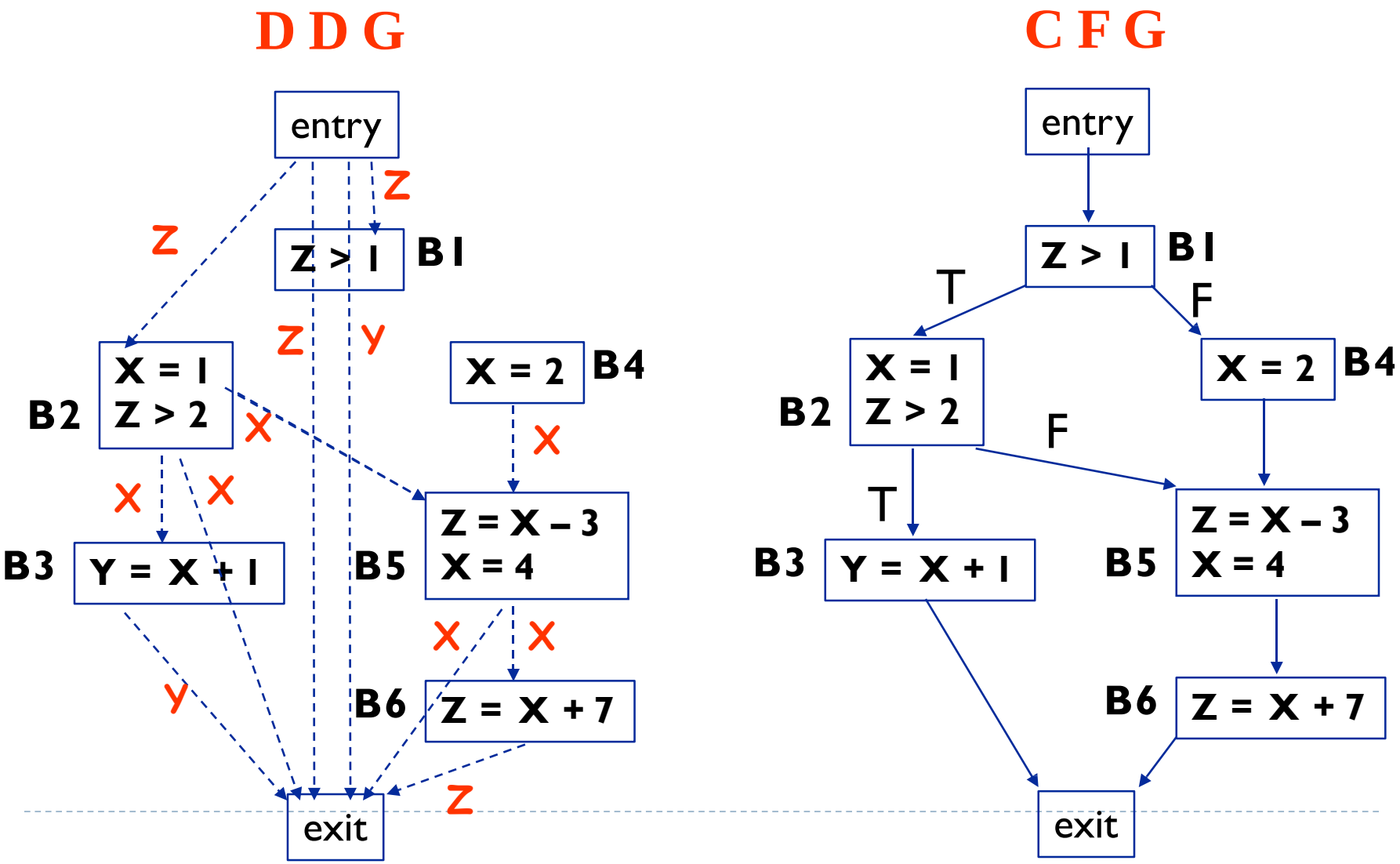
Software Verification & Validation

This slide deck uses some content from multiple sources. Credit goes to the original authors and Alex ☺

Program Dependence Graph



Static vs Dynamic Analysis

- Static Analysis
  - operates on a model of the SW (w/o executing it)
  - Can produce definitive information that holds for all inputs
- Dynamic Analysis
  - Operates on dynamic information collected by running the SW
  - Produces "sampling information" that holds for the inputs considered
- Combined static and dynamic analysis
  - Leverage complementary strategies

Software Testing

General Concepts

Software is Buggy

- On average, 1-5 errors per 1KLOC
- Windows 2000 - 35M LOC
- 63,000 known bugs at the time of release
- 2 per 1,000lines

- For mass market software 100% correct is infeasible, but we must verify the SW as much as possible

Failure, Fault, Error

Failure

Observable incorrect behavior of a program.

Conceptually related to the behavior of the program, rather than its code.

Fault (bug)

Related to the code. Necessary (not sufficient!) condition for the occurrence of a failure.

Error

Cause of a fault. Usually a human error (conceptual, typo, etc.)

Approaches to Verification

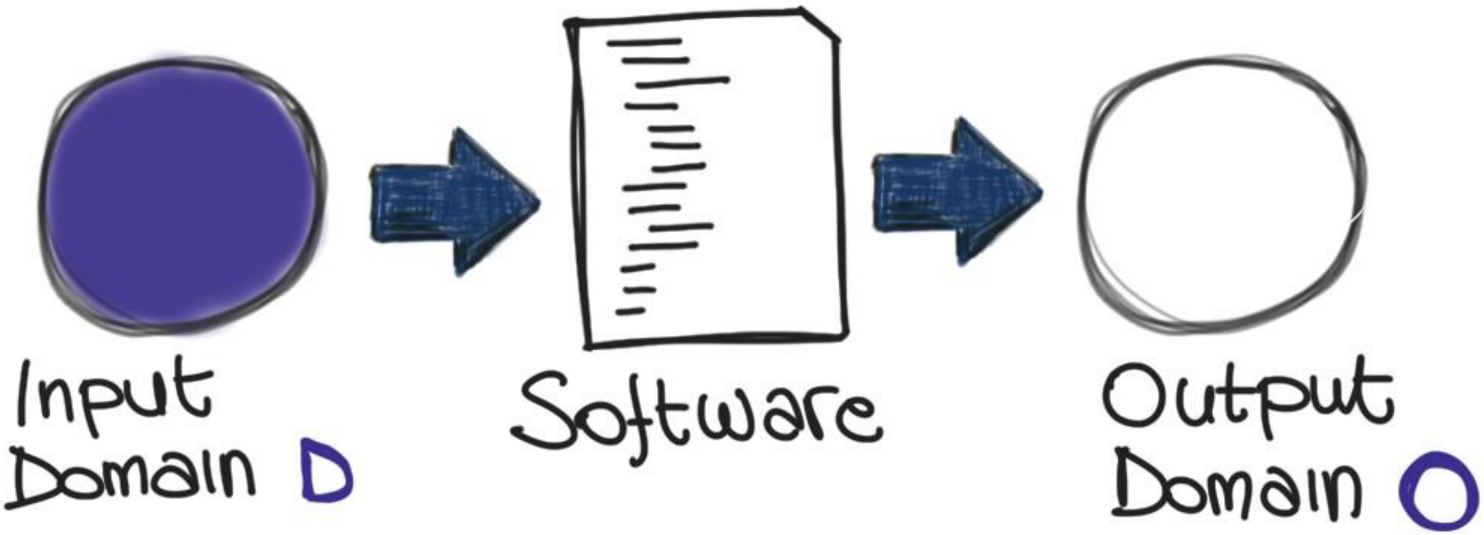
- *Testing: exercising software to try and generate failures*
- *Static verification: identify (specific) problems statically, that is, considering all possible executions*
- *Inspection/review/walkthrough: systematic group review of program text to detect faults*
- *Formal proof: proving that the program text implements the program specification*

TESTING

Test case : $\{i \in D, o \in O\}$

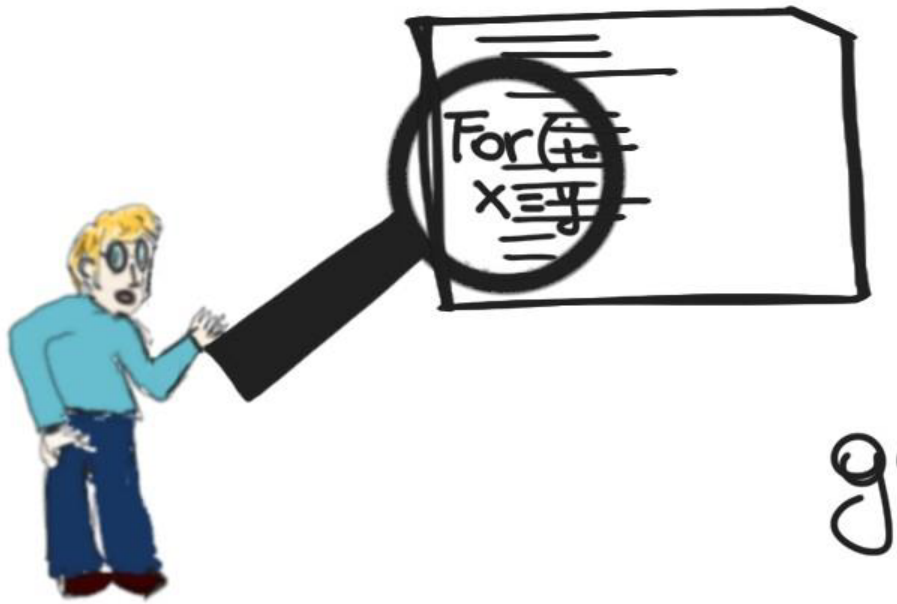
Test suite : set of test cases

VERIFICATION



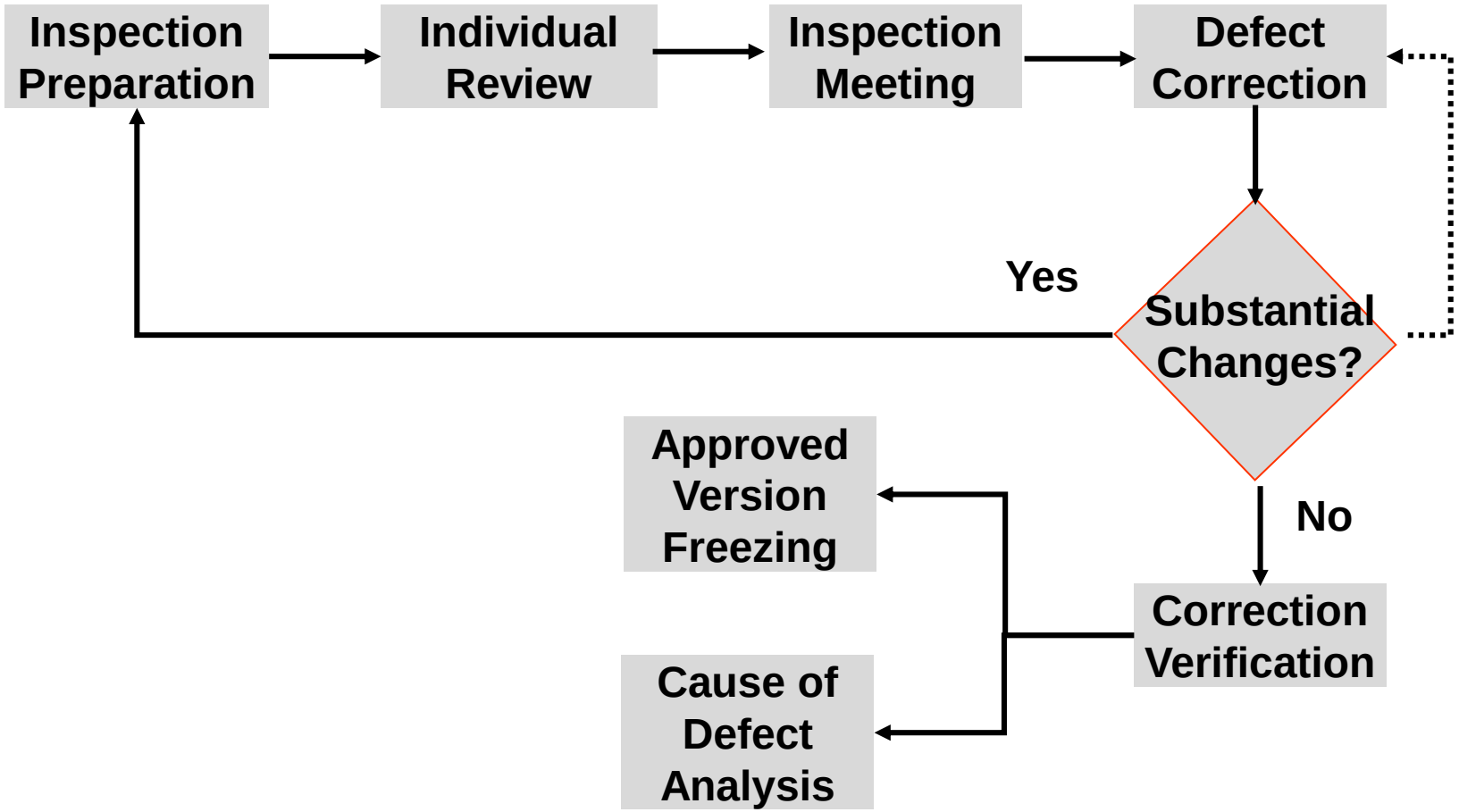
Considers all possible inputs (executions)

INSPECTIONS (AKA
- reviews
- walkthroughs)



Manual
group activity

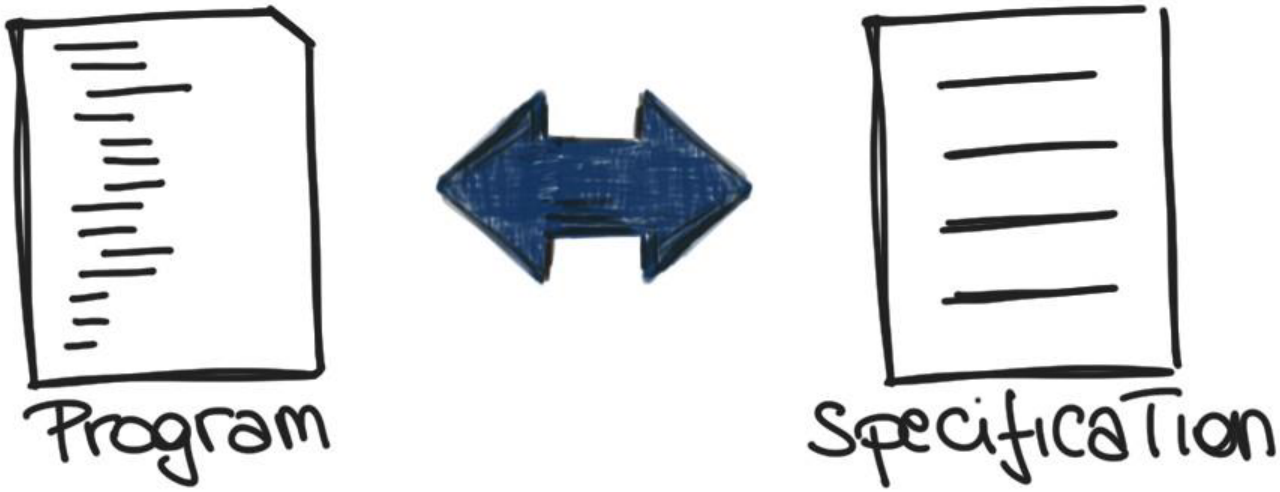
Inspection Phases



Defect Prevention

- In addition to removing defects through inspections, we can eliminate defects using
  - Checklists: common mistakes, concerns to address, activities to do
  - Templates: standard document formats that list the different aspects to be covered
    - Reduce work and avoid incompleteness
  - Tools and workflow automation
    - Avoid errors, inconsistencies and missing steps
    - Reduce effort too!

Formal Proof (of correctness)



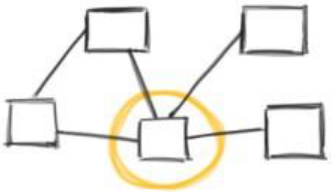
Given a formal specification, checks that the code corresponds to such specification

What is Testing?

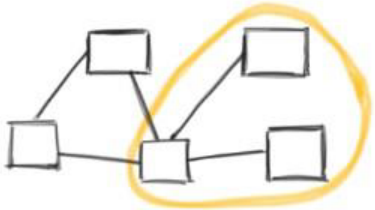
Testing -> To execute a program with a sample of the input data

- *Dynamic technique: program must be executed*
- *Optimistic approximation:*
  - *The program under test is exercised with a (very small) subset of all the possible input data*
  - *We assume that the behavior with any other input is consistent with the behavior shown for the selected subset of input data*

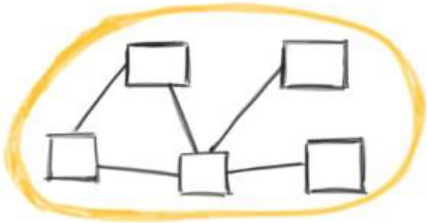
Testing Granularity Levels



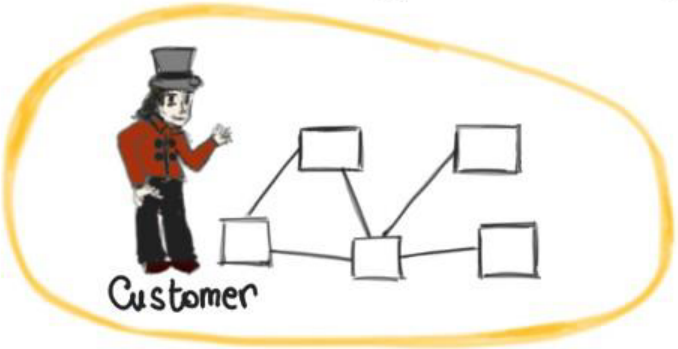
Unit Testing



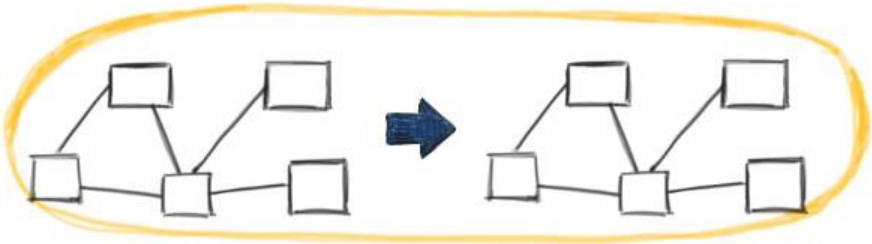
Integration Testing



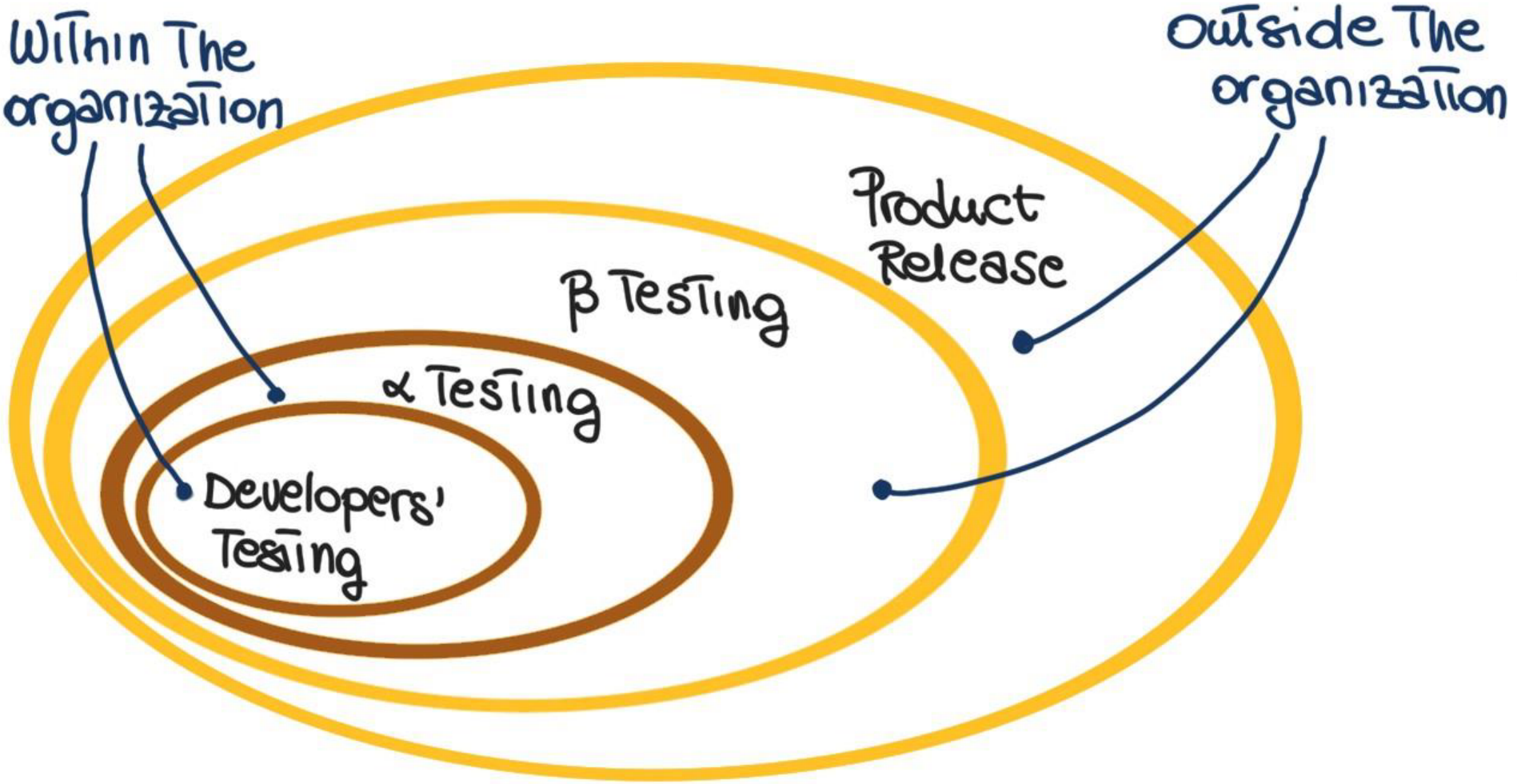
System Testing



Acceptance Testing



Regression Testing


Within The organization
Outside The organization
Product Release
β Testing
α Testing
Developers' Testing

Functional vs. Structural Testing

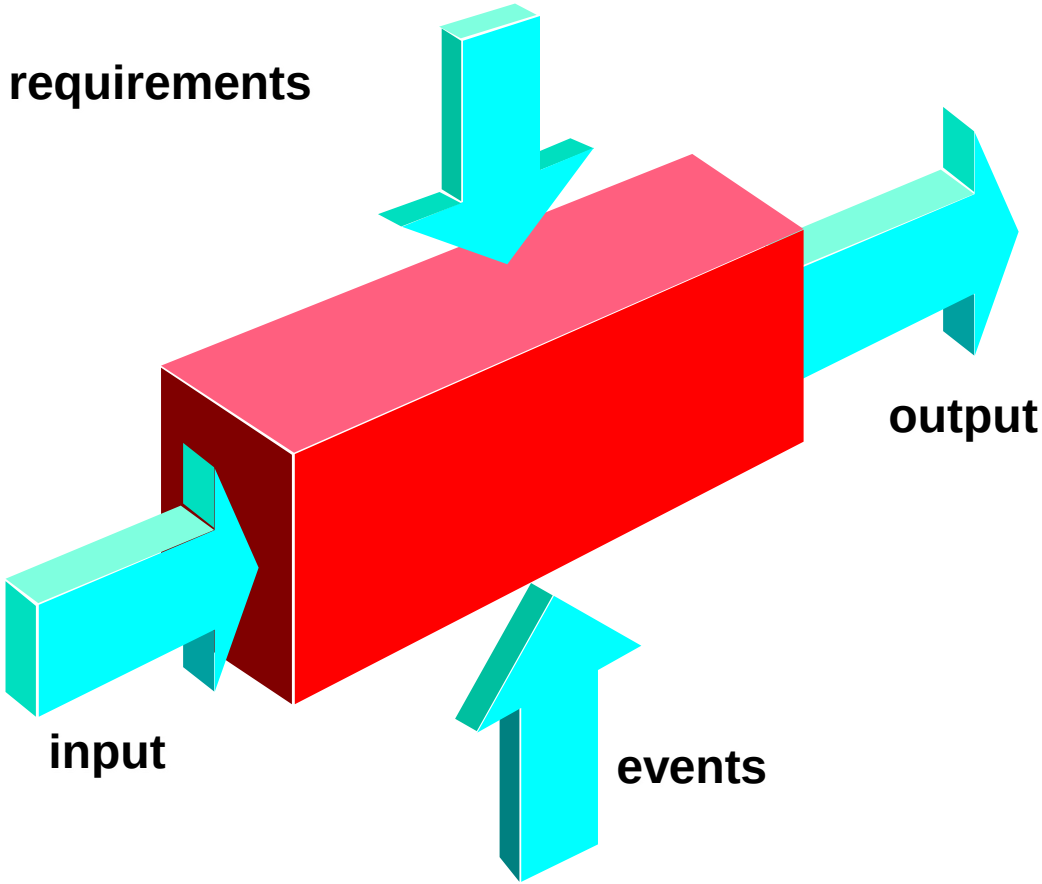
BLACK-BOX TESTING

- based on a description of the software (specification)
- cover as much specified behavior as possible
- cannot reveal errors due to implementation details

WHITE-BOX TESTING

- based on the code
- cover as much coded behavior as possible
- cannot reveal errors due to missing paths

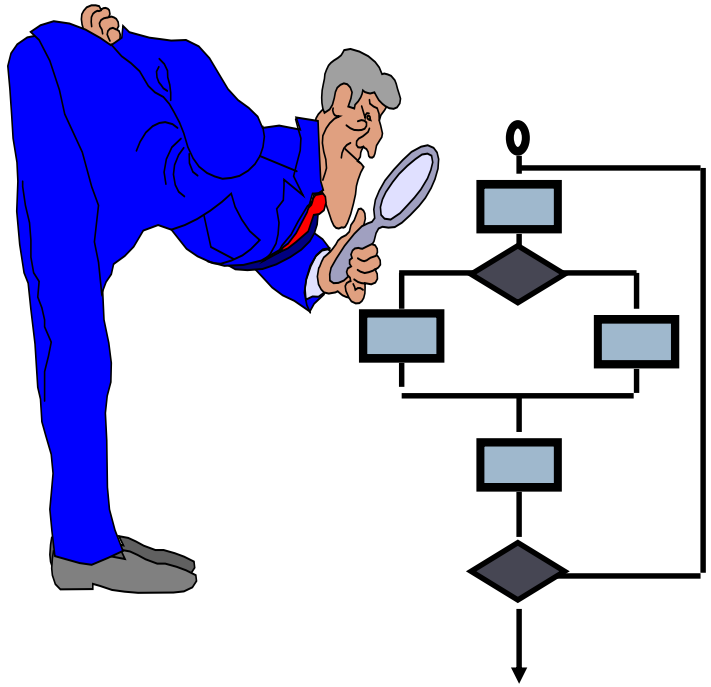
Black-Box Testing



BLACK-BOX TESTING EXAMPLE

Specification: inputs an integer and prints it

White-Box Testing



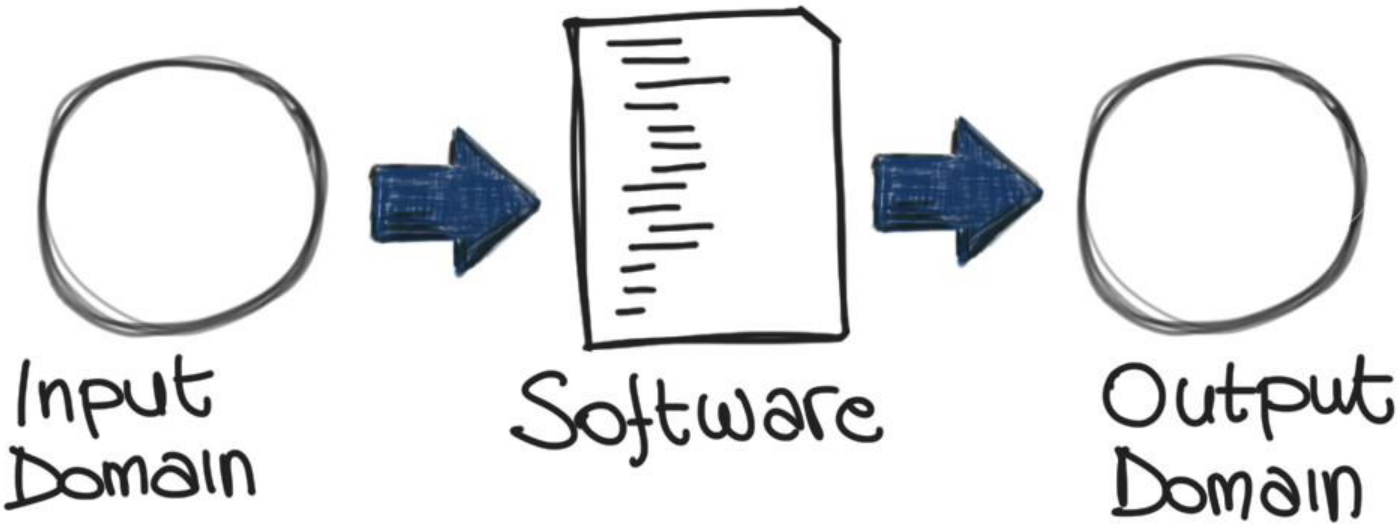
... our goal is to ensure that all statements and conditions have been executed at least once ...

White-box testing

- Also called 'glass-box' or 'structural' testing
- Testers have access to the system design
  - They can
    - Examine the design documents
    - View the code
    - Observe at run time the steps taken by algorithms and their internal data
  - Individual programmers often informally employ glass-box testing to verify their own code

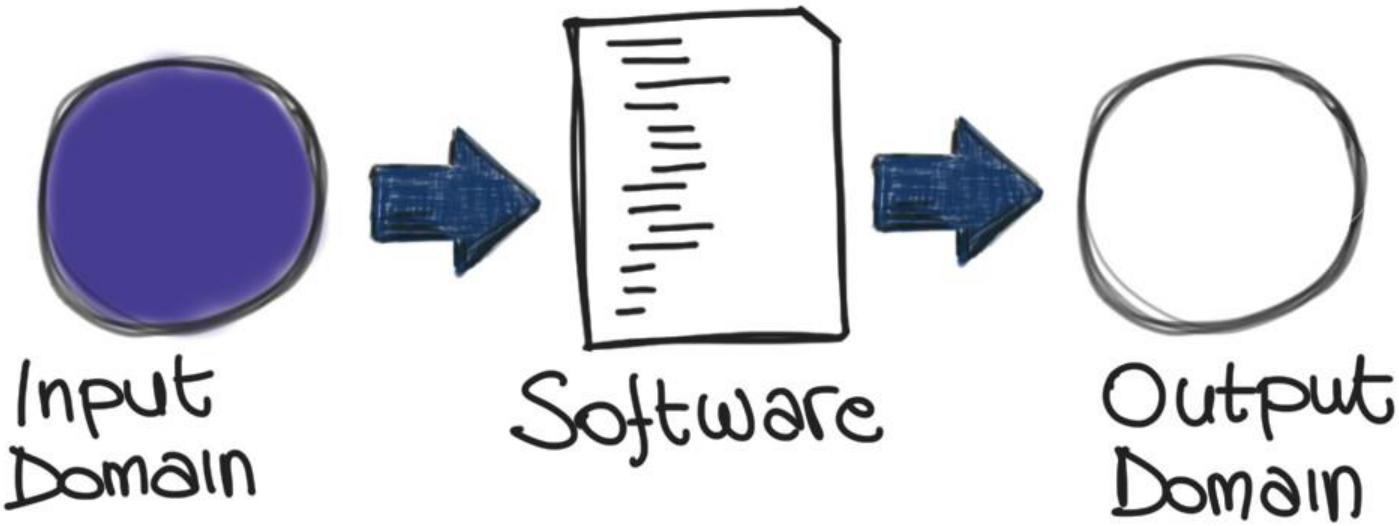
Test-Data Selection

TEST DATA SELECTION



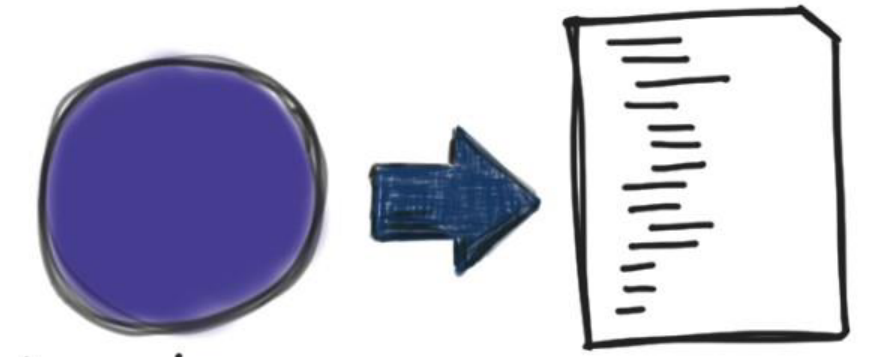
STRAW-MAN IDEA : EXHAUSTIVE TESTING !

STRAW-MAN IDEA: EXHAUSTIVE TESTING!



Considers all possible inputs (executions)

STRAW-MAN IDEA: EXHAUSTIVE TESTING!



Input Domain

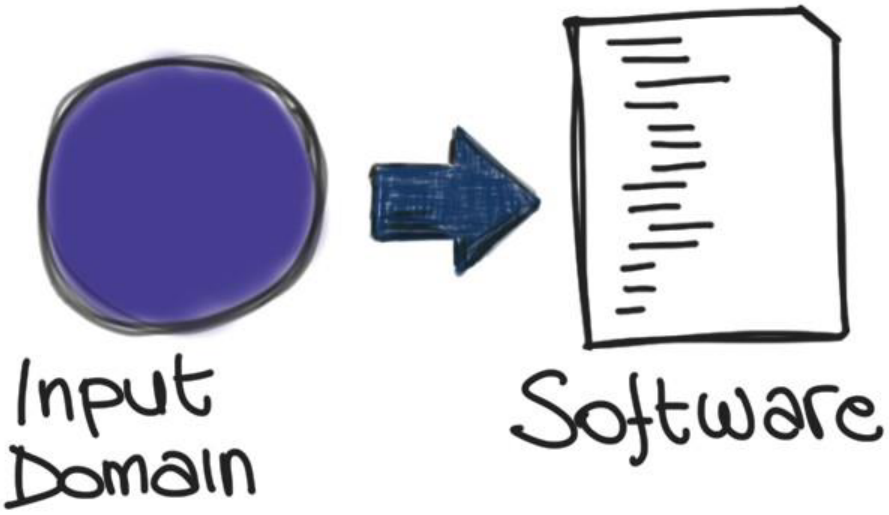
Software

How long would it take to exhaustively test the function

`printSum(int a, int b)`?

[ ]

STRAW-MAN IDEA: EXHAUSTIVE TESTING!



How long would it take to exhaustively test the function

printSum(int a, int b) ?

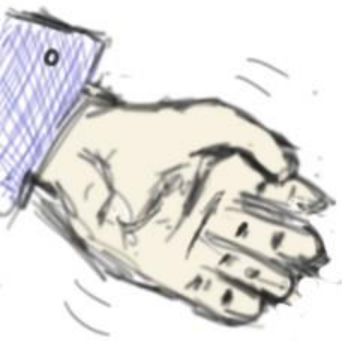
$2^{32} \times 2^{32} = 2^{64} \cong 10^{19}$ tests

1 test per nanosecond (10^9 tests/sec)

=> 10^{10} seconds

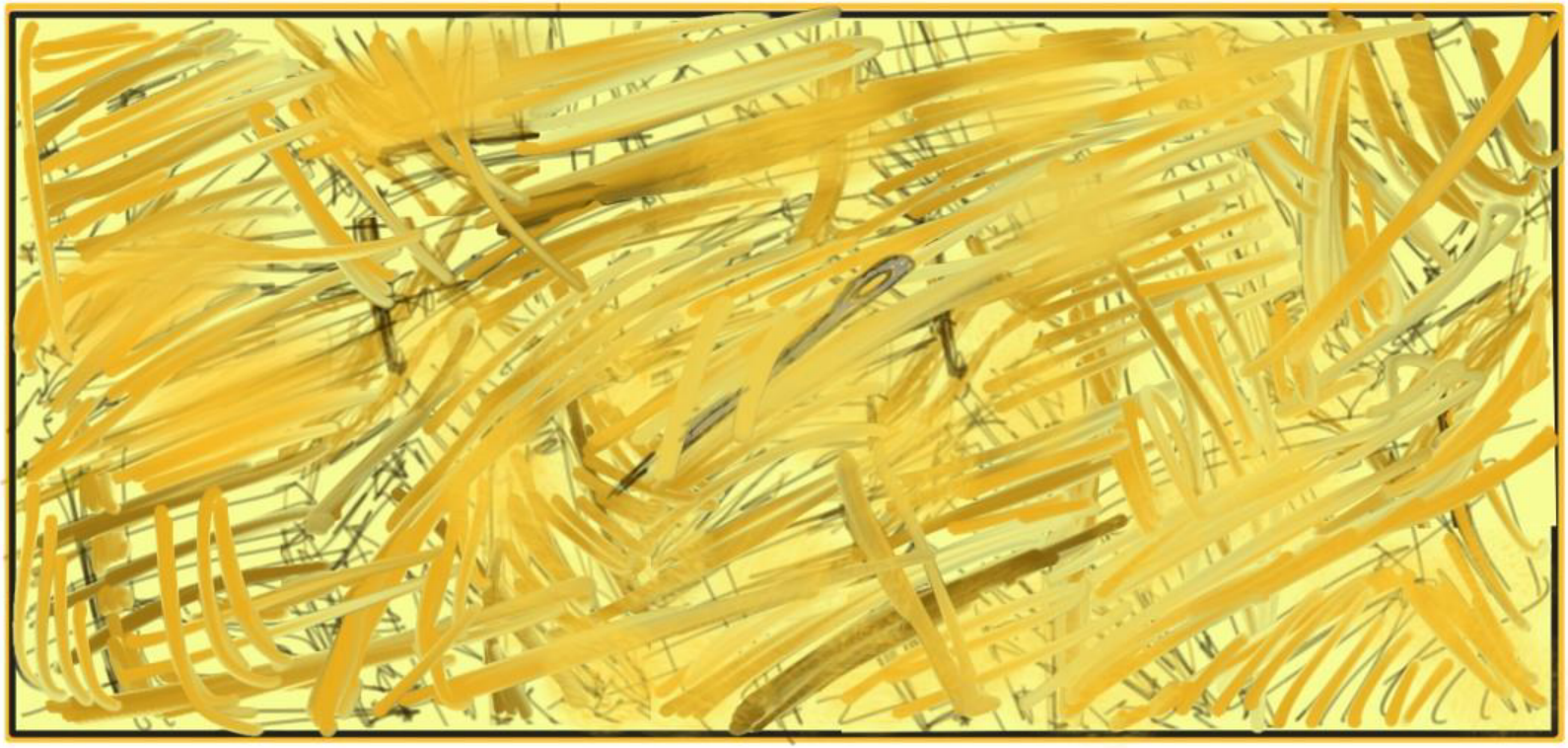
[~600 years]

RANDOM TESTING

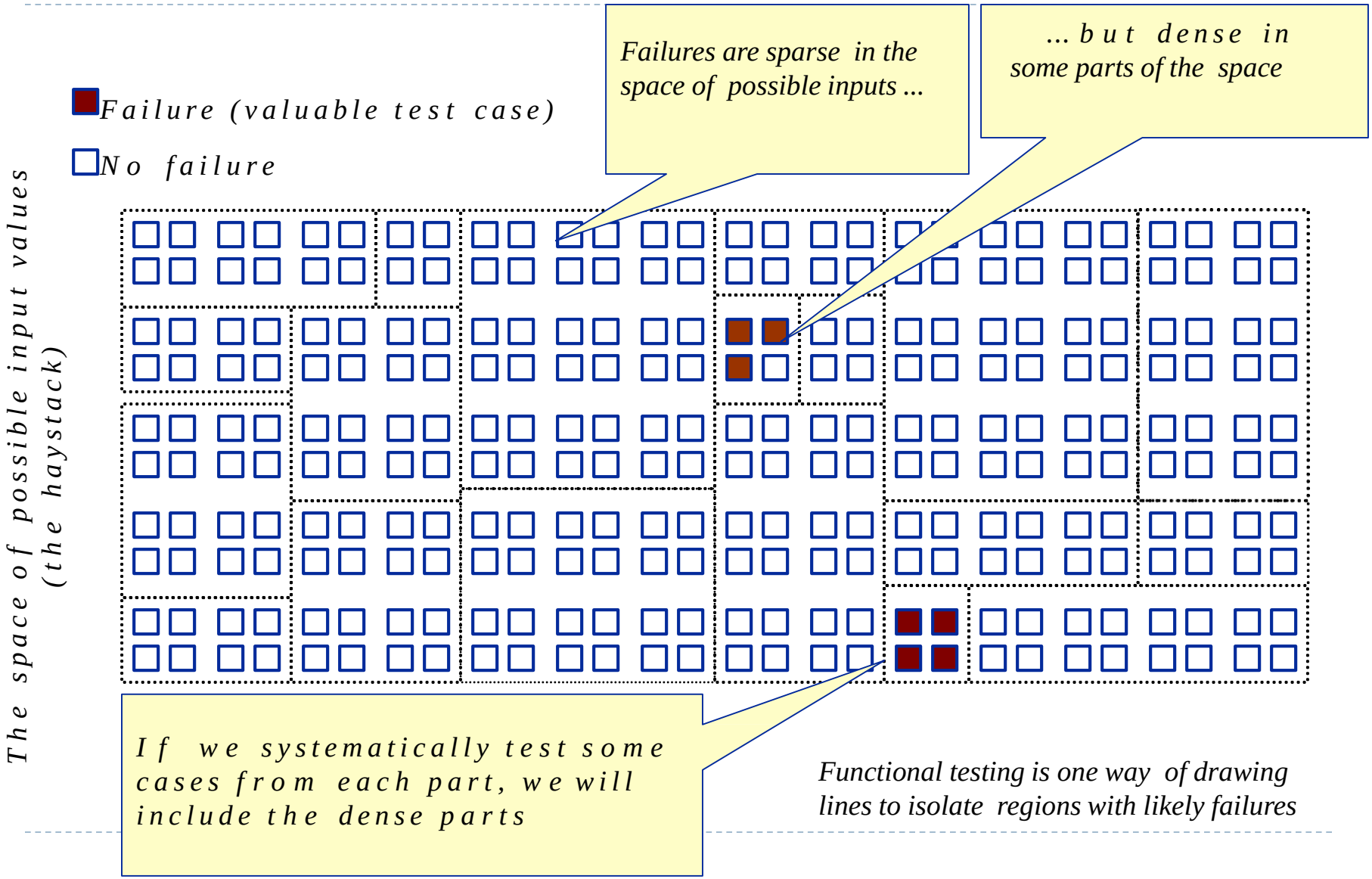


- pick inputs uniformly
- all inputs considered equal
- no designer bias

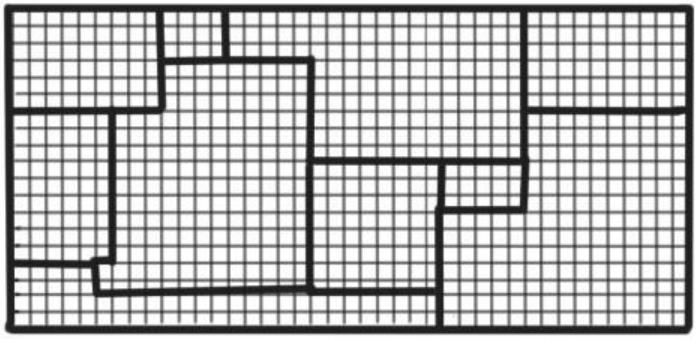
SO WHY NOT RANDOM ?



Systematic Partition Testing



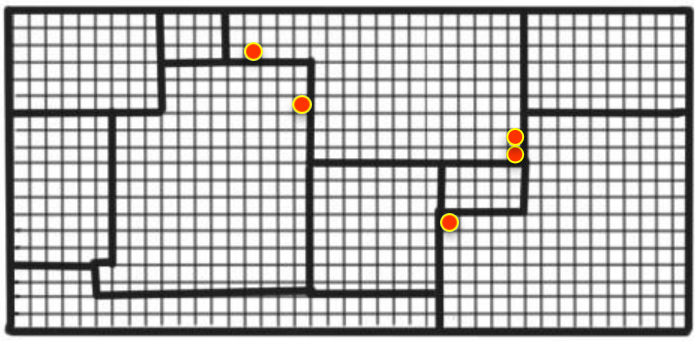
BOUNDARY VALUES



Basic idea

Errors tend to occur at the boundary of a (sub) domain

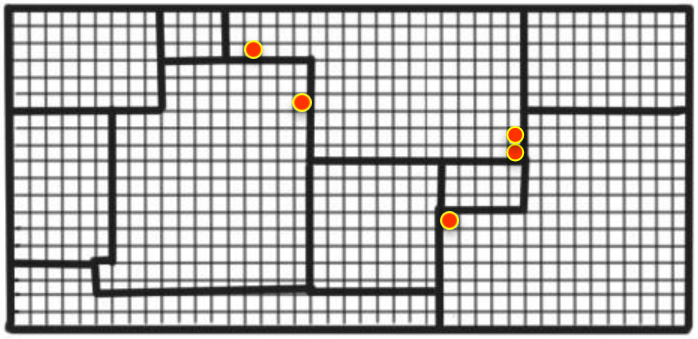
BOUNDARY VALUES



Basic idea

Errors tend to occur at the boundary of a (sub) domain

BOUNDARY VALUES



Basic idea

Errors tend to occur at the boundary of a (sub) domain

⇒ Select inputs at these boundaries

Equivalence classes

- It is inappropriate to test by *brute force*, using *every possible* input value
  - Takes a huge amount of time
  - Is impractical
  - Is pointless!

- You should divide the possible inputs into groups which you believe will be treated similarly by all algorithms.
  - Such groups are called *equivalence classes*.
  - A tester needs only to run one test per equivalence class
  - The tester has to
    - understand the required input,
    - appreciate how the software may have been designed

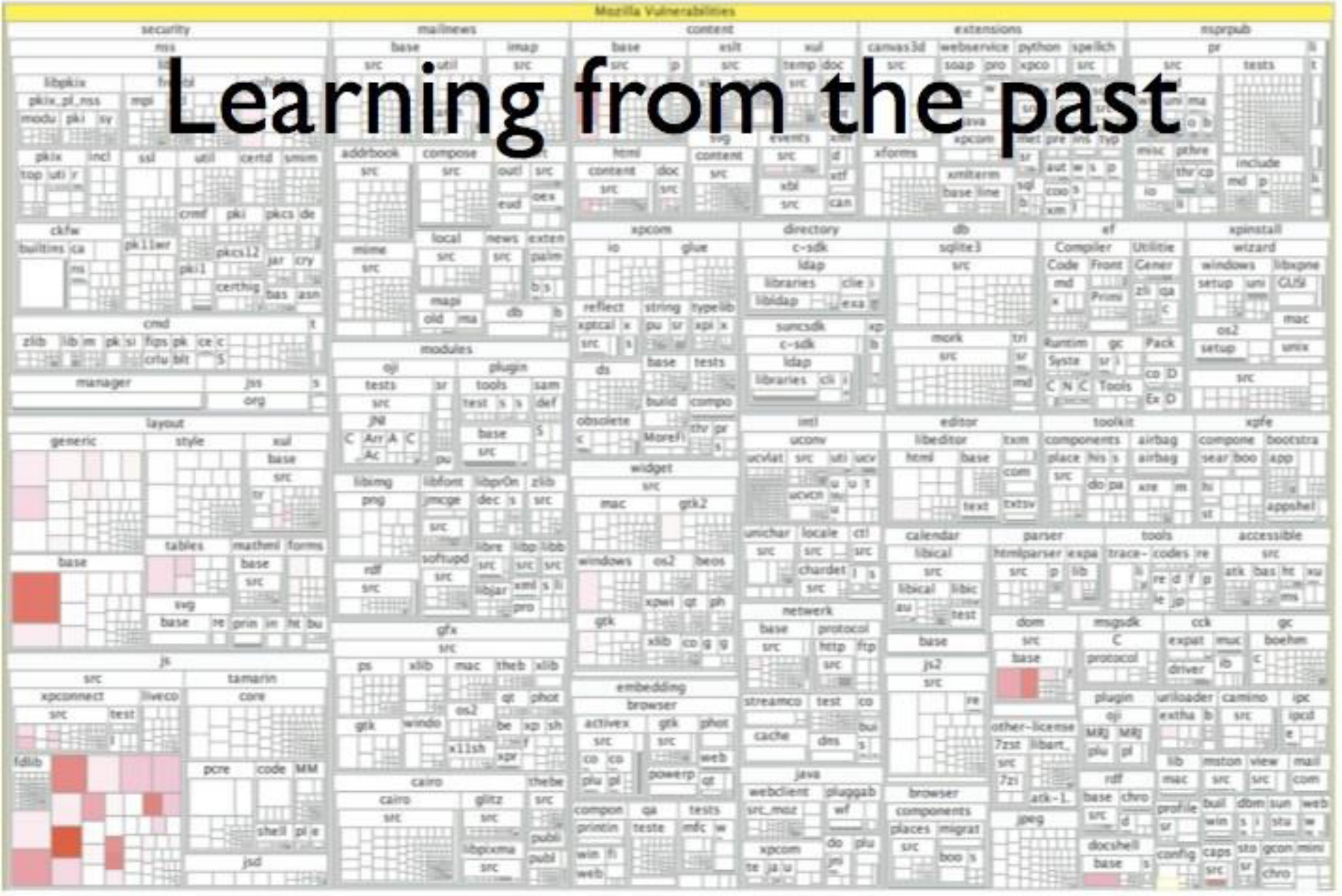
Valid input is a month number (1-12)

- Equivalence classes are: [-∞..0], [1..12], [13.. ∞]

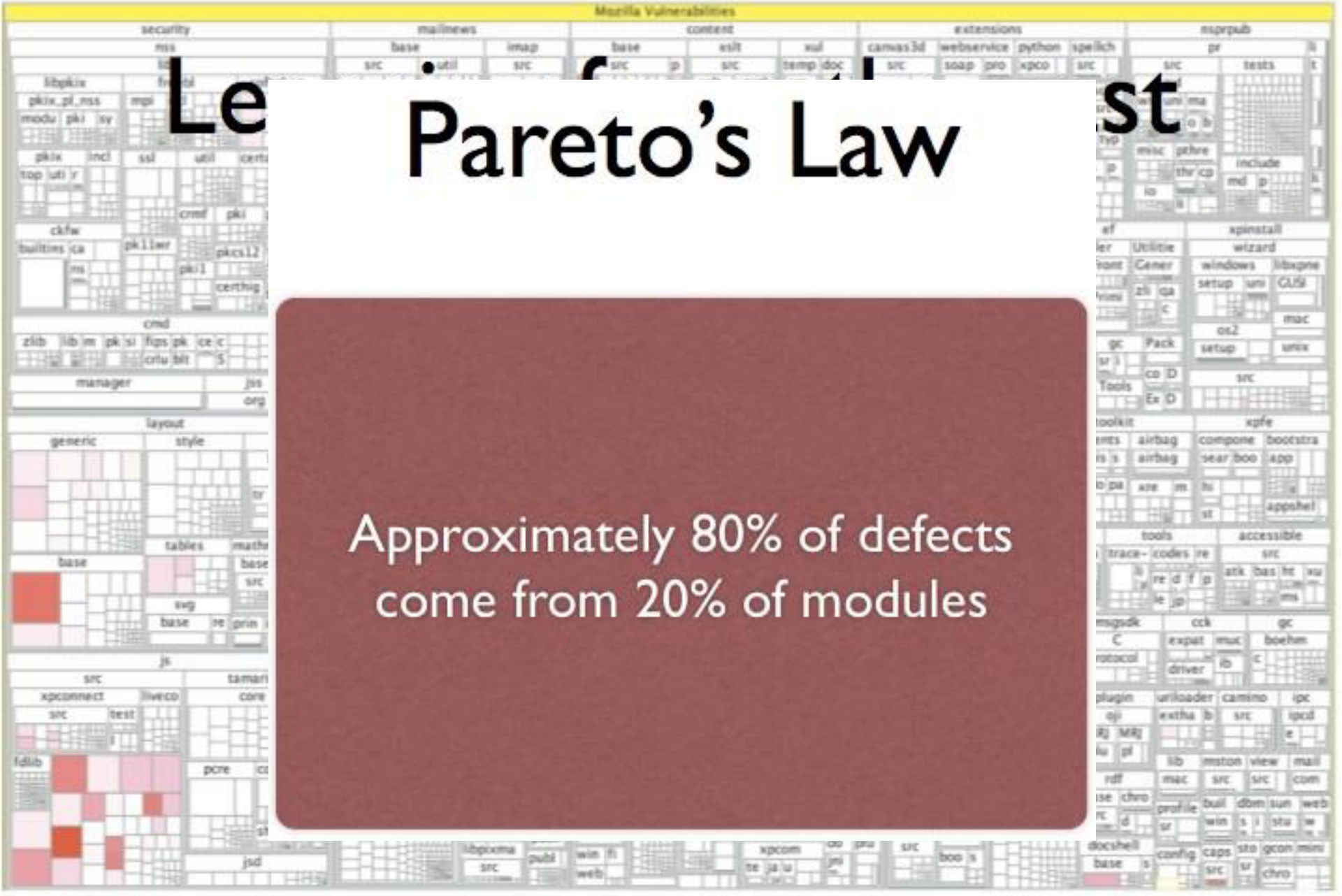
Combinations of equivalence classes

- Combinatorial explosion means that you cannot realistically test every possible system-wide equivalence class.
  - If there are 4 inputs with 5 possible values there are 5^4 (i.e.625) possible system-wide equivalence classes.
- You should first make sure that at least one test is run with every equivalence class of every individual input.
- You should also test all combinations where an input is likely to *affect the interpretation* of another.
- You should test a few other random combinations of equivalence classes.

Historical models



Historical models



Coverage

- Function coverage: Each function/method executed by at least one test case
- Statement coverage: Each line of code covered by at least one test case (need more test cases than above)
- Path coverage: Every possible path through code covered by at least one test case (need lots of test cases)

Levels of Coverage

- Level 1: 100% statement coverage
- Level 2: 100% decision coverage or branch coverage
- Level 3: 100% condition coverage
- Level 4: 100% decision/condition coverage
- Level 5: 100% multiple condition coverage
- Level 6: Limited path coverage
- Level 7: 100% path coverage

COVERAGE CRITERIA

Defined in terms of

- test requirements

Result in

- test specifications
- test cases

STATEMENT COVERAGE

Test requirements

statements in the program

Coverage measure

$$\frac{\text{number of executed statements}}{\text{total number of statements}}$$

BRANCH COVERAGE

Test requirements

branches in the program

Coverage measure

$$\frac{\text{number of executed branches}}{\text{total number of branches}}$$

CONDITION COVERAGE

Test requirements

individual conditions in the program

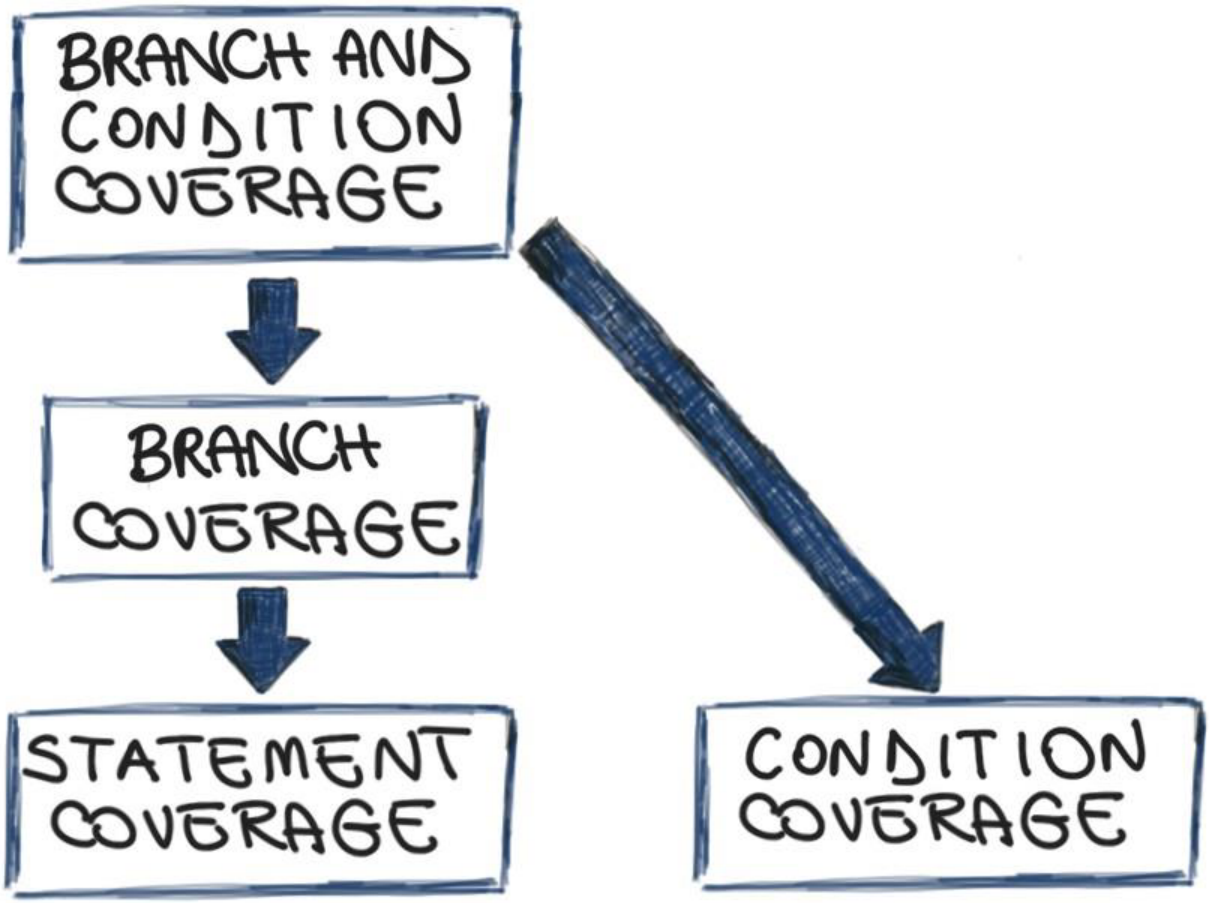
Coverage measure

$$\frac{\text{number of conditions that are both T and F}}{\text{total number of conditions}}$$

BRANCH AND CONDITION COVERAGE (DECISION)

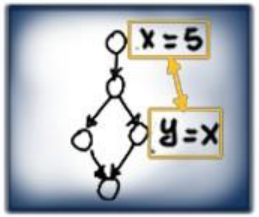
Test requirements	branches and individual conditions in the program
Coverage measure	Computed considering both coverage measures

TEST CRITERIA SUBSUMPTION

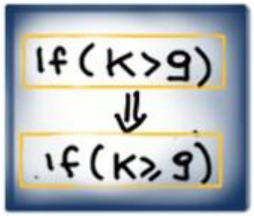


OTHER CRITERIA

Path coverage



Data-flow coverage

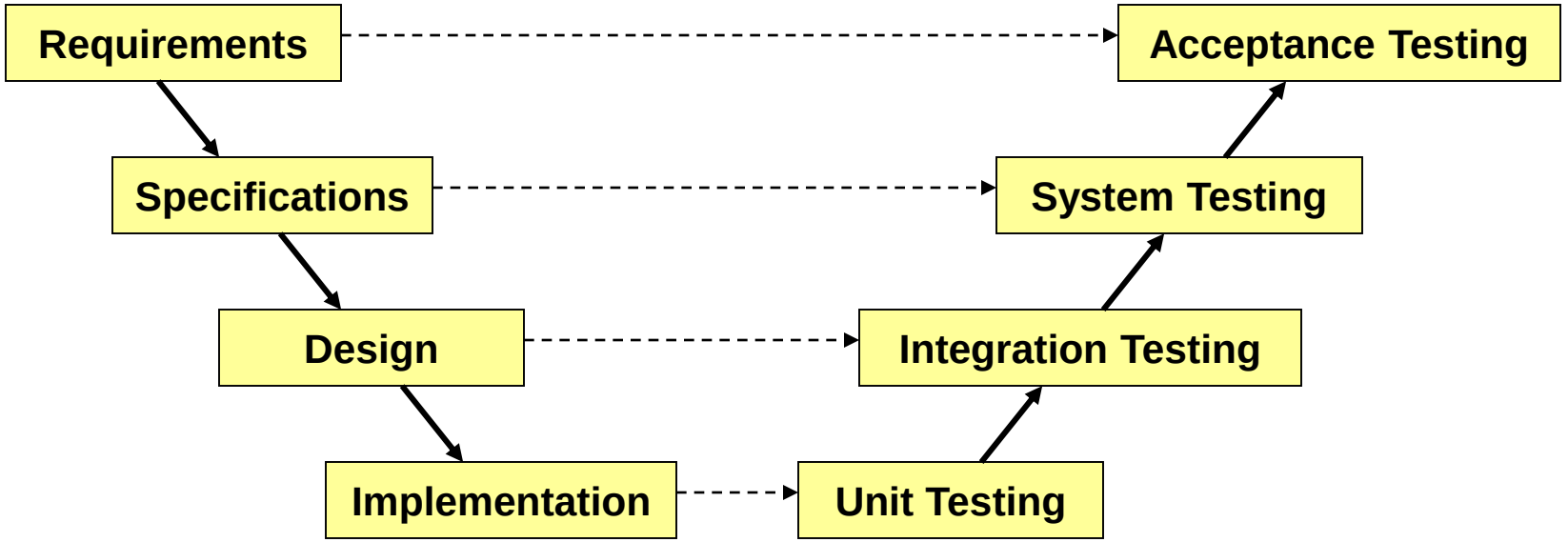


Mutation coverage

Software Testing Strategies

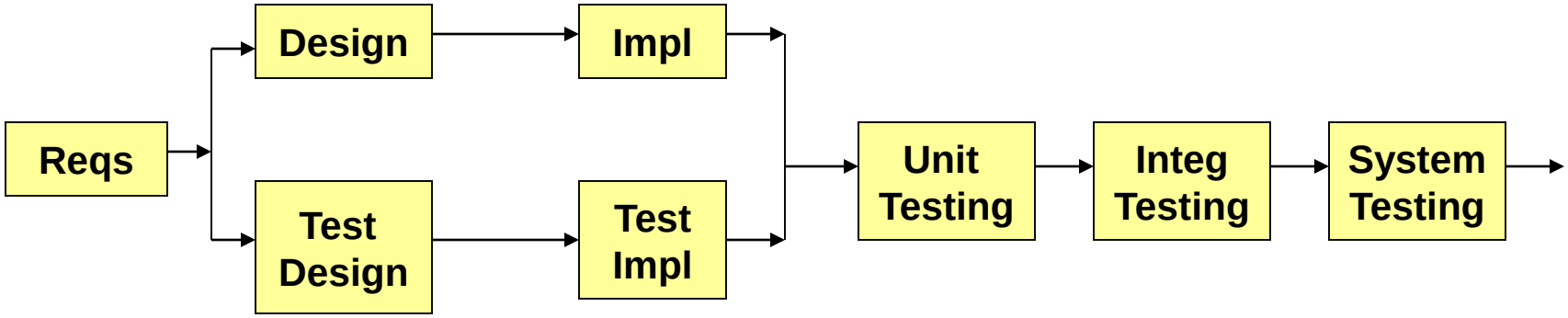
Some Material adapted from Lethbridge & Laganiere;
Some Material adapted from Pressman.

Testing phases: V model



A lifecycle view that shows relationships between development and test phases

Concurrent Test Development



Testing is done after development, but test design & impl. often happens concurrently with development

Testing Phases

- Unit Testing
  - Developer tests individual modules
- Integration testing
  - Put modules together, try to get them working together
  - Integration testing is complete when the different pieces are able to work together
- System testing
  - Black-box testing of entire deliverable against specs
- Acceptance testing
  - Testing against user needs, often by the user

Unit testing

- During unit testing, modules are tested in isolation:
  - If all modules were to be tested together:
    - it may not be easy to determine which module has the error.
- Unit testing reduces debugging effort several folds.
  - Programmers carry out unit testing immediately after they complete the coding of a module.

Role of Unit Testing

- Assure minimum quality of units before integration into system
- Focus attention on relatively small units
- Testing forces us to read our own code – spend more time reading than writing
- Automated tests support maintainability and extendibility
- Marks end of development step

Integration testing

- After different modules of a system have been coded and unit tested:
  - modules are integrated in steps according to an integration plan
  - partially integrated system is tested at each integration step.

Objectives:

- Gain confidence in the integrity of overall system design
- Ensure proper interaction of components
- Run simple system-level tests

Integration Testing Strategies

- Big-bang
- Top-down
- Bottom-up
- Critical-first
- Function-at-a-time
- As-delivered
- Sandwich

Big Bang Integration Testing

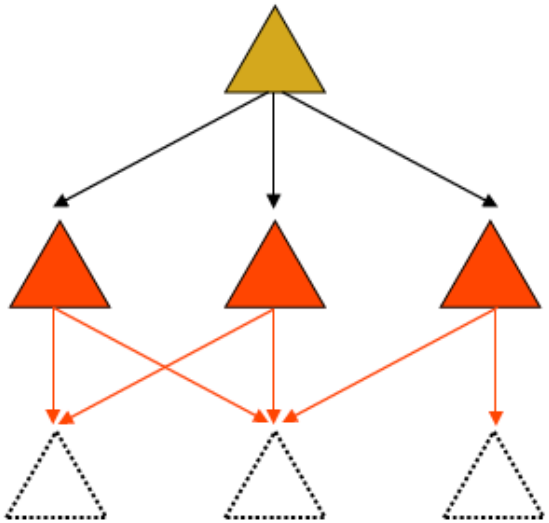
- Big bang approach is the simplest integration testing approach:
  - all the modules are simply put together and tested.
  - this technique is used only for very small systems.

Issues:

- avoids cost of scaffolding (stubs or drivers)
- does not provide any locality for finding faults

Top-down integration testing

- Start with top-level modules
- Use stubs for lower-level modules
- As each level is completed, replace stubs with next level of modules



Pros:

- Always have a top-level system
- Stubs can be written from interface specifications

Cons:

- May delay performance problems until too late
- Stubs can be expensive

Bottom-up Integration Testing

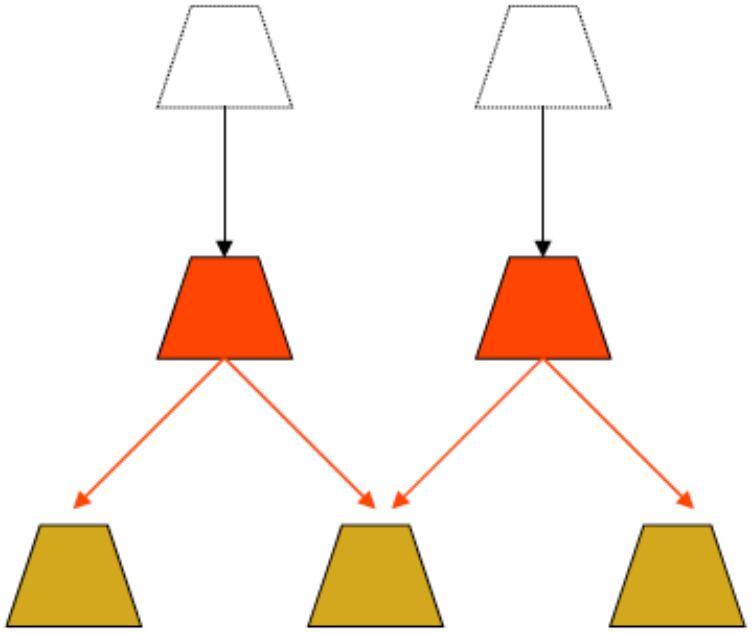
- Start with bottom-level modules
- Use drivers for upper-level modules
- As each level is completed, replace drivers with next level of modules

Pros:

- Primitive functions get most testing
- Drivers are usually cheap

Cons:

- Only have a complete system at the end



Critical-first Integration

- Integrate the most critical components first
- Add the remaining pieces later

Issues:

- Guarantees that the most important components work
- May be difficult to integrate

Function-at-a-time Integration

- Integrate all modules needed to perform a particular function
- For each function, add another set of modules

Issues

- Makes for easier test generation
- May postpone function interaction for too long
  - Dependencies may create a problem

As-Delivered Integration

- Integrate the modules as and when they become available

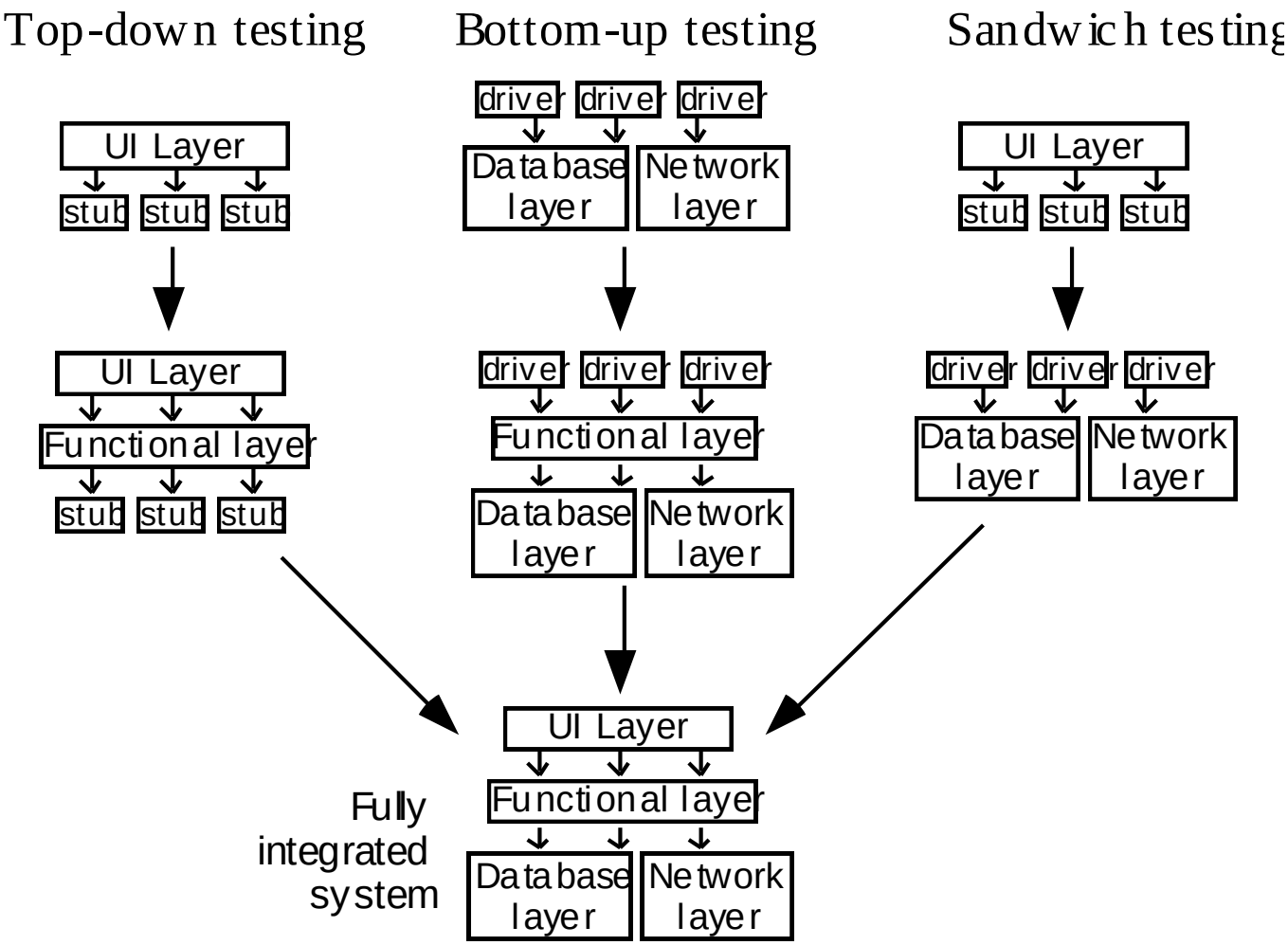
Issues

- Efficient – Just-in-time Integration
- Lazy – may lead to missed schedules

Sandwich Integration Testing

- Mixed (or sandwiched) integration testing:
  - uses both top-down and bottom-up testing approaches.
  - Most common approach

Example of different integration strategies



System Testing

Objectives

- Gain confidence in the integrity of the system as a whole
  - Ensure compliance with functional requirements
  - Ensure compliance with performance requirements

Testing Functional Requirements

1. Prepare a test plan from the functional specification of the system
2. Prepare tests for all areas of functionality
3. Review test plan and tests
4. Execute tests
5. Monitor fault rate

Testing Performance Requirements

1. Identify stress points of system
2. Create or obtain load generators
   - might use existing system
   - might buy/make special purpose tools
3. Run stress tests
4. Monitor system performance
   - usually needs instrumentation

Acceptance Testing

- Testing performed by the customer or end-user himself:
  - to determine whether the system should be accepted or rejected.

Other paths to acceptance:

- Beta testing
  - Distribute system to volunteers
  - Collect change requests, fix, redistribute
  - Collect statistics on beta use
- Shadowing
  - Collect or redistribute real-time use of existing system
  - Compare results
  - Collect statistics

The test-fix-test cycle

- When a failure occurs during testing:
  - Each failure report is entered into a failure tracking system.
  - It is then screened and assigned a priority.
  - Low-priority failures might be put on a *known bugs list* that is included with the software's *release notes*.
  - Some failure reports might be merged if they appear to result from the same defects.
  - Somebody is assigned to investigate a failure.
  - That person tracks down the defect and fixes it.
  - Finally a new version of the system is created, ready to be tested again.

Deciding when to stop testing

- All of the level 1 ("critical") test cases must have been successfully executed.
- Certain pre-defined percentages of level 2 and level 3 test cases must have been executed successfully.
- The targets must have been achieved and are maintained for at least two cycles of 'builds'.
  - A *build* involves compiling and integrating all the components.
  - Failure rates can fluctuate from build to build as:
    - Different sets of regression tests are run.
    - New defects are introduced.

The roles of people involved in testing

- The first pass of unit and integration testing is called *developer testing*.
  - Preliminary testing performed by the software developers who do the design.
- *Independent testing* may be performed by separate group.
  - They do not have a vested interest in seeing as many test cases pass as possible.
  - They develop specific expertise in how to do good testing, and how to use testing tools.

Test planning

- Decide on overall test strategy
  - What type of integration
  - Whether to automate system tests
  - Whether there is an independent test team
- Decide on the coverage strategy for system tests
  - Compute the number of test cases needed
- Identify the test cases and implement them
  - The set of test cases constitutes a "test suite"
  - May categorize into critical, important, optional tests (level 1, 2, 3)
- Identify a subset of the tests as regression tests

Testing performed by users and clients

- *Alpha testing*
  - Performed by the user or client, but under the supervision of the software development team.
- *Beta testing*
  - Performed by the user or client in a normal work environment.
  - Recruited from the potential user population.
  - An *open beta release* is the release of low-quality software to the general population.
- *Acceptance testing*
  - Performed by users and customers.
  - However, the customers do it on their own initiative.

Inspections Vs Testing

- Both testing and inspection rely on different aspects of human intelligence.
- Testing can find defects whose consequences are obvious but which are buried in complex code.
- Inspecting can find defects that relate to maintainability or efficiency.
- The chances of mistakes are reduced if both activities are performed.

Testing or inspecting, which comes first?

- It is important to inspect software *before* extensively testing it.
- The reason for this is that inspecting allows you to quickly get rid of many defects.
- Even before developer testing

Packaging for Delivery

- Software we deliver to the user must include
  - Executable in a convenient format e.g. EXE, JAR file
  - Release notes
  - User documentation: instructions on usage
    - Tutorials, user manuals, "getting started" instructions
  - Installation instructions
- May create "installables"
  - Compressed packages e.g. zip files, tar files
  - Scripts that automate installation procedures