中山大學
SUN YAT-SEN UNIVERSITY

Part III [Objectives]
1. Overview

SE-307 Software Testing Techniques

http://my.ss.sysu.edu.cn/wiki/display/SE307/Home

Instructor: Dr. Wang Xinming, School of Software, Sun Yat-Sen University

Previously ...

- Dimension 1: Problems
  - Test management
  - Test adequacy
  - Test generation
  - Test oracle
  - Test automation

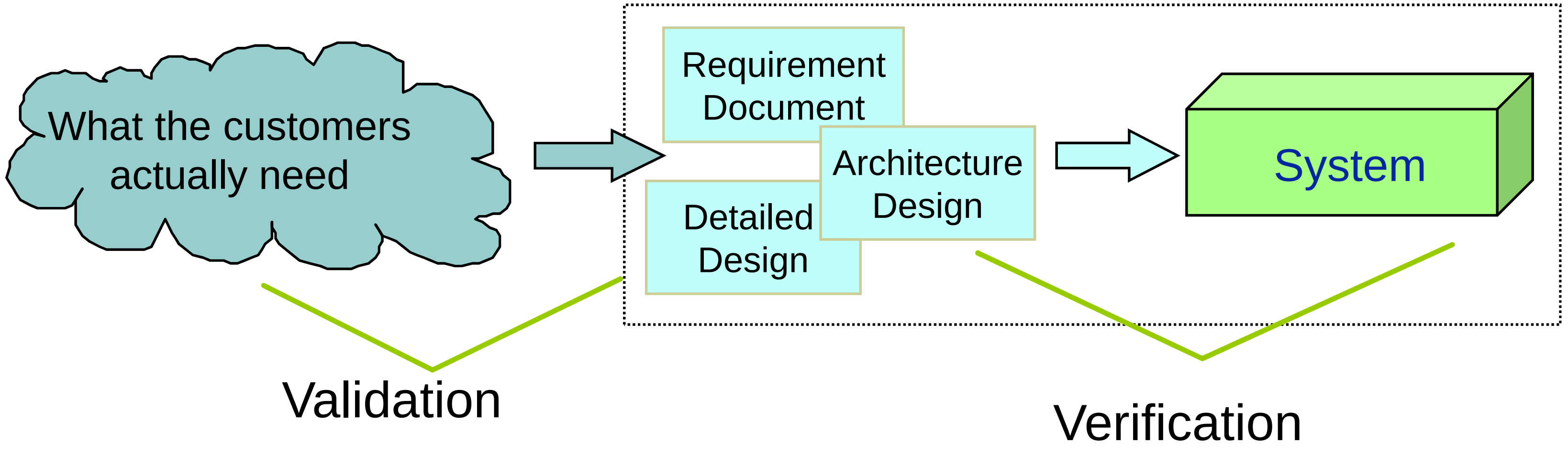
- To learn dimension 2 (objectives), we need to first clarify several concepts.

So Many Concepts in Testing!

- What are the exact meanings of these concepts? And what are their relations?
  - Unit testing, Integration testing, System testing, Acceptance testing
  - Functional testing, Non-functional testing
  - Black-box testing, White-box testing, Model-based testing
  - Alpha testing, Beta testing
  - Specification-based testing, Code-based testing
  - Structural testing, Behavior testing
  - Developer testing, QA team testing, User testing
  - Validation testing, Verification testing
  - Regression testing, Smoke testing, Continuous testing

They cover different aspects of testing
Our goal today: understand them.

Aspect 1: Validation vs. Verification



Aspect 1: Validation vs. Verification

- **Verification testing** *evaluates software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.*
  - It is often an internal process. (verification testing typically **does not involve** the customers)
  - It answers the questions like: **Am I building the product right?**
  - Unit, integration, and system testing are all verification testing.

- **Validation testing** *evaluate software during or at the end of the development process to determine whether it really addresses what the customers need.*
  - It often involves acceptance and suitability with customers. (validation testing typically **involve** the customers)
  - It answers the question like: **Am I building the right product?**
  - Acceptance testing is validation testing.

Aspect 2: Static vs. Dynamic

- **Static testing** is a form of software testing where the software isn't actually used.
  - Check the sanity of the requirement, design, and code (e.g. whether there is omission, contradiction, or conceptual error).
  - Some might argue against this term: testing is always dynamic.
  - Peer review, code inspection, program analysis are static testing.
- **Dynamic testing** is to examine the software by running it
  - Check whether the software behaves as expected on test inputs.
  - Most of our SE-307 course (e.g. the five problems) covers dynamic testing.
  - Unit, integration, system, and acceptance testing are dynamic testing.
- What are common:
  - Both can be either verification testing or validation testing.
  - Both can be done by either the developers (developer testing) or QA team (QA team testing)
  - Both can either check the behavior (black-box testing) or check the structure (white-box testing) of the software.

Types of Static Testing

- **Peer Reviews**
  - The least formal. Sometimes called buddy reviews, this method is really more of an "I'll show you mine if you show me yours" type discussion.
  - The programmer shows her/his code to one or two other programmers or testers who act as reviewers and receive casual comments.
- **Walkthroughs**
  - More formal. The programmer presents her/his source code to a small group (five or more) of programmers and testers.
  - The reviewers should receive copies of the source code in advance of the review, so that they can examine it and prepare comments and questions.
  - At the walkthrough, the presenter reads through the source code line by line, or function by function, explaining what it does and why. The reviewers listen and question anything that looks suspicious.
- **Inspections**
  - The most formal. They are highly structured and require training for each participant.
  - The presenter or reader isn't the original programmer. The other participants are called inspectors.
  - Some inspectors are also assigned tasks such as moderator and recorder to assure that the rules are followed and that the review is run effectively.

Faults Detected in Static Testing

- **Violation of coding standard:**
  - e.g. C++ coding standard

- **Data declaration errors**: improperly declaring or using variables or constants
  - Are all the variables assigned the correct length, type, and storage class?
  - If a variable is initialized at the same time as it's declared, is it properly initialized and consistent with its type?
  - Are there any variables with similar names?
  - Are any variables declared that are never referenced or are referenced only once?

Faults Detected in Static Testing

- **Data reference errors**: using a variable, constant, array, string, structure, or class that hasn't been properly declared or initialized.
  - Is an uninitialized variable referenced?
  - Are array and string subscripts integer values and are they always within the bounds of the array's or string's dimension?
  - Are there any potential “off by one” errors in indexing operations or subscript references to arrays？
  - Is a variable used where a constant would actually work better?
  - Is a variable ever assigned a value that's of a different type than the variable?
  - Is memory allocated for referenced pointers?
  - If a data structure is referenced in multiple functions or subroutines, is the structure defined identically in each one?

Faults Detected in Static Testing

- **Computation errors**: bad math
  - Do any calculations that use variables have different data types, such as adding an integer to a floating-point number?
  - Do any calculations that use variables have the same data type but are different lengths, such as adding a byte to a word?
  - Are the compiler's conversion rules for variables of inconsistent type or length understood and taken into account in any calculations?
  - Is the target variable of an assignment smaller than the right-hand expression?
  - Is overflow or underflow in the middle of a numeric calculation possible?
  - Is it ever possible for a divisor/modulus to be zero?
  - For cases of integer arithmetic, does the code handle that some calculations, particularly division, will result in loss of precision?
  - Can a variable's value go outside its meaningful range? For example, could the result of a probability be less than 0% or greater than 100%?
  - For expressions containing multiple operators, is there any confusion about the order of evaluation and is operator precedence correct? Are parentheses needed for clarification?

Faults Detected in Static Testing

- **Comparison errors**: boundary condition problems
  - Are the comparisons correct? It may sound pretty simple, but there's always confusion over whether a comparison should be less than or less than or equal to.
  - Are there comparisons between fractional or floating-point values? If so, will any precision problems affect their comparison? Is 1.00000001 close enough to 1.00000002 to be equal?
  - Does each Boolean expression state what it should state? Does the Boolean calculation work as expected? Is there any doubt about the order of evaluation?
  - Are the operands of a Boolean operator Boolean? For example, is an integer variable containing integer values being used in a Boolean calculation?

Faults Detected in Static Testing

- **Control flow errors**: loops and other control constructs in the language not behaving as expected.
  - If the language contains statement groups such as if...else and do...while, are the ends explicit and do they match their appropriate groups?
  - Will the program, module, subroutine, or loop eventually terminate? If it won't, is that acceptable?
  - Is there a possibility of premature loop exit?
  - Is it possible that a loop never executes? Is it acceptable if it doesn't?
  - If the program contains a multi-way branch such as a switch...case statement, can the index variable ever exceed the number of branch possibilities? If it does, is this case handled properly?
  - Are there any "off by one" errors that would cause unexpected flow through the loop?

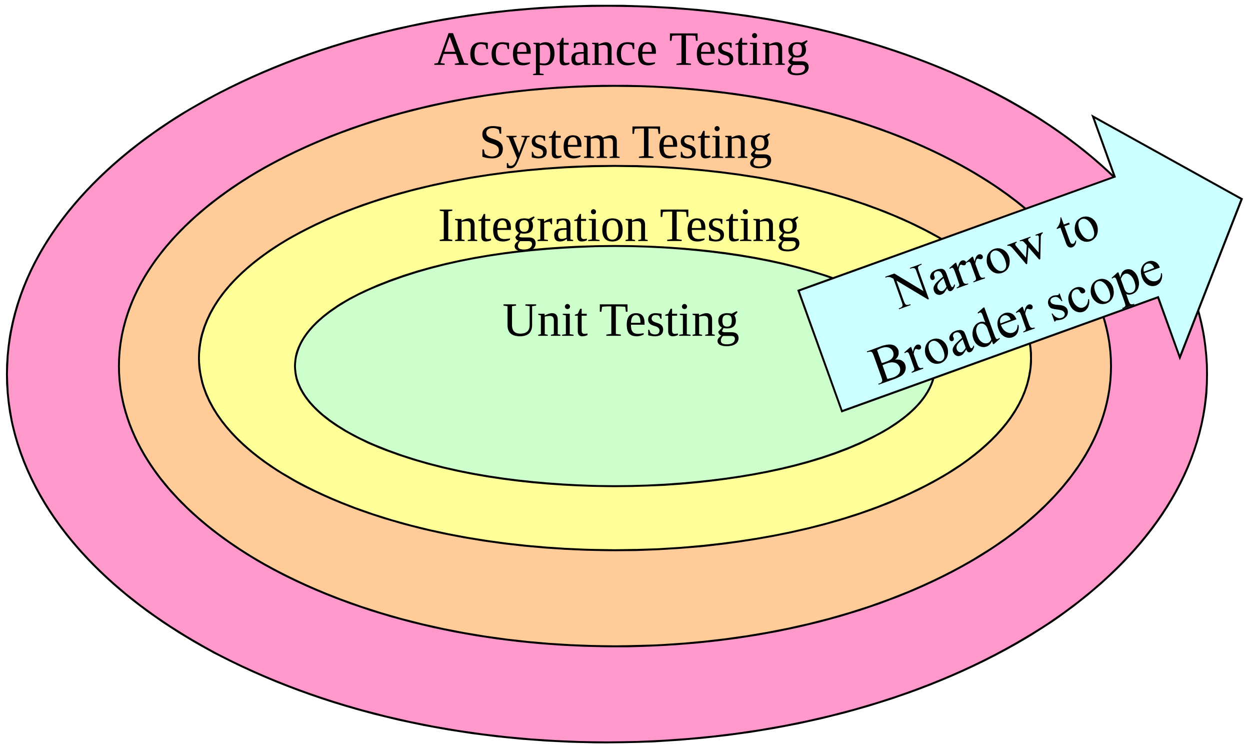
Faults Detected in Static Testing

- **Subroutine parameter errors**: incorrect passing of data to and from software subroutines.
  - Do the types and sizes of parameters received by a subroutine match those sent by the calling code? Is the order correct?
  - If constants are ever passed as arguments, are they accidentally changed in the subroutine?
  - Does a subroutine alter a parameter that's intended only as an input value?
  - Do the units of each parameter match the units of each corresponding argument English versus metric, for example?
  - If global variables are present, do they have similar definitions and attributes in all referencing subroutines?

Faults Detected in Static Testing

- **Input or output errors:**
  - Does the software strictly adhere to the specified format of the data being read or written by the external device?
  - If the file or peripheral isn't present or ready, is that error condition handled?
  - Does the software handle the situation of the external device being disconnected, not available, or full during a read or write?
  - Are all conceivable errors handled by the software in an expected way?
  - Have all error messages been checked for correctness, appropriateness, grammar, and spelling?

Aspect 3: The Level of Testing



Aspect 3: The Level of Testing

- **Unit testing**: *tests individual module to determine whether they correctly implement what specified by the software design*.
  - Require the actual execution of the software (unit testing is dynamic testing, not static testing)
  - Check against the software design. The customers are not involved (unit testing is verification testing, not validation testing)
  - Mostly written and run by software developers (unit testing is typically developer testing, not independent testing).
  - Designed based on the knowledge of source code (unit testing is typically white-box testing, not black-box testing)
  - Does not directly address user requirements (indirectly through the software design). (unit testing is neither functional testing nor non-functional testing)

Target of Unit Testing

- Concentrates on the internal processing logic and data structures of a module (e.g. method or class)
  - Module input/output
    - Ensure that information flows properly into and out of the module
  - Local data structures
    - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
  - Boundary conditions
    - Ensure that the module operates properly at boundary values established to limit or restrict processing
  - Basis paths
    - Paths are exercised to achieve control-/data- flow coverage.
  - Exception handling
    - Ensure that the module responds correctly to specific error conditions

Faults Detected in Unit Testing

- Misunderstood or incorrect arithmetic precedence
- Mixed mode operations (e.g., int, float, char)
- Incorrect initialization of values
- Precision inaccuracy and round-off errors
- Incorrect symbolic representation of an expression (e.g. int vs. float)
- Comparison of different data types
- Incorrect logical operators or precedence
- Expectation of equality when precision error makes equality unlikely (using == with float types)
- Incorrect comparison of variables
- Improper or nonexistent loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables
- Boundary value violations
- Exception condition processing is incorrect

Aspect 3: The Level of Testing

- **Integration testing:** *tests the interfaces of modules and their interaction through these modules*.
  - Require the actual execution of the software (integration testing is dynamic testing, not static testing)
  - Check against software requirements. The customers are not involved (integration testing is verification testing, not validation testing)
  - Done by either software developers or QA team, depended on the granularity of integration (integration testing can be either developer testing or QA team testing).
  - Designed based on the knowledge of either software behavior or source code (integration testing can be either white-box or black-box testing)
  - Verify both functional and non-functional requirements. (integration testing can be either functional testing or non-functional testing)

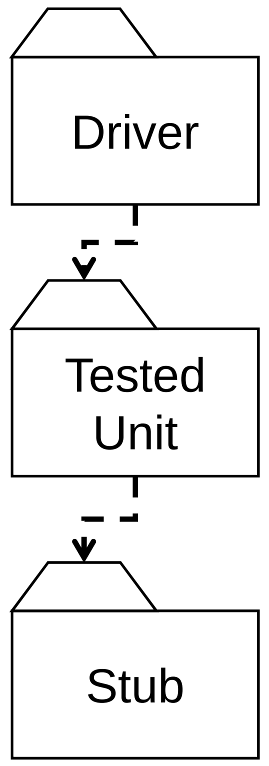
Target of Integration Testing

- The entire system is viewed as a collection of modules (sets of methods, classes, sub-systems) determined during the system and object design
    - Goal: Test all interfaces between modules and the interaction of modules

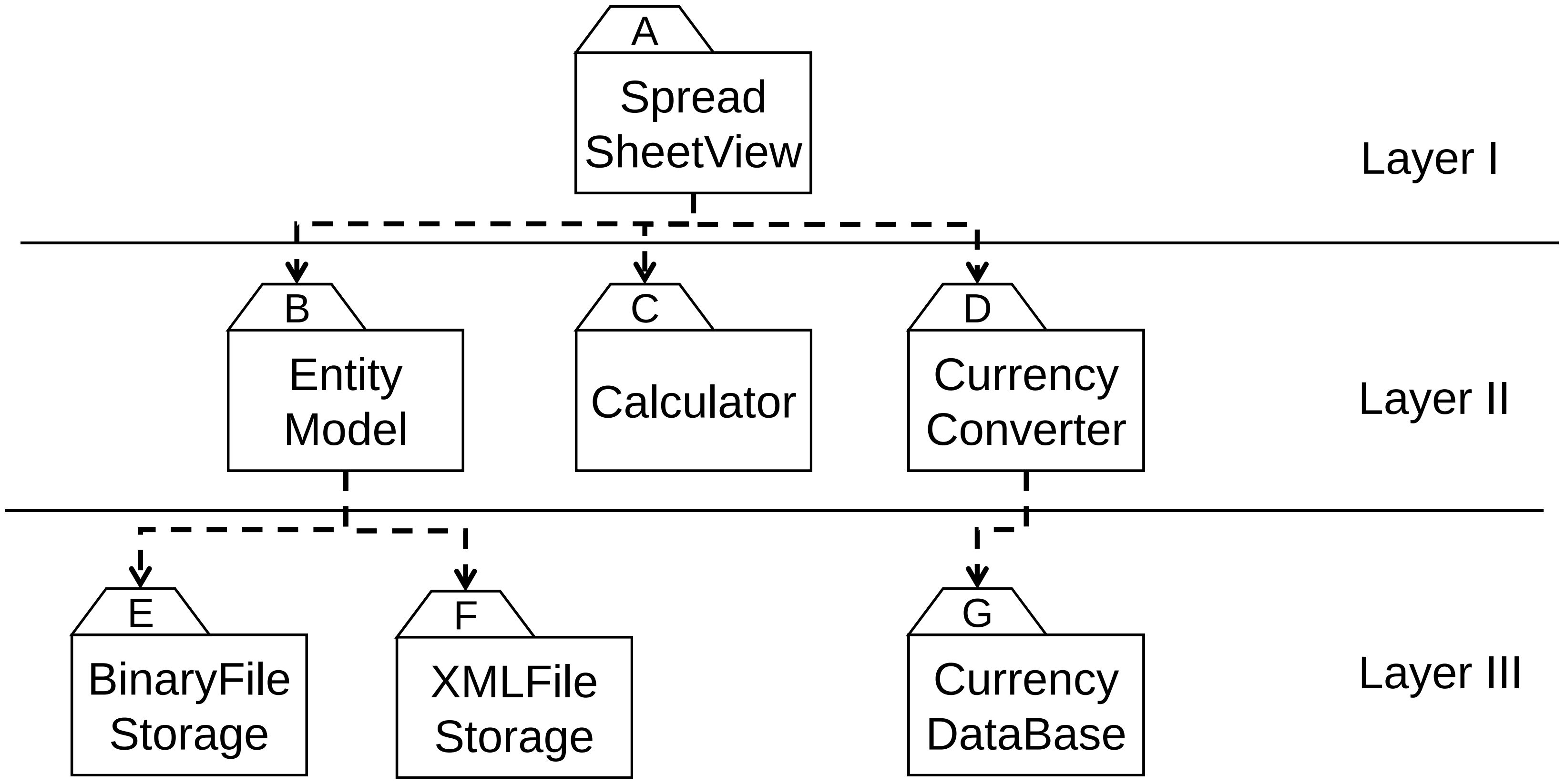
- Types of Integration Testing
    - Big-bang, Top-down, Bottom-up, and Sandwich
        - Assume a conventional software development lifecycle (design $\rightarrow$ coding $\rightarrow$ unit test $\rightarrow$ integration testing)
    - Continuous testing
        - Assume an agile lifecycle

Facility for Integration Testing

- **Driver**:
  - A component, that calls the `TestedUnit`
  - Controls the test cases
- **Stub**:
  - A component, the `TestedUnit` depends on
  - Partial implementation
  - Returns fake values.

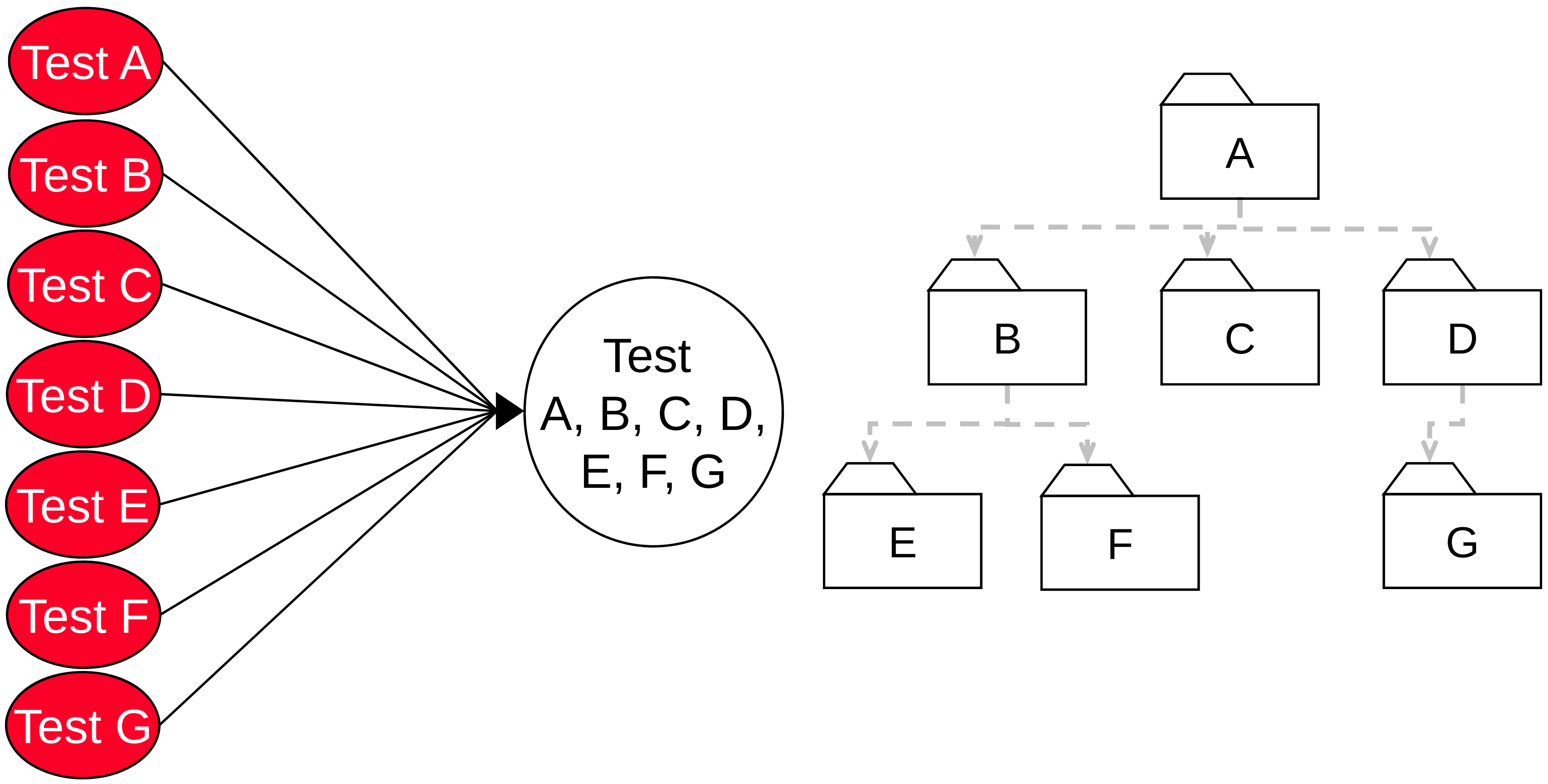


Example: A 3-Layer-Design



(TerpSpreadsheet)

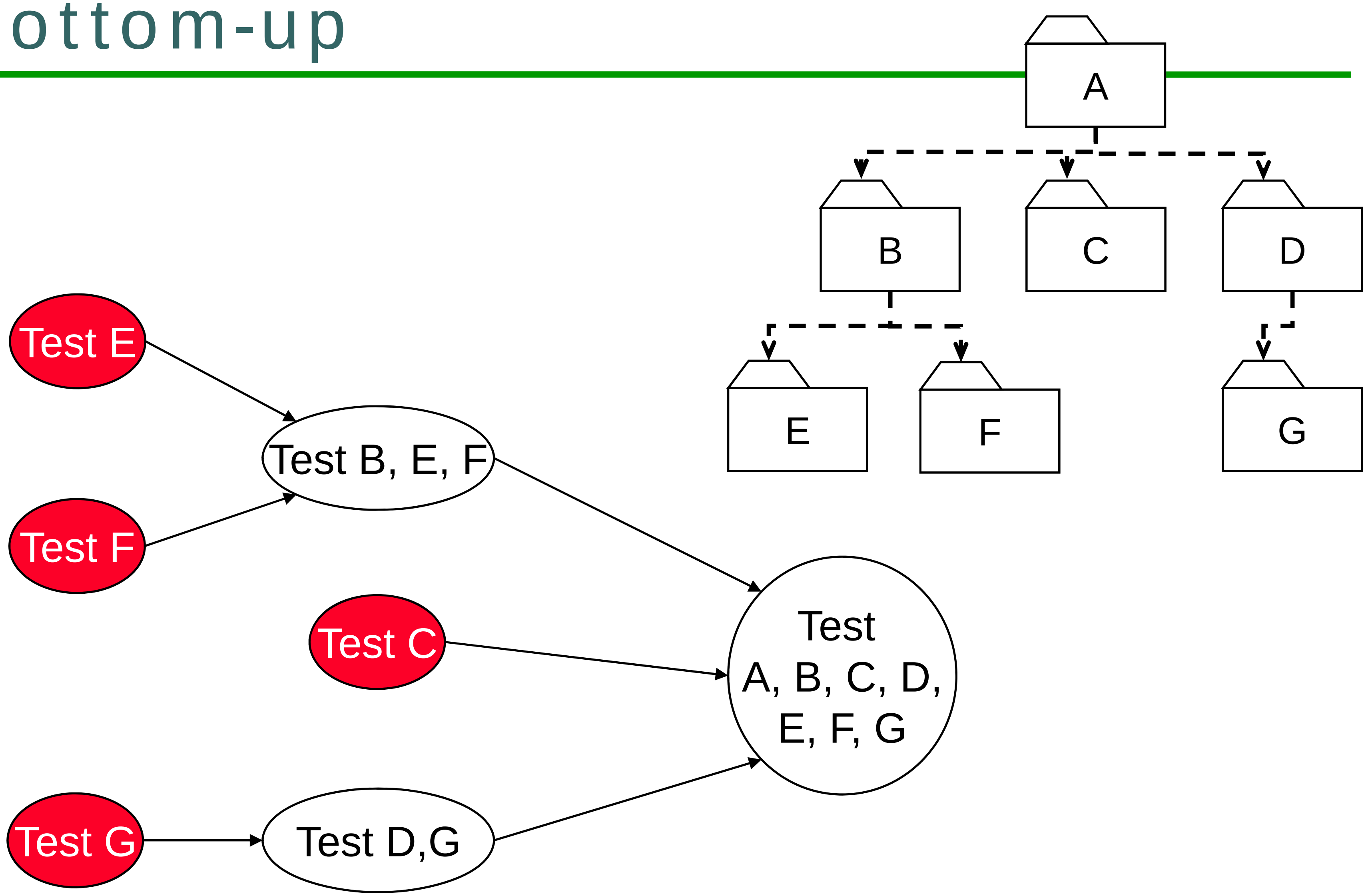
Big-Bang


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

Bottom-up

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously tested subsystems
- This is repeated until all subsystems are included
- Drivers are needed.

Bottom-up


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

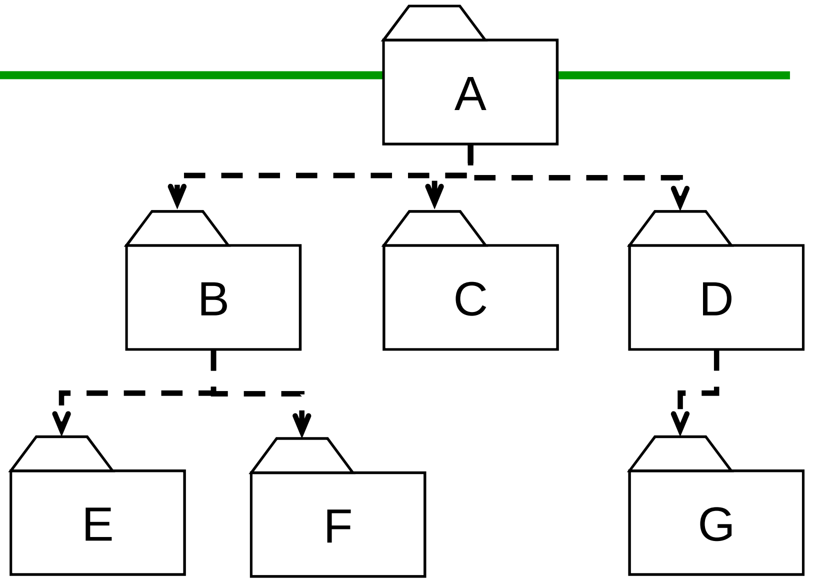
Pros and Cons

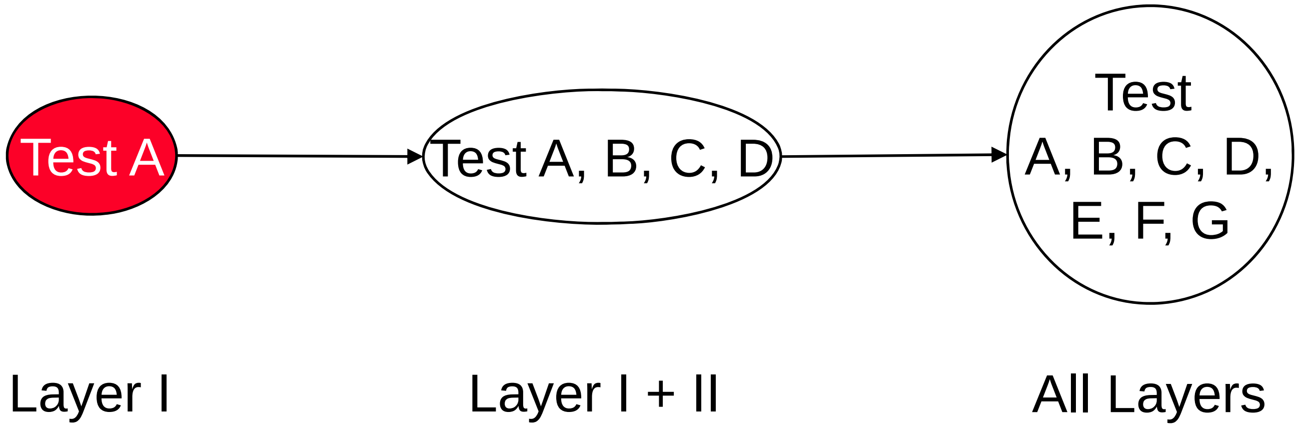
- Con:
  - Tests the most important sub-system (user interface) last
  - Drivers needed
- Pro
  - No stubs needed

Top-down

- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Stubs are needed to do the testing.

Top-down


A
B
C
D
E
F
G


Test A
Test A, B, C, D
Test A, B, C, D, E, F, G
Layer I
Layer I + II
All Layers

Pros and Cons

Pro

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- No drivers needed

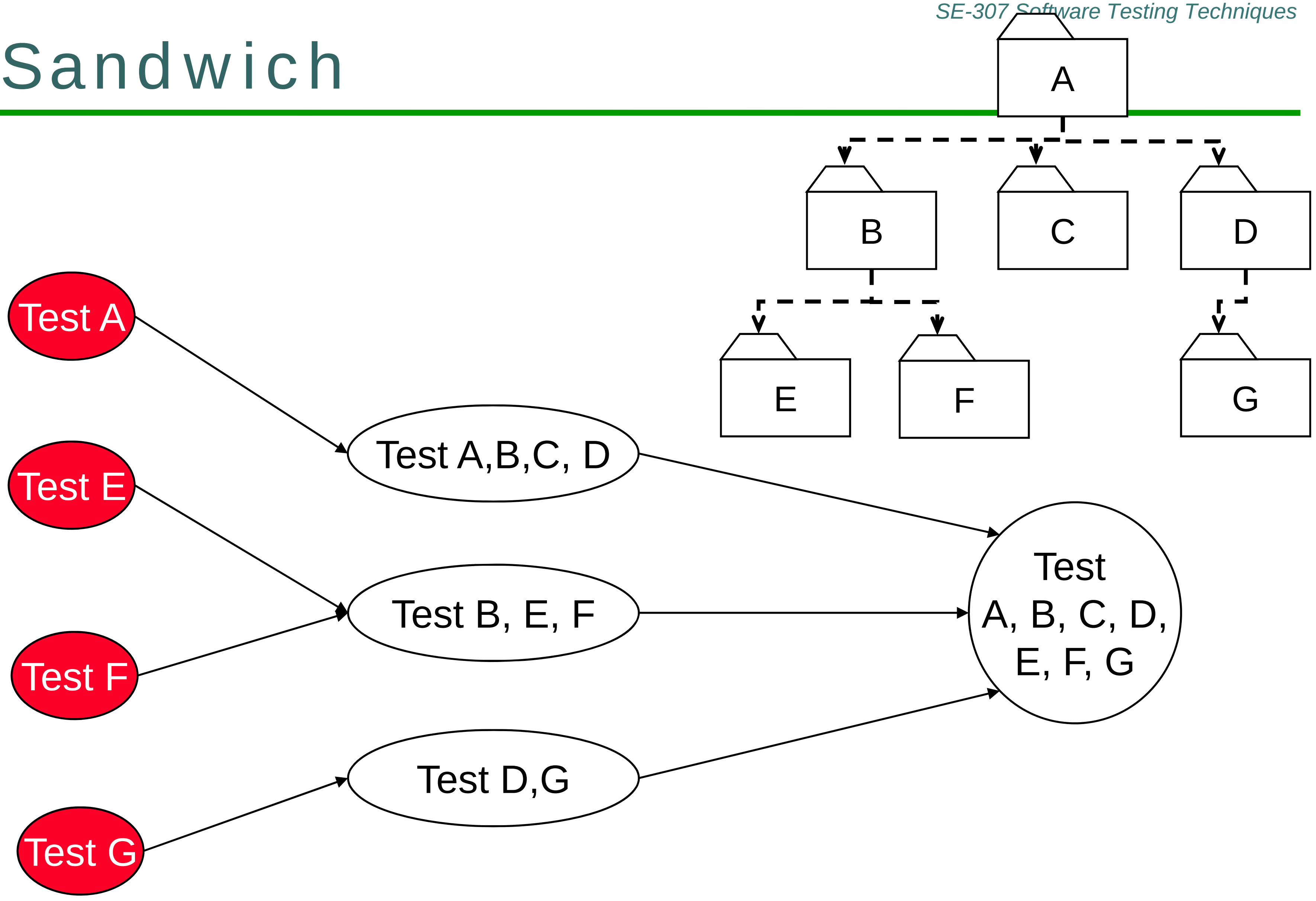
Cons

- Writing stubs is difficult: Stubs must allow all possible conditions to be tested.
- Large number of stubs may be required, especially if the lowest level of the system contains many methods.
- Some interfaces are not tested separately.

Sandwich

- Combines top-down strategy with bottom-up strategy
- 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.

Sandwich


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

Pros and Cons

- Top and Bottom Layer Tests can be done in parallel
- Problem: Does not test the individual subsystems and their interfaces thoroughly before integration
- Solution: Modified sandwich testing strategy

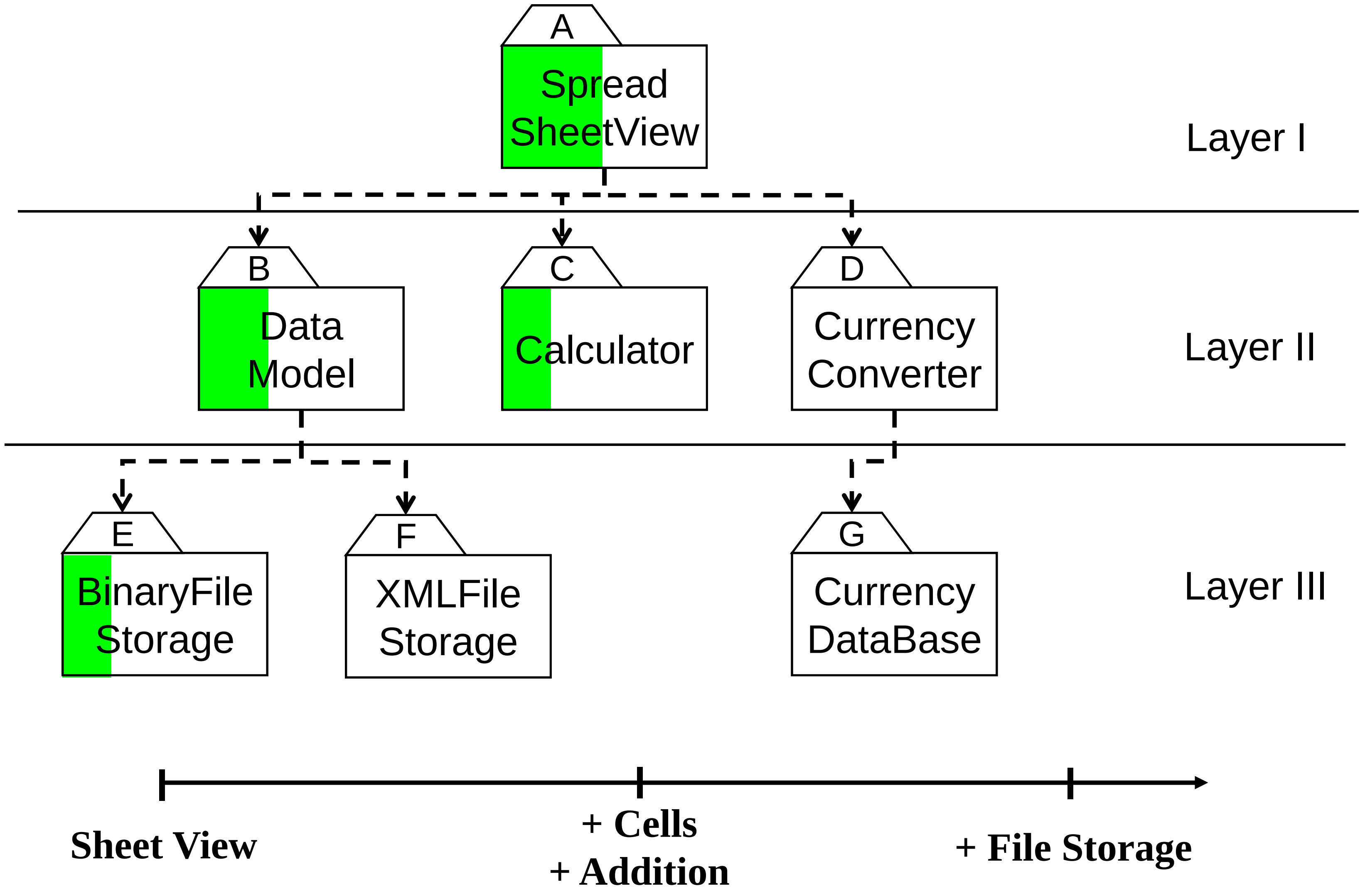
Continuous Testing

- **Continuous testing** assumes an agile development process that deploys *continuous build* (持续集成):
  - Build from day one
  - Test from day one
  - Integrate from day one

  ⟹ System is always runnable

- Requires tool support:
  - Continuous build server (daily build)
  - Automated test cases.
  - Software configuration management

Continuous Testing



Aspect 3: The Level of Testing

- **System testing**: *conducted on a complete, integrated system to test the system's compliance with its specified requirements.*
  - Require the actual execution of the software (system testing is dynamic testing, not static testing)
  - Check against software requirements. The customers are not involved (system testing is verification testing, not validation testing)
  - Designed based on the knowledge of software behavior (system testing is typically blackbox testing)
  - Typically done by the QA team (system testing is QA team testing , not developer testing).
  - Verify functional and non-functional requirements. (system testing can be either functional testing or non-functional testing)

Aspect 3: The Level of Testing

- **Acceptance testing**: *done by the customers to validate whether the software really provides what they need.*
  - Require the actual execution of the software (acceptance testing is dynamic testing, not static testing)
  - The customers are involved (acceptance testing is validation testing)
  - Focuses on user-visible actions and user-recognizable output from the system.
  - Designed based on the knowledge of software behavior (acceptance testing is blackbox testing)
  - Validate functionality, performance, and reliability, etc. (acceptance testing can be either functional testing or non-functional testing)
    - Especially for some quality factors that might be overlooked in other levels of testing, such as usability, compatibility, maintainability.

Alpha and Beta Testing

- Alpha and beta testing are two special kinds of acceptance testing
- **Alpha testing** is conducted at the developer's site by customers
  - Software is used in a natural setting with developers watching intently
  - Testing is conducted in a controlled environment
- **Beta testing** is conducted at customer's site by customers
  - Developer is generally not present
  - It serves as a live application of the software in an environment that cannot be controlled by the developer
  - The customer records all problems that are encountered and reports these to the developers at regular intervals
- After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base

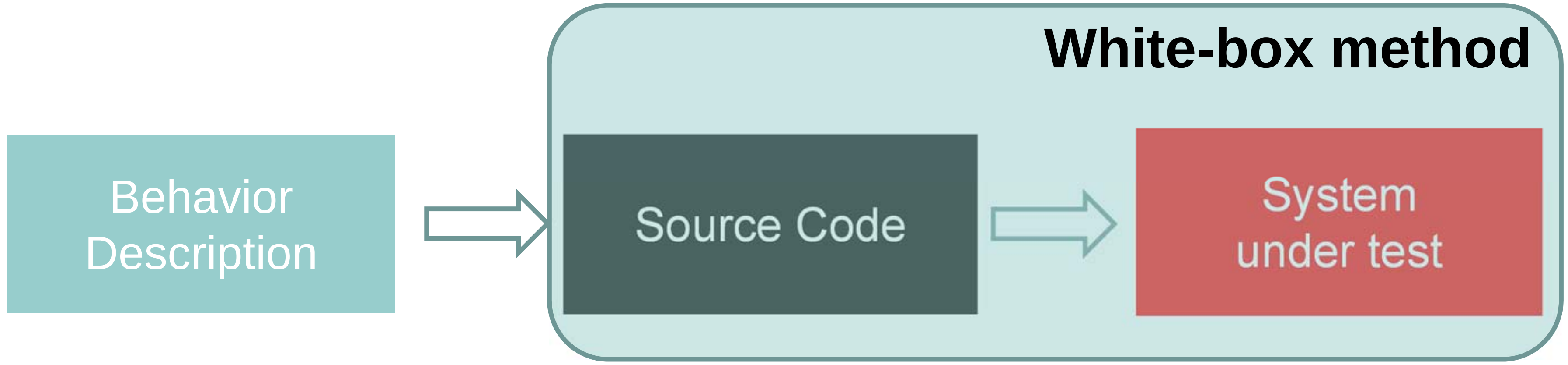
Aspect 4: The Objective of Testing

- **Functional testing**: verify that the software performs and functions correctly according to the functionality requirements.
  - Typically answer questions like "can the user do this" or "does this particular feature work"
- **Non-functional testing**: verify the non-functional requirements of the software, such as reliability, performance, etc.
  - These non-functional requirements are NOT directly related to any specific feature (or we shall say that they apply to all features).
- What are common:
  - Integration, system and acceptance testing can be either functional or non-functional testing.
  - Unit testing is neither functional nor non-functional testing, as it checks against software design, not requirements.
    - e.g., testing the constructor/destructor of a class.
  - Typically, both use black-box testing method (Both functional testing and non-functional testing are typically black-box testing)
    - There are some exceptions, though. e.g. use code analysis to verify the security of the program.

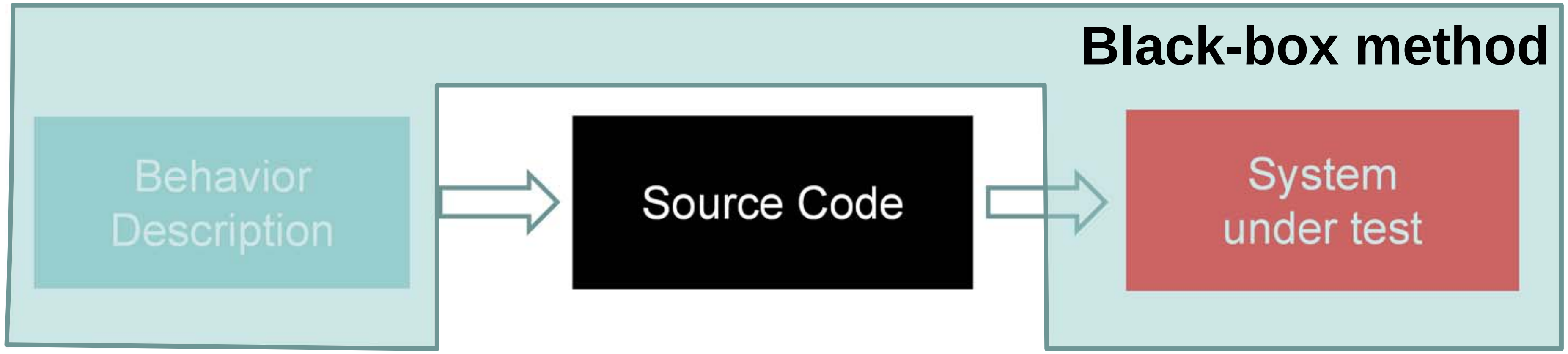
Examples of Non-functional Testing

- **Recovery testing**
  - Tests for recovery from system faults
  - Forces the software to fail in a variety of ways and verifies that recovery is properly performed. Tests re-initialization, check-pointing mechanisms, data recovery, and restart for correctness
- **Security testing**
  - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access
- **Load testing**
  - Tests the run-time performance of software within the context of an integrated system
  - Often coupled with stress testing and usually requires both hardware and software instrumentation
  - Can uncover situations that lead to degradation and possible system failure
- **Stress testing**
  - Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- **Usability testing**
  - Evaluate the usability of the software by the customers.

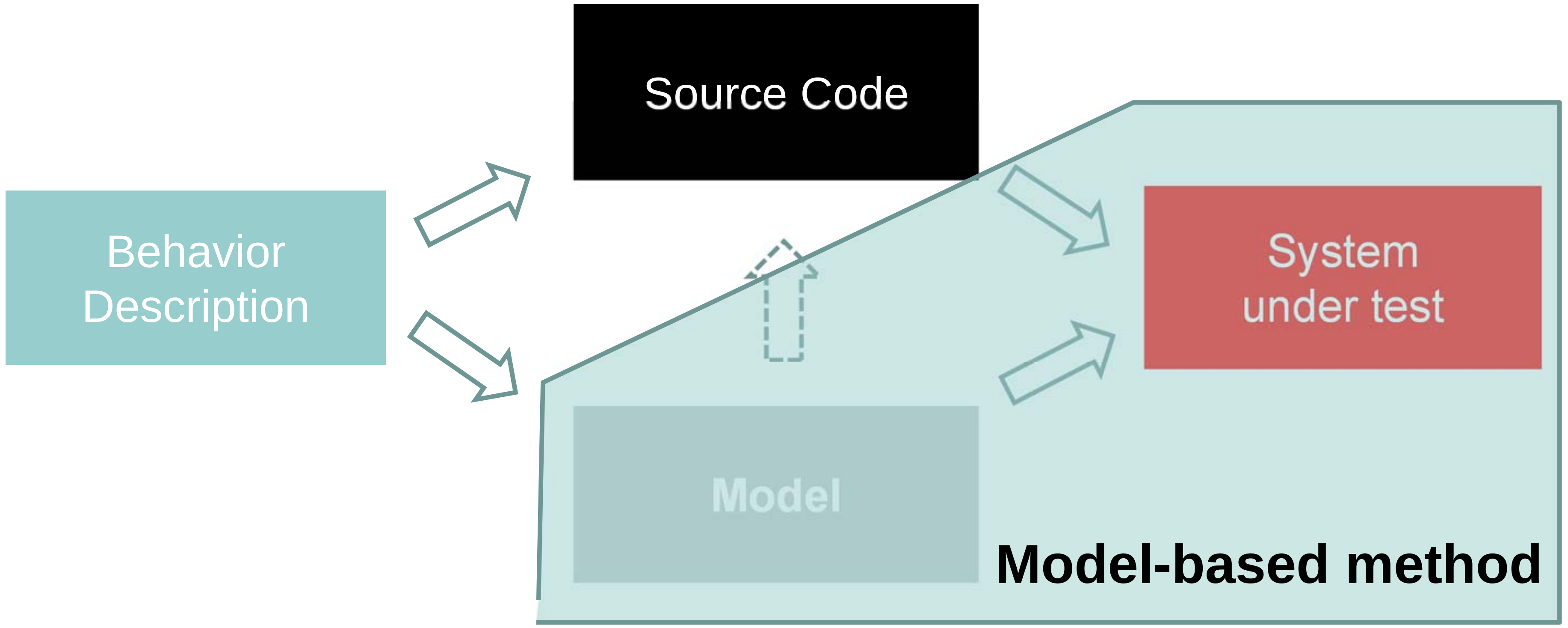
Aspect 5: The Method of Testing



Aspect 5: The Method of Testing



Aspect 5: The Method of Testing



Aspect 5: The Method of Testing

- **Black-box testing**: *a method of software testing that examines the behavior of an application without peering into its internal structures or workings*.
  - Also known as **specification-based testing**, **behavior based testing**, or **behavioral testing.**
  - Black-box testing can be applied at any level of the software testing process.
    - Black-box unit testing: test the unit at its interface (e.g. input/output parameters) without looking into its source code.
  - Which problem the black-box method address:
    - **black-box test adequacy**: measure the adequacy of test suite by how the test cases cover the behaviors.
    - **black-box test generation**: use the knowledge of the behavior (e.g. input structure and constraints) to generate test data that satisfies the desired test adequacy (can be either white-box or black-box adequacy).
    - **black-box test oracle**: use program output to check whether the test cases pass/fail.

Aspect 5: The Method of Testing

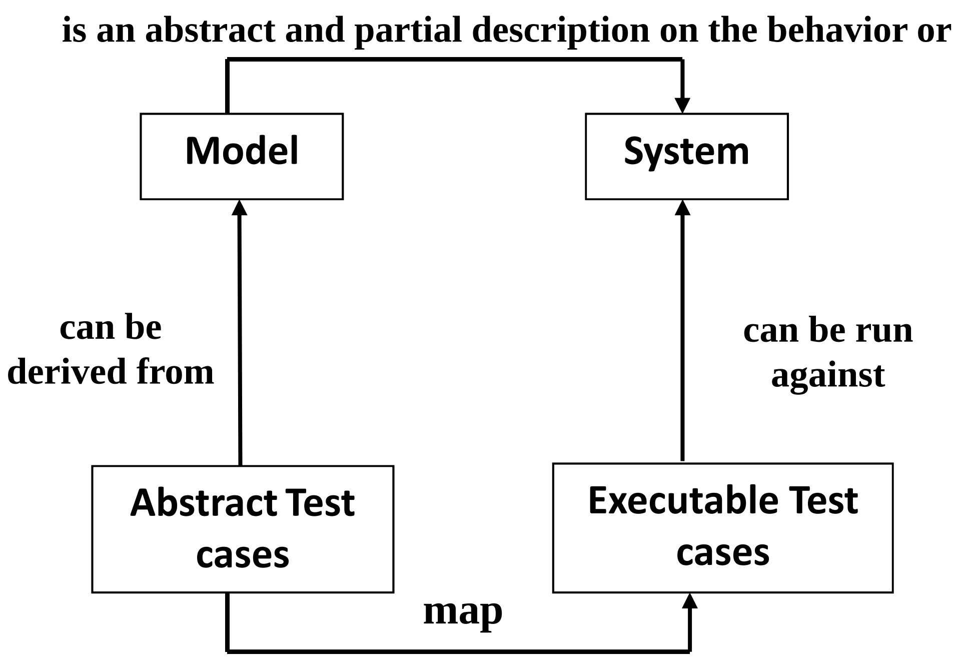
- **White-box testing**: *a method of software testing that examines the internal structures or workings of the software*.
  - Also known as **structure-based testing**, **structural testing**, or **code-based testing**
  - Although theoretically white-box testing can be applied at any level of the software testing process, it is typically applied in unit testing and low-level integration testing only.
  - Which problem the white box method address:
    - **white-box test adequacy**: measure the adequacy of test suite by how the test cases cover code units.
    - **white-box test generation**: use the knowledge of source code to generate test data that satisfies the desired test adequacy (typically white-box adequacy).
    - **white-box test oracle**: use internal program variables to check whether the test cases pass/fail.

Aspect 5: The Method of Testing

- **Model-based testing**: *a special kind of black-box testing that is based on a rigorous (typically formal) model of the expected software behavior*.
  - The model is usually an abstract, partial presentation of the desired behavior.
    - Examples: finite-state machine, context-free grammars (push-down machine).
  - Which problem the model-based method address:
    - **Model-based test adequacy**: measure the adequacy of test suite by how the test cases cover the model (e.g. state/transition coverage).
    - **Model-based test generation**: use the model to generate test cases (e.g. grammar-based testing)
    - **Model-based test oracle**: use the model to specify the expect output (e.g. after a sequence of input, the program shall reach a certain state in the finite-state machine).

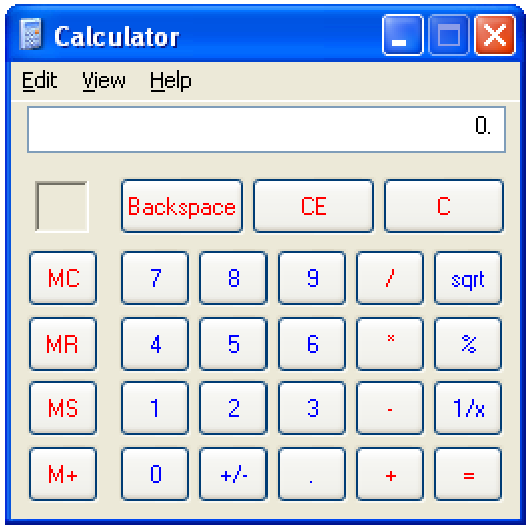
A More Formal Definition

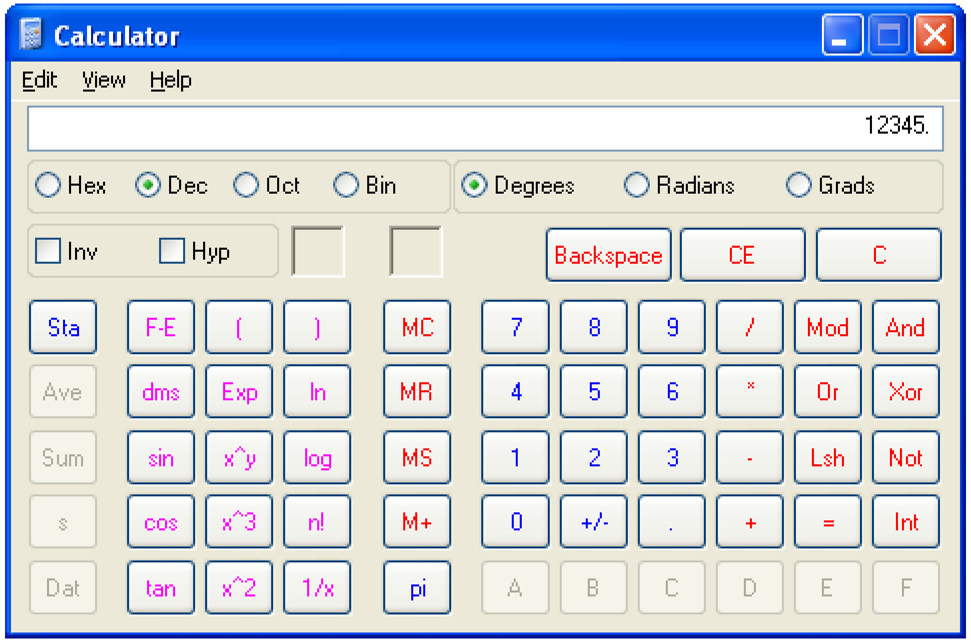
Model-based testing is a process of deriving abstract test cases from a model that abstract certain behaviours of the system under test and mapping those abstract test cases to concrete executable test cases



Partiality and Abstraction

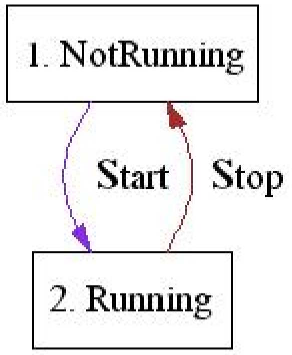
- Models are partial and abstract description of the system under test.
  - First of all, models do not need to cover all features of the system.
  - Secondly, for each covered feature, models do not need to cover all the details of the related behaviors.
- Example: calculator

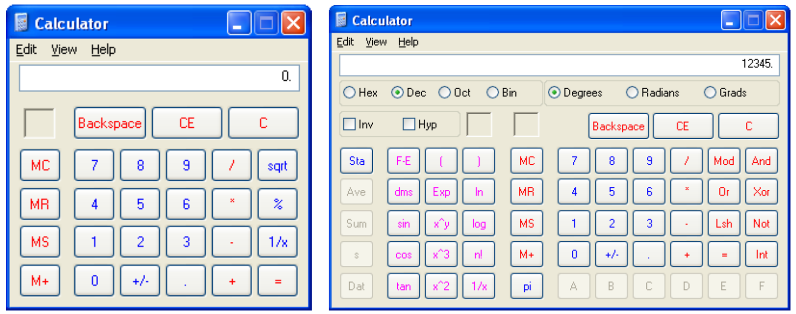



Calculator
Edit View Help
12345.
Hex Dec Oct Bin
Degrees Radians Grads
Inv Hyp
Backspace CE C
Sta F-E ( ) MC 7 8 9 / Mod And
Ave dms Exp ln MR 4 5 6 * Or Xor
Sum sin x^y log MS 1 2 3 - Lsh Not
s cos x^3 n! M+ 0 +/- . + = Int
Dat tan x^2 1/x pi A B C D E F

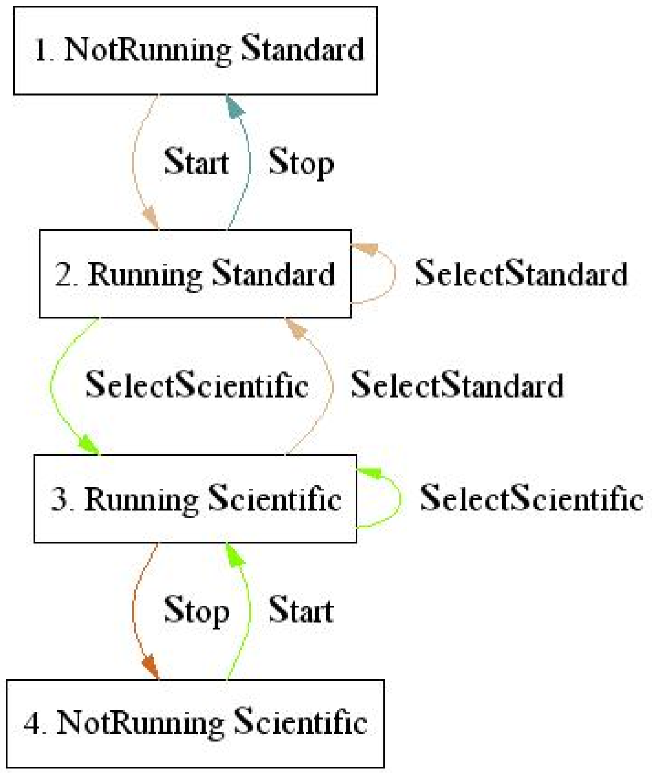
Models of Different Abstraction Levels

Model #1

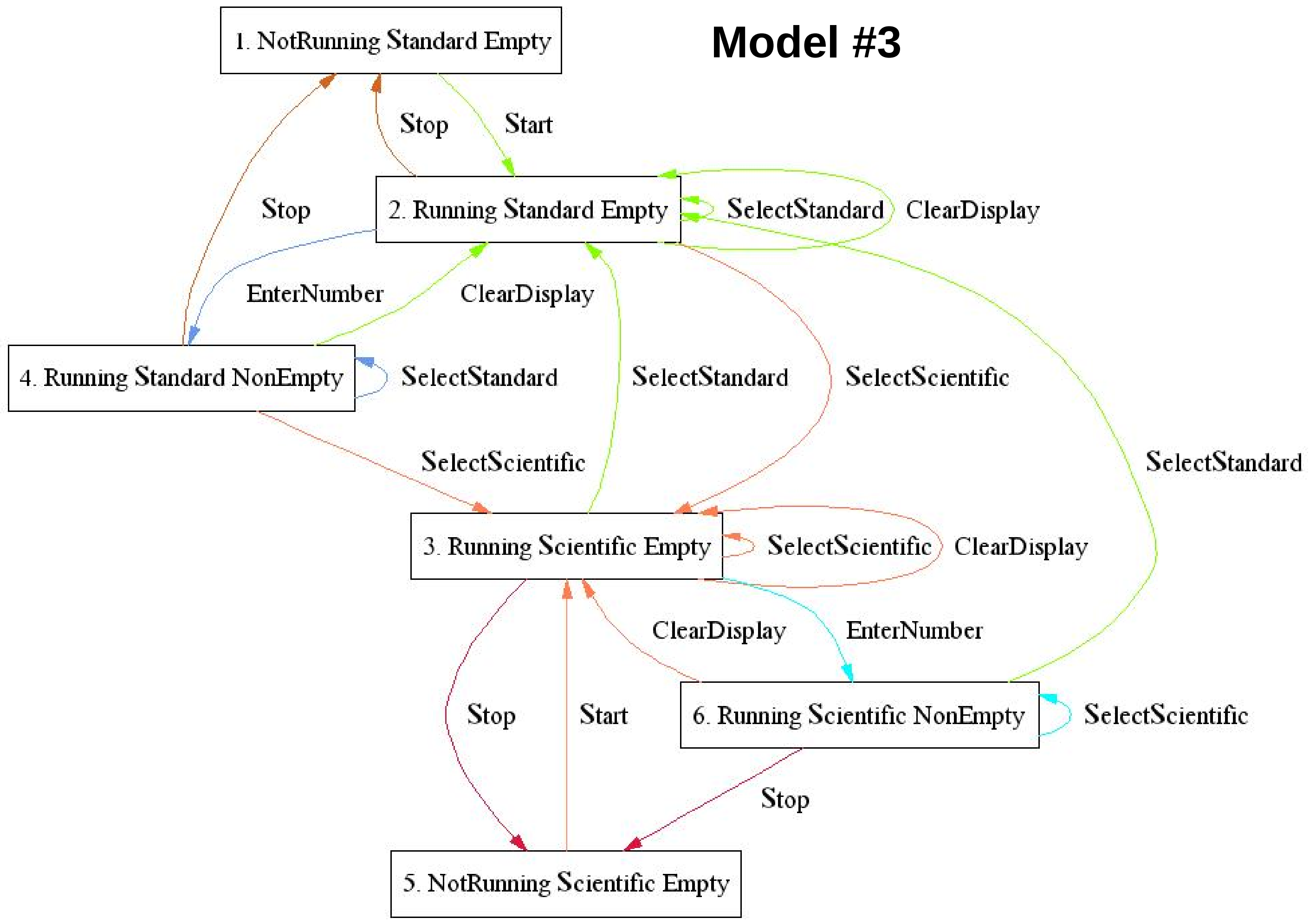




Model #2



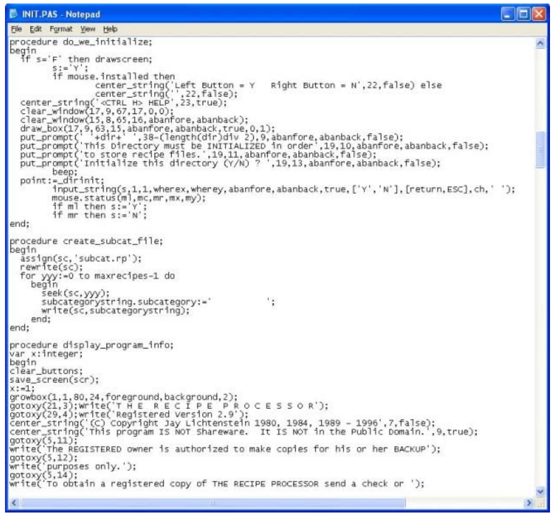
Models of Different Abstraction Levels



Models of Different Abstraction Levels

- The source code is the ultimate model!
  - Complete
  - Zero level of abstraction

Model #4



```
procedure do_we_initialize;
begin
  if s='F' then drawscreen;
        s:='Y';
        if mouse.installed then
                center_string('Left Button = Y   Right Button = N',22,false) else
                center_string('',22,false);
  center_string('<CTRL H> HELP',23,true);
  clear_window(17,9,67,17,0,0);
  clear_window(15,8,65,16,abanfore,abanback);
  draw_box(17,9,63,15,abanfore,abanback,true,0,1);
  put_prompt(' '+dir+' ',38-(length(dir)div 2),9,abanfore,abanback,false);
  put_prompt('This Directory must be INITIALIZED in order',19,10,abanfore,abanback,false);
  put_prompt('to store recipe files.',19,11,abanfore,abanback,false);
  put_prompt('Initialize this directory (Y/N) ? ',19,13,abanfore,abanback,false);
        beep;
  point:=_dirinit;
        input_string(s,1,1,wherex,wherey,abanfore,abanback,true,['Y','N'],[return,ESC],ch,' ');
        mouse.status(ml,mc,mr,mx,my);
        if ml then s:='Y';
        if mr then s:='N';
end;

procedure create_subcat_file;
begin
  assign(sc,'subcat.rp');
  rewrite(sc);
  for yyy:=0 to maxrecipes-1 do
    begin
      seek(sc,yyy);
      subcategorystring.subcategory:='           ';
      write(sc,subcategorystring);
    end;
end;

procedure display_program_info;
var x:integer;
begin
clear_buttons;
save_screen(scr);
x:=1;
growbox(1,1,80,24,foreground,background,2);
gotoxy(21,3);write('T H E  R E C I P E  P R O C E S S O R');
gotoxy(29,4);write('Registered Version 2.9');
center_string('(C) Copyright Jay Lichtenstein 1980, 1984, 1989 - 1996',7,false);
center_string('This program IS NOT Shareware.  It IS NOT in the Public Domain.',9,true);
gotoxy(5,11);
write('The REGISTERED owner is authorized to make copies for his or her BACKUP');
gotoxy(5,12);
write('purposes only.');
gotoxy(5,14);
write('To obtain a registered copy of THE RECIPE PROCESSOR send a check or ');
```

Various Kind of Models

- "Small", "partial" (and more practical) models:
  - Finite state model. e.g. UML state diagrams
  - Context-free grammar. e.g. XML grammar, CSS grammar
- "Big", "complete" (and more theoretical) models:
  - Abstract models. e.g. VDM, Z
  - Algebraic models. e.g. OBJ3, FOOPS
  - Concurrency models. e.g. CSP, CCS, Petri nets

Aspect 6: Who Tests

- Developer testing
  - Test cases written from the developer's perspective.
  - Tends to focus on the structure (developer testing is typically white-box testing)
- QA team testing
  - Also known as independent testing
    - Independent from the developers. Without the knowledge of how the software is implemented (QA team testing is typically black-box testing)
  - Test cases written from the customer’s perspective, but testing is conducted by the QA team.
- User testing
  - Usually another name of acceptance testing.

Aspect 7: New or Old

- **Regression testing** re-executes test cases that have already been conducted to ensure that changes to the software do not introduce unintended behavior.
  - We don’t need to “design” regression test cases – the test cases are already there.
    - They might need maintenance, though.
  - The execution can be done manually or automatically.
  - Verify whether the modified software passes the test cases that have passed on the original version (regression testing is verification testing).
  - Can be either functional or non-functional testing.
  - Can be either white-box testing or black-box testing.

Aspect 7: New or Old

- **Smoke testing** is a special kind of regression testing that is executed immediately after the whole system is built.
  - Taken from the world of hardware. Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure
  - Typically a small set of test cases chosen to expose "show stopper" faults (e.g. crash on start).
    - Test the critical functionality of the system (smoke testing runs black-box, system test cases).
    - The goal is to uncover errors that have the highest likelihood of throwing the software project behind schedule.
    - Typically functional test cases only, ignoring non-functional test cases.
  - After a smoke test is completed, a more detailed regression test might be conducted.

Quiz: Mark Related Concepts

And explain why!

	Unit testing	Integratio n testing	System testing	Acceptan ce testing	Function al testing	Non-functional testing	Black-box testing	White-box testing	Developer testing	QA team testing	Smoke testing	Regressio n testing
Unit testing												
Integration testing												
System testing												
Acceptance testing												
Functional testing												
Non-functional testing												
Black-box testing												
White-box testing												
Developer testing												
QA team testing												
Smoke testing												
Regression testing												

Quiz: Mark Related Concepts

And explain why!

	Unit testing	Integration testing	System testing	Acceptance testing	Functional testing	Non-functional testing	Black-box testing	White-box testing	Developer testing	QA team testing	Smoke testing	Regression testing
Unit testing							●	●	●			●
Integration testing					●	●	●	●	●	●		●
System testing					●	●	●			●	●	●
Acceptance testing					●	●	●					
Functional testing							●	●	●	●	●	●
Non-functional testing							●	●	●	●		●
Black-box testing									●	●	●	●
White-box testing									●			●
Developer testing											●	●
QA team testing											●	●
Smoke testing												●
Regression testing												

Thank you!