and test with appropriate drivers
- Drivers are more complicated to construct than stubs



Bottom Up



Main
A
B
C
D
E
F
Drivers for Main, A, B, C, E, F
Test D
Drivers for Main, A, B, C, D, F
Test E
Drivers for Main, A, B, C, D, E
Test F
Test D, E, A
Drivers for Main, A, B, D, E
Test F, C
Drivers for Main, A, C, D, E, F
Test B
Drivers for Main, B, C, F
Test Main, A, B, C D, E, F
No drivers

Bottom Up / Top Down Comparison

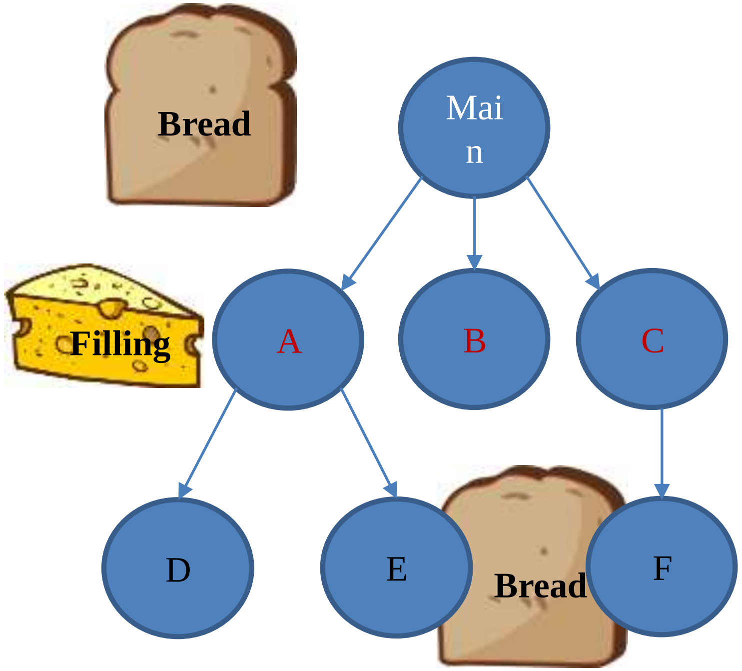
Bottom up best when:
- Low level modules are often invoked by other modules, so it is more useful to test them first
- Bottom up design methodology is used
- Major flaws occur towards the bottom of the program.

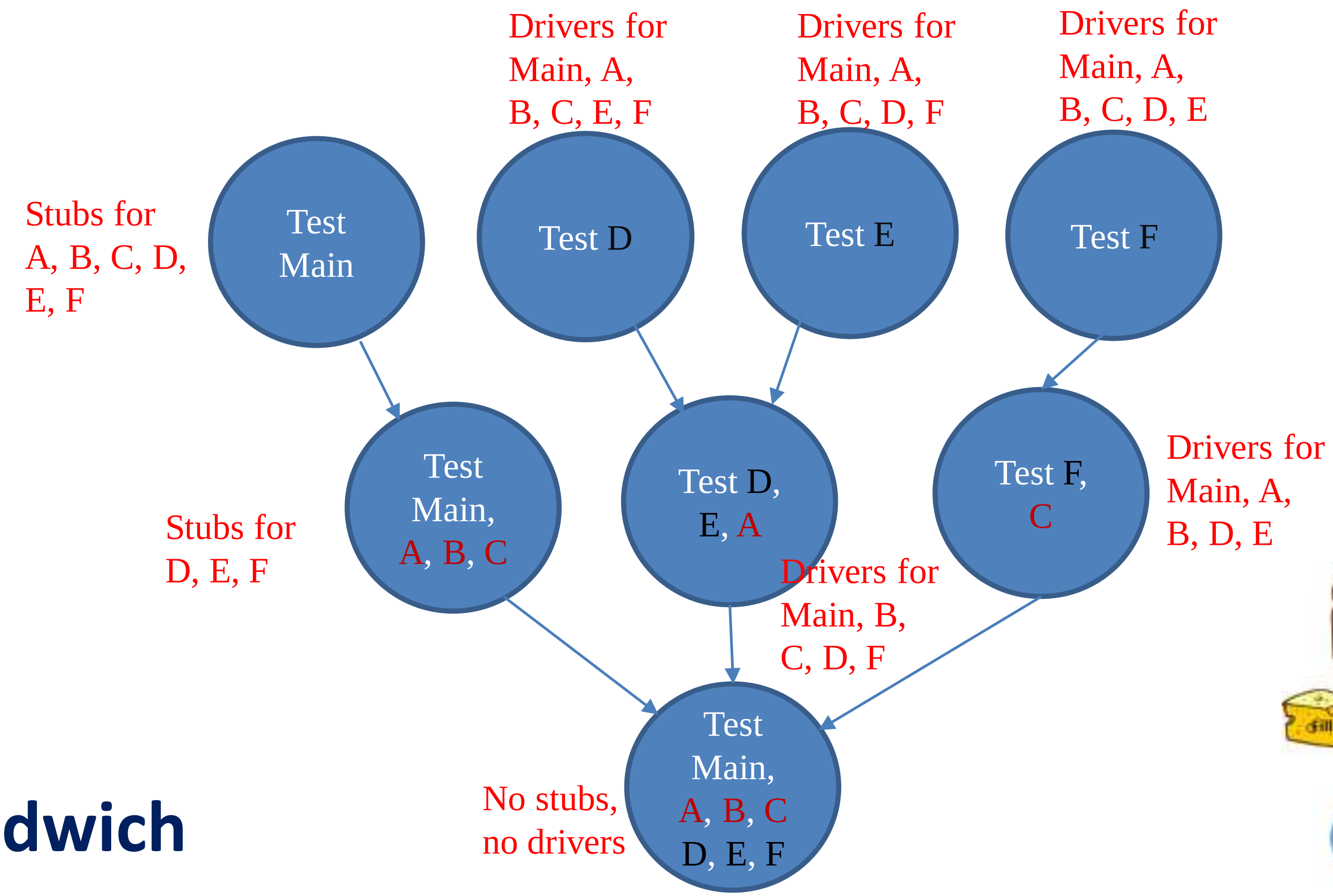
Top down best when:
- Top down design methodology is used
- Major flaws occur toward the top of the program.

- With bottom up design, critical modules are generally built and tested first and therefore any errors or mistakes in these forms of modules are identified early in the process.
- With top down testing, stub design becomes increasingly difficult when stubs lie far away from the top level module.

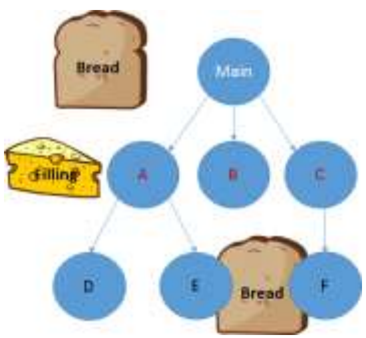
Sandwich

- Combination of top-down and bottom-up integration to try to get the best of both approaches
- Top and bottom levels are the "bread"
- Intermediate levels are the filling
- Test the "bread" first and then add the "filling"
- Less stub and driver development effort but offset by difficulty of fault isolation that comes with integrating too many modules at once




Stubs for
A, B, C, D,
E, F
Test
Main
Drivers for
Main, A,
B, C, E, F
Test D
Drivers for
Main, A,
B, C, D, F
Test E
Drivers for
Main, A,
B, C, D, E
Test F
Stubs for
D, E, F
Test
Main,
A, B, C
Test D,
E, A
Drivers for
Main, B,
C, D, F
Test F,
C
Drivers for
Main, A,
B, D, E
No stubs,
no drivers
Test
Main,
A, B, C
D, E, F

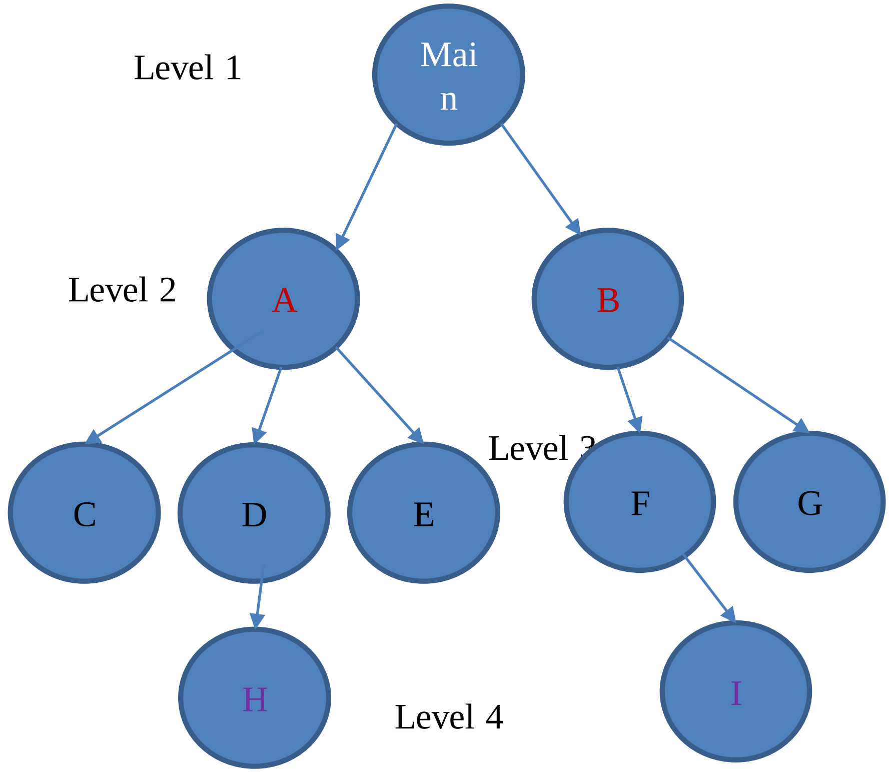
Sandwich


Bread
Main
Filling
Bread

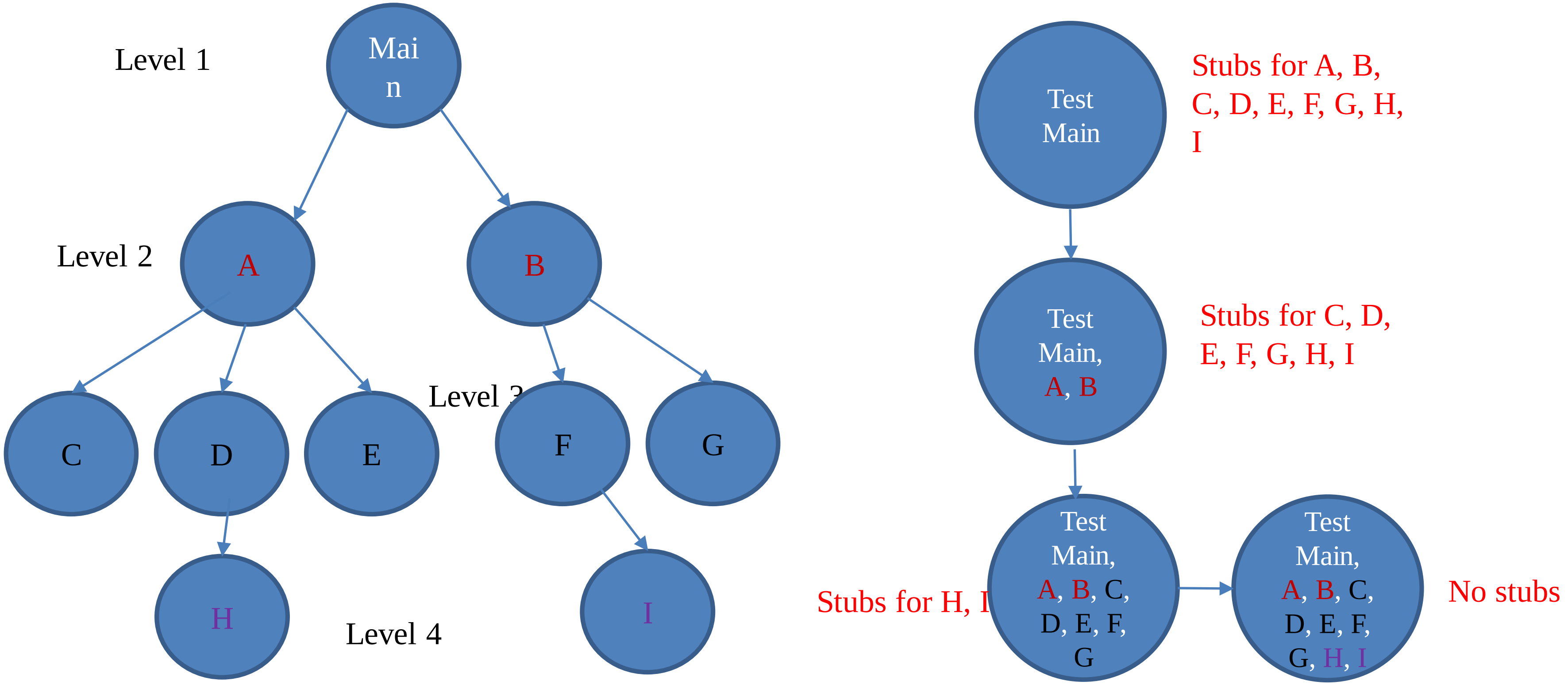
Group Exercise

- Draw the testing schedule for the following program structure for each integration strategy
  - Top down all same-level stubs at once
  - Top down one stub at a time
  - Bottom up
  - Sandwich

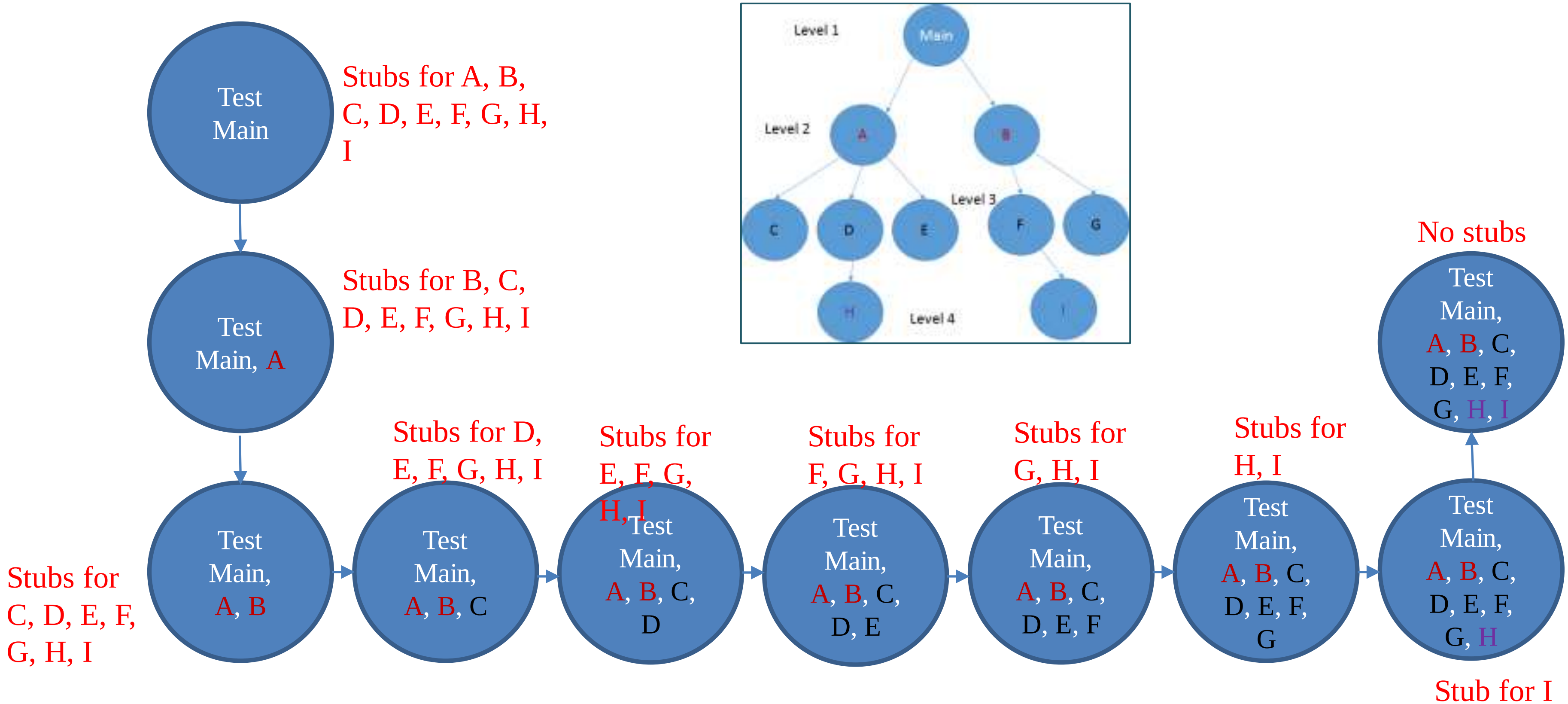
(10 mins)



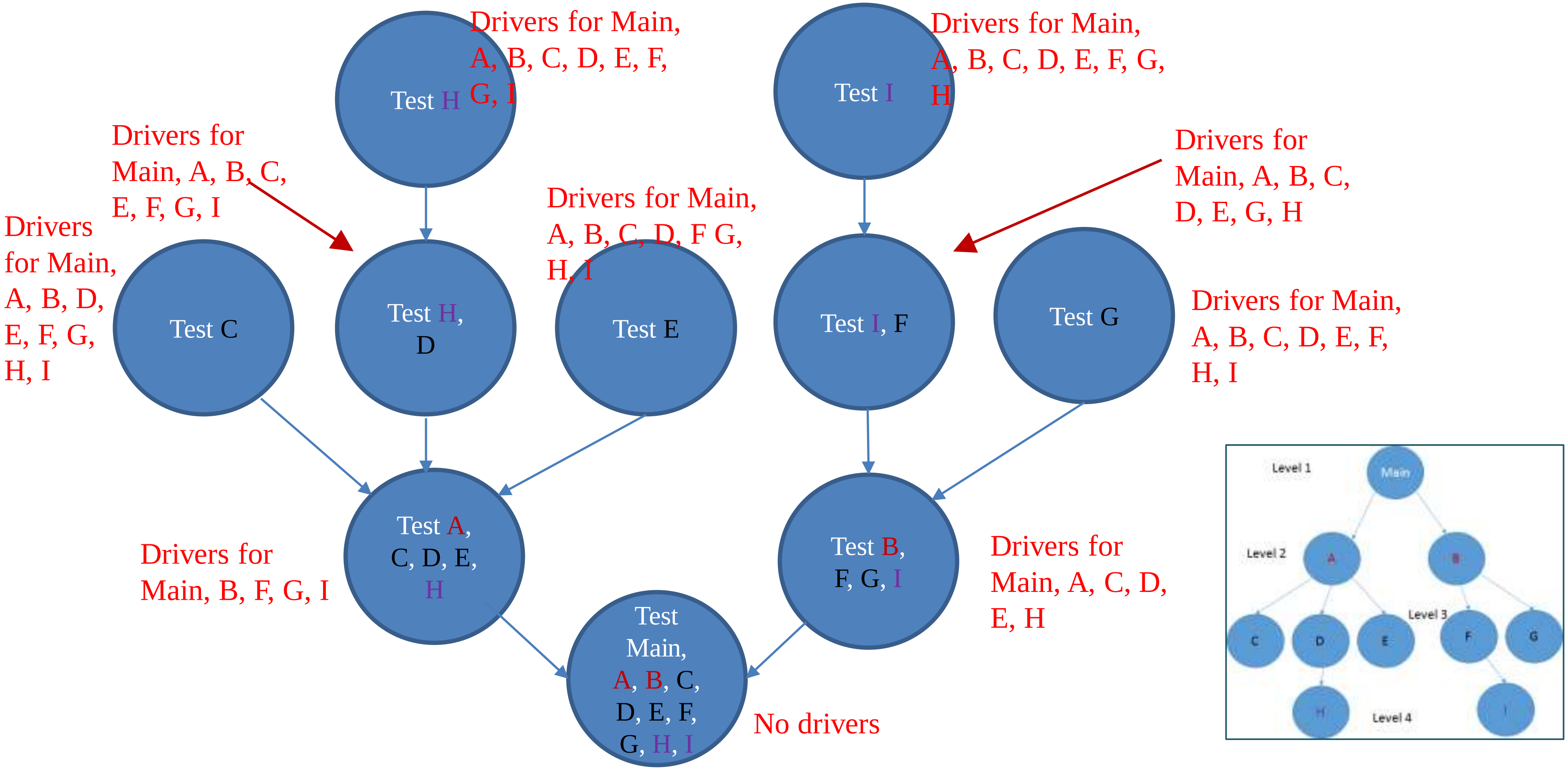
Top Down – Replace All Same-level Stubs at Once



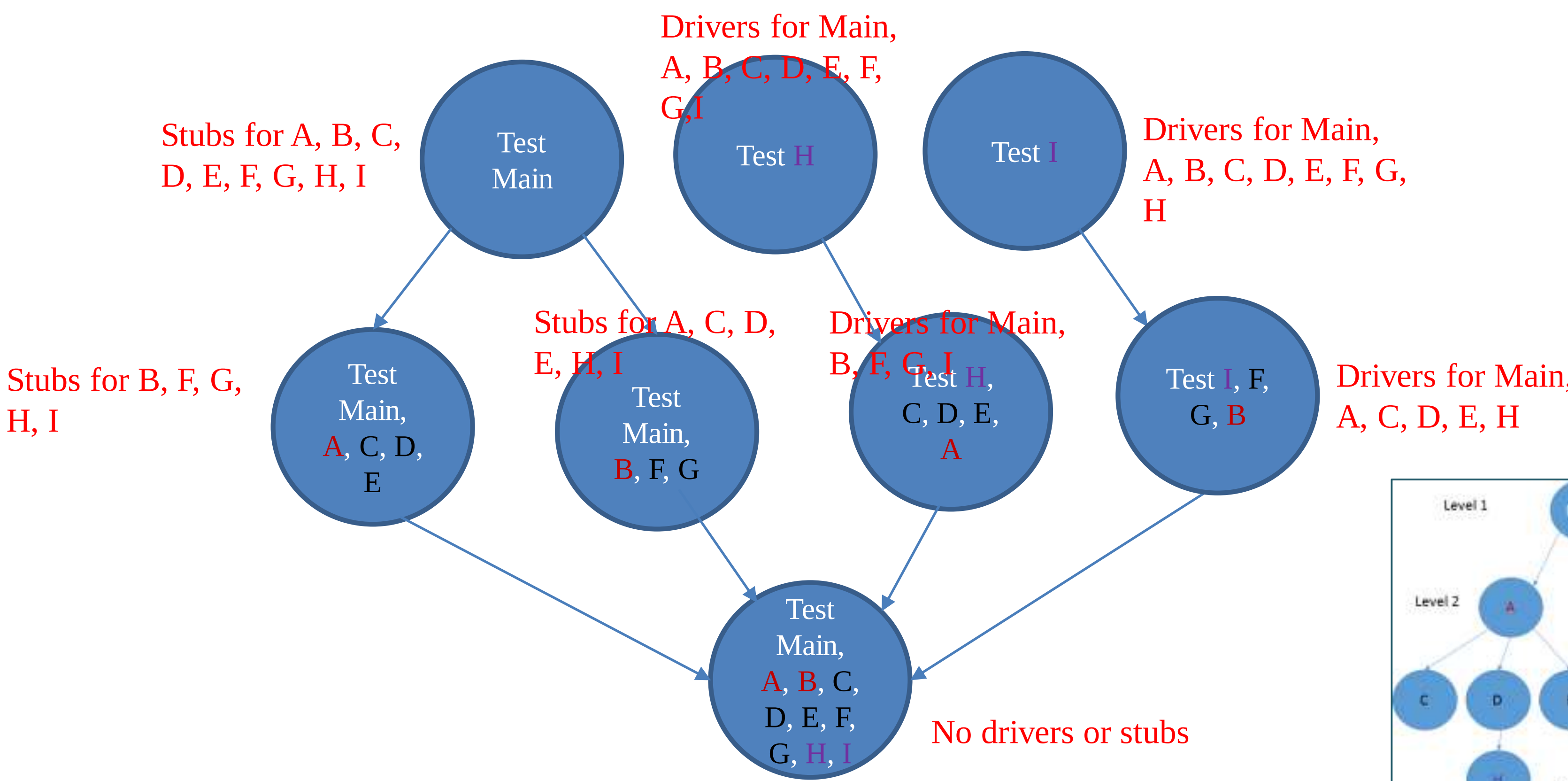
Top Down – Replace One Stub at a Time



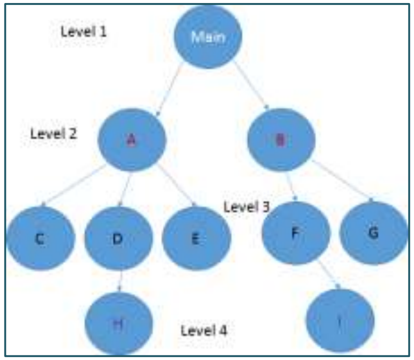
Bottom Up


Test H
Drivers for Main, A, B, C, D, E, F, G, I
Test I
Drivers for Main, A, B, C, D, E, F, G, H
Drivers for Main, A, B, C, E, F, G, I
Drivers for Main, A, B, C, D, F G, H, I
Drivers for Main, A, B, C, D, E, G, H
Drivers for Main, A, B, D, E, F, G, H, I
Test C
Test H, D
Test E
Test I, F
Test G
Drivers for Main, A, B, C, D, E, F, H, I
Test A, C, D, E, H
Drivers for Main, B, F, G, I
Test B, F, G, I
Drivers for Main, A, C, D, E, H
Test Main, A, B, C, D, E, F, G, H, I
No drivers
Level 1
Main
Level 2
A
B
Level 3
C
D
E
F
G
H
Level 4
I

Sandwich

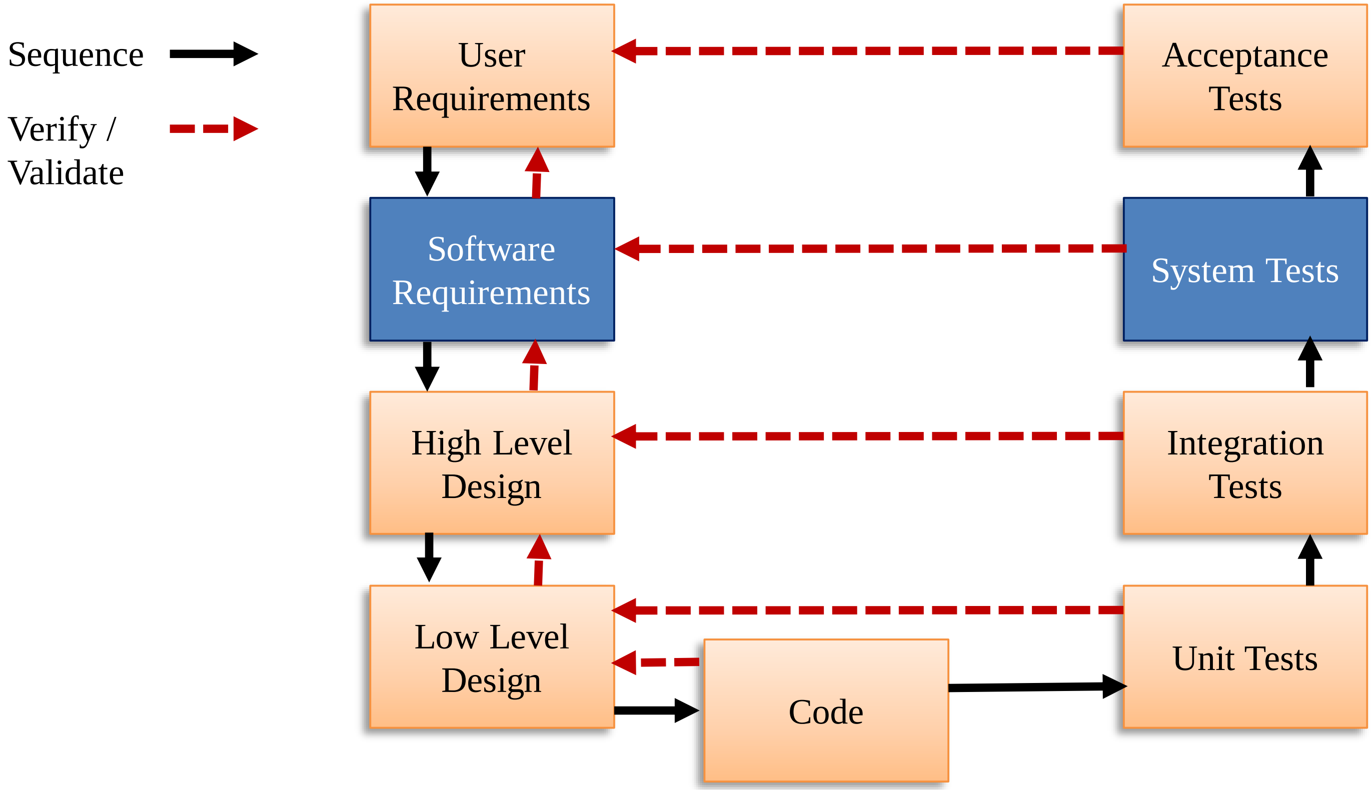


Stubs for A, B, C, D, E, F, G, H, I
Test Main
Drivers for Main, A, B, C, D, E, F, G,I
Test H
Test I
Drivers for Main, A, B, C, D, E, F, G, H
Stubs for B, F, G, H, I
Test Main, A, C, D, E
Stubs for A, C, D, E, H, I
Test Main, B, F, G
Drivers for Main, B, F, G, I
Test H, C, D, E, A
Test I, F, G, B
Drivers for Main, A, C, D, E, H
Test Main, A, B, C, D, E, F, G, H, I
No drivers or stubs



Level 1
Main
Level 2
A
B
Level 3
C
D
E
F
G
H
Level 4
I

System Testing


Sequence
Verify /
Validate
User
Requirements
Acceptance
Tests
Software
Requirements
System Tests
High Level
Design
Integration
Tests
Low Level
Design
Unit Tests
Code

System Testing

- System testing is conducted on a complete, integrated system to evaluate its compliance with the specified software requirements.
- It falls within the scope of black-box testing, and, as such, should require no knowledge of the inner design of the code or logic.
- It tests the software system itself (the "integrated" software components that have passed integration testing) integrated with any applicable hardware system(s).

Integration Testing and System Testing

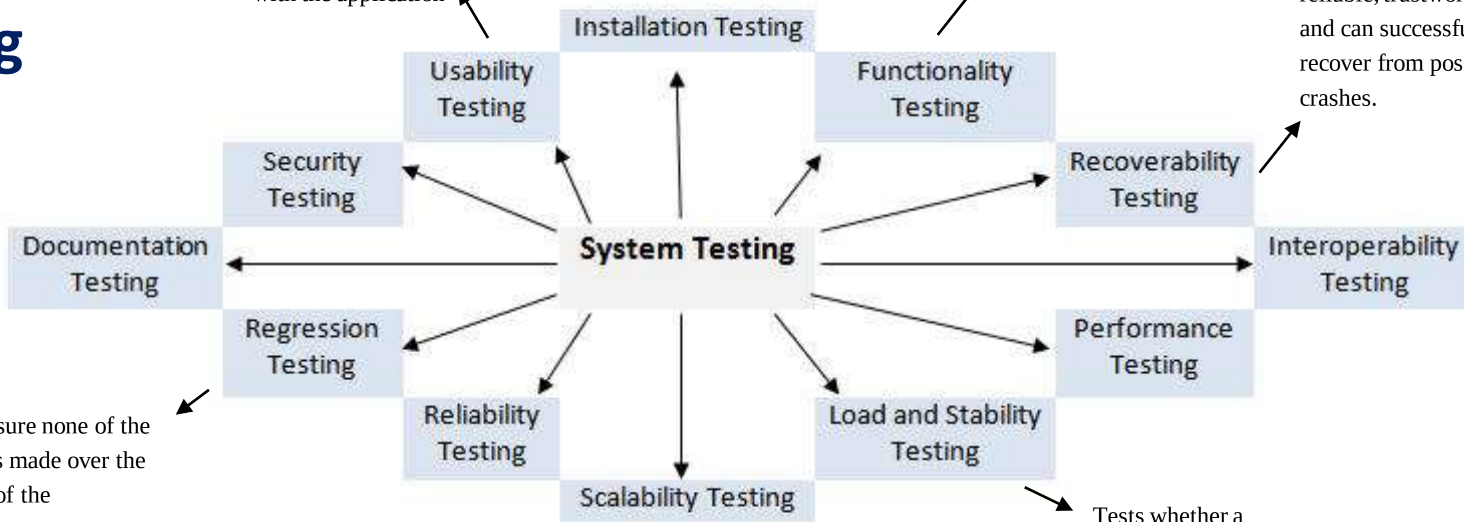
Integration testing	System Testing
Testing **integrated components** to check that they give the expected result	Testing the **completed product** to check that it meets the software specification requirements
Only **functional** testing is performed to check that the integrated components produce the expected outcome.	Testing of both **functional and non-functional requirements** are covered
Low level testing performed after unit testing	**High level testing** performed after integration testing
Both black box and white box testing are involved, so the testing requires knowledge of internal structure and code	**Black Box testing only**, so no knowledge of internal structure or code is required
Performed by **test engineers and developers**	Performed by **test engineers** only
Testing is performed on **integrated components**. Any defect found is attributed to a particular component	Testing is performed on **the system as a whole** including all the external interfaces. Any defect is regarded as defect of the whole system
Test cases are developed to simulate the **interaction between components**	Test cases are developed to **simulate real life scenarios**

Types of System Testing

Focuses on how easy it is for a user to interact with the application

Involves thinking of possible missing functions or additional functionalities that a product could have to improve it

Demonstrates that a software solution is reliable, trustworthy and can successfully recover from possible crashes.



Makes sure none of the changes made over the course of the development process have caused new bugs.

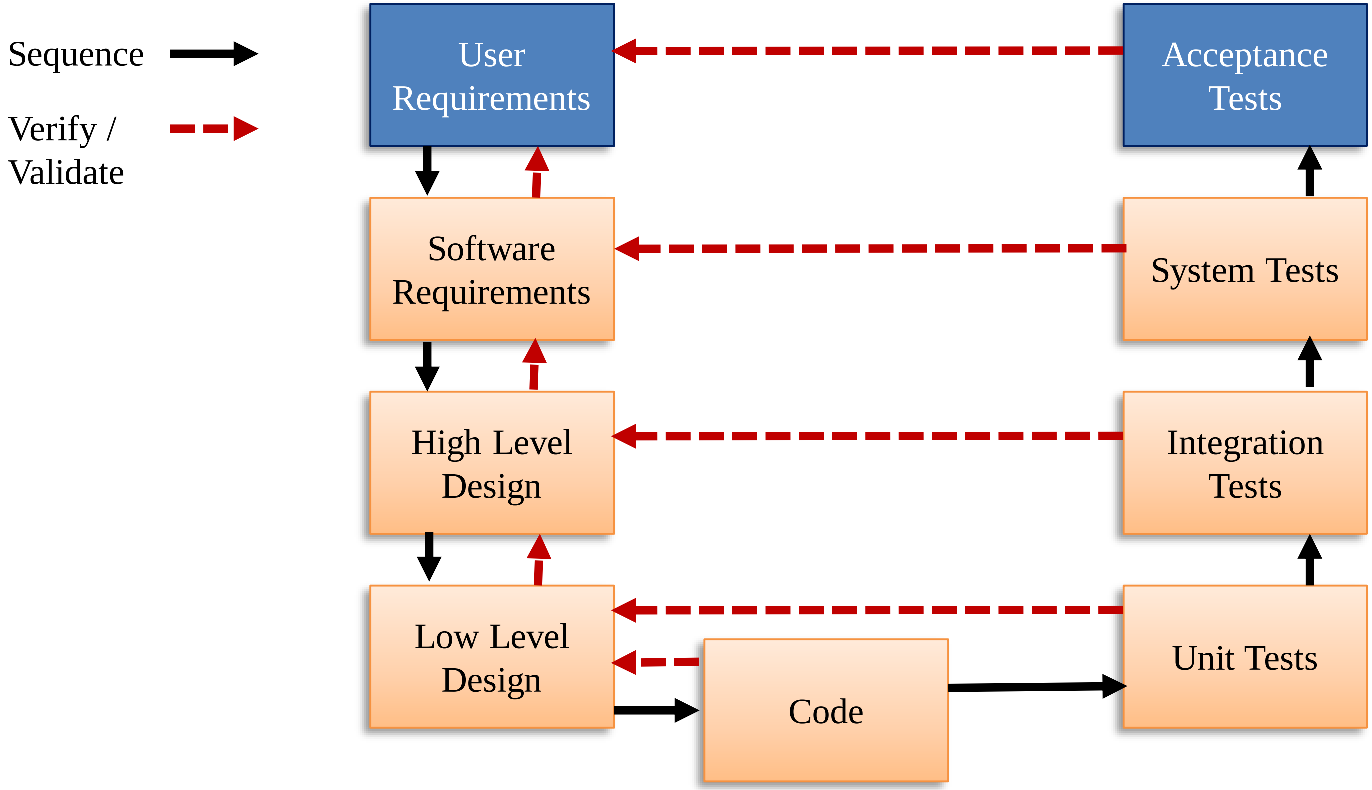
Tests whether a software solution will perform well under real-life loads.

- There are more than 50 types of System Test.
- Some of the more common test types are explained here

Acceptance Testing

Acceptance Testing

- Performed by the customer / end-user to ensure that the system does what they think it should.
- Acceptance tests often also focus on areas such as the subjective user experience.


Sequence
Verify / Validate
User Requirements
Acceptance Tests
Software Requirements
System Tests
High Level Design
Integration Tests
Low Level Design
Unit Tests
Code

System Testing / Acceptance Testing Differences and Similarities

- Both types of tests are executed against the entire system and it is possible that many of the tests will overlap.
- A system test is often executed by an independent QA team test of engineers in a simulated real world environment. This may be the first time the product has been tested as a whole system
- Acceptance testing can be run in the real customer environment, and the testing team consists of a subset of the **actual users** of the system.
- Actual users are usually able to identify scenarios and defects, and observe behaviours that a regular tester might overlook.
- The system test aligns with system level design (software requirements) and acceptance testing aligns with business/user requirements.

KSPs

Thanks

Next Lecture:

Software Deployment