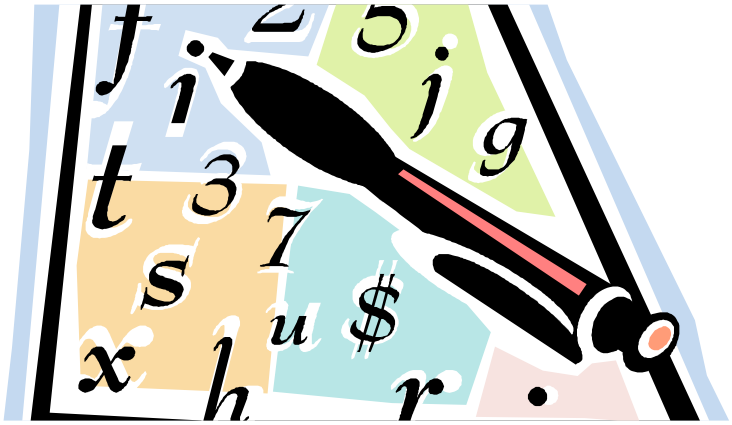
Software Engineering Theory

Lena Buffoni (including material by Kristian Sandahl/Mariam Kamkar)
Department of Computer and Information Science
2024-09-16

LINKÖPING UNIVERSITY

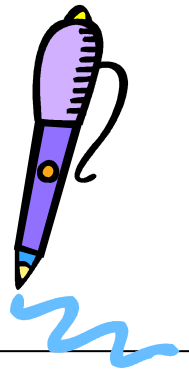
How do you test a ballpoint pen?

- Does the pen write in the right color, with the right line thickness?
- Is the logo on the pen according to company standards?
- Is it safe to chew on the pen?
- Does the click-mechanism still work after 100 000 clicks?
- Does it still write after a car has run over it?



What is expected from this pen?

Intended use!!



Verification vs Validation

Are we building the system right?

Are we building the right system?

Testing software

- Are the functions giving correct output?
- Are the integrated modules giving correct output?
- Is the entire system giving correct output when used?
- Is the correct output given in reasonable time?
- Is the output presented in an understandable way?
- Was this what we really expected?

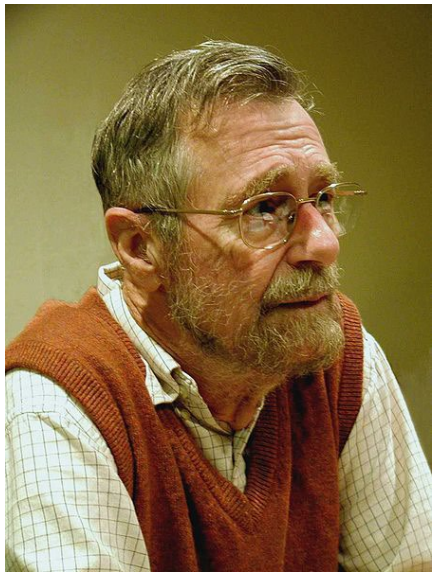
Software testing is an activity in which a program is **executed** under specified conditions, the results are **observed**, and an **evaluation** is made of the program.

Other methods for Validation & Verification

- Formal verification (Z and B methods)
- Model checking
- Prototyping
- Simulation
- Software reviews

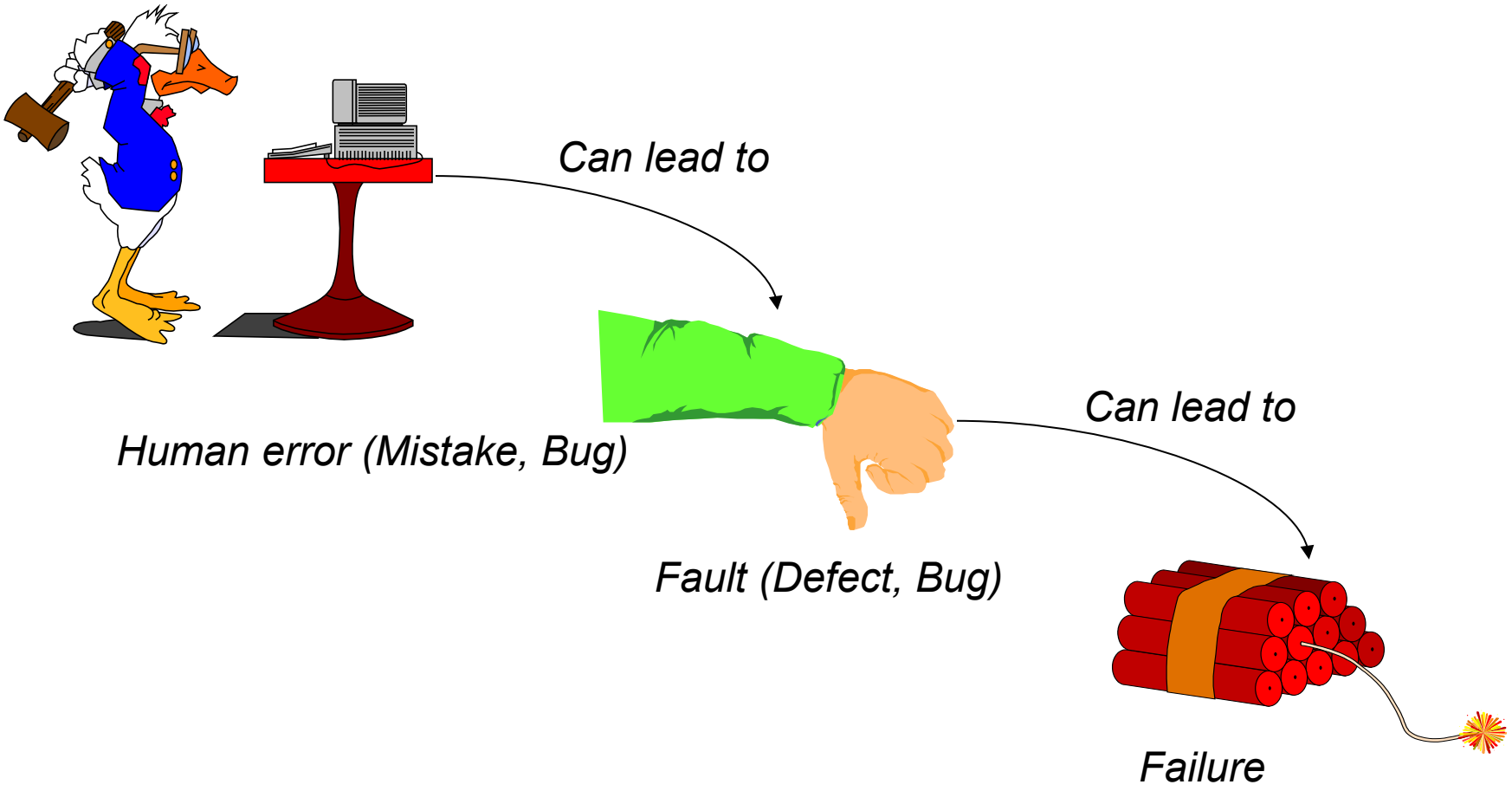
”*Testing shows the presence, not the absence of bugs*“

(Edsger Wybe Dijkstra)



…but you might use experience and statistics to make some kind of assessment.

Error, Fault, Failure



Basic Definitions – lecture notes

The terminology here is taken from standards developed by the institute of Electronics and Electrical Engineers (IEEE) computer Society.

Error: people make **errors**. A good synonym is **mistake**. When people make mistakes while coding, we call these mistakes **bugs**. Errors tend to propagate; a requirements error may be magnified during design and amplified still more during coding.

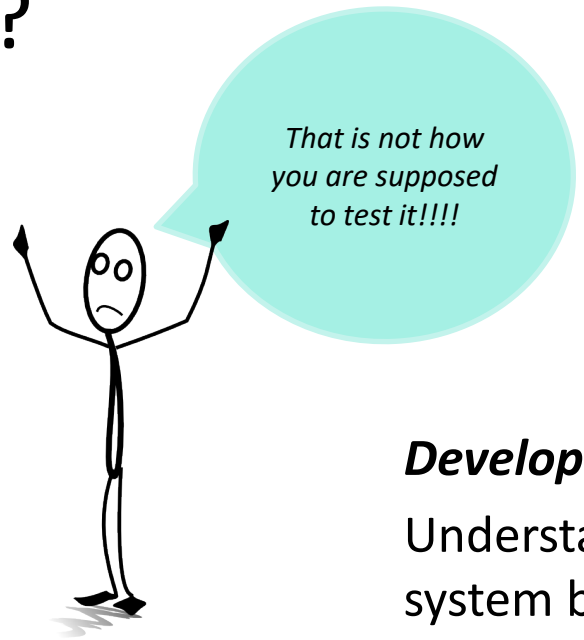
Fault: a fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, hierarchy charts, source code, and so on. **Defect** is a good synonym for fault, as is **bug**. Faults can be elusive. When a designer makes an error of omission, the resulting fault is that something is missing that should be present in the representation. We might speak of faults of commission and faults of omission. A **fault of commission** occurs when we enter something into a representation that is incorrect. **Faults of omission** occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.

Failure (anomaly): a failure occurs when a **fault executes**. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object; the second subtlety is that this definition relates failures only to faults of commission. How can we deal with failures that correspond to faults of omission?

Who does the testing?

Independent Tester

Must learn about the system, but will attempt to break it and, is driven by quality



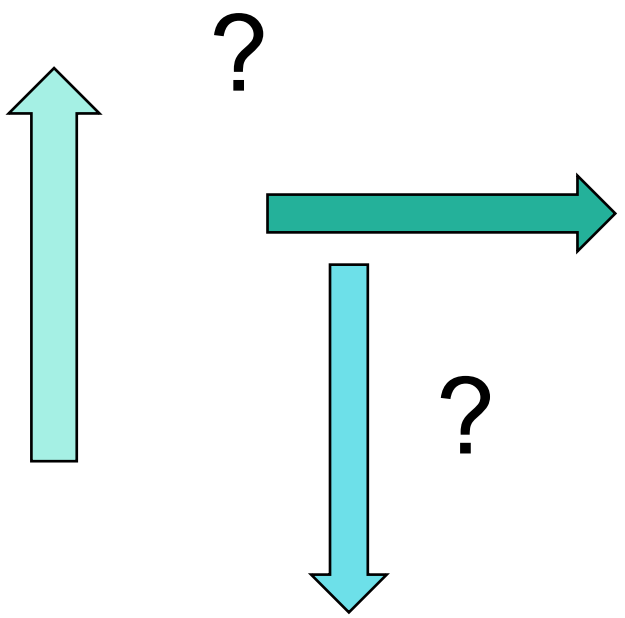
Developer

Understands the system but, will test "gently"
and is driven by "delivery”

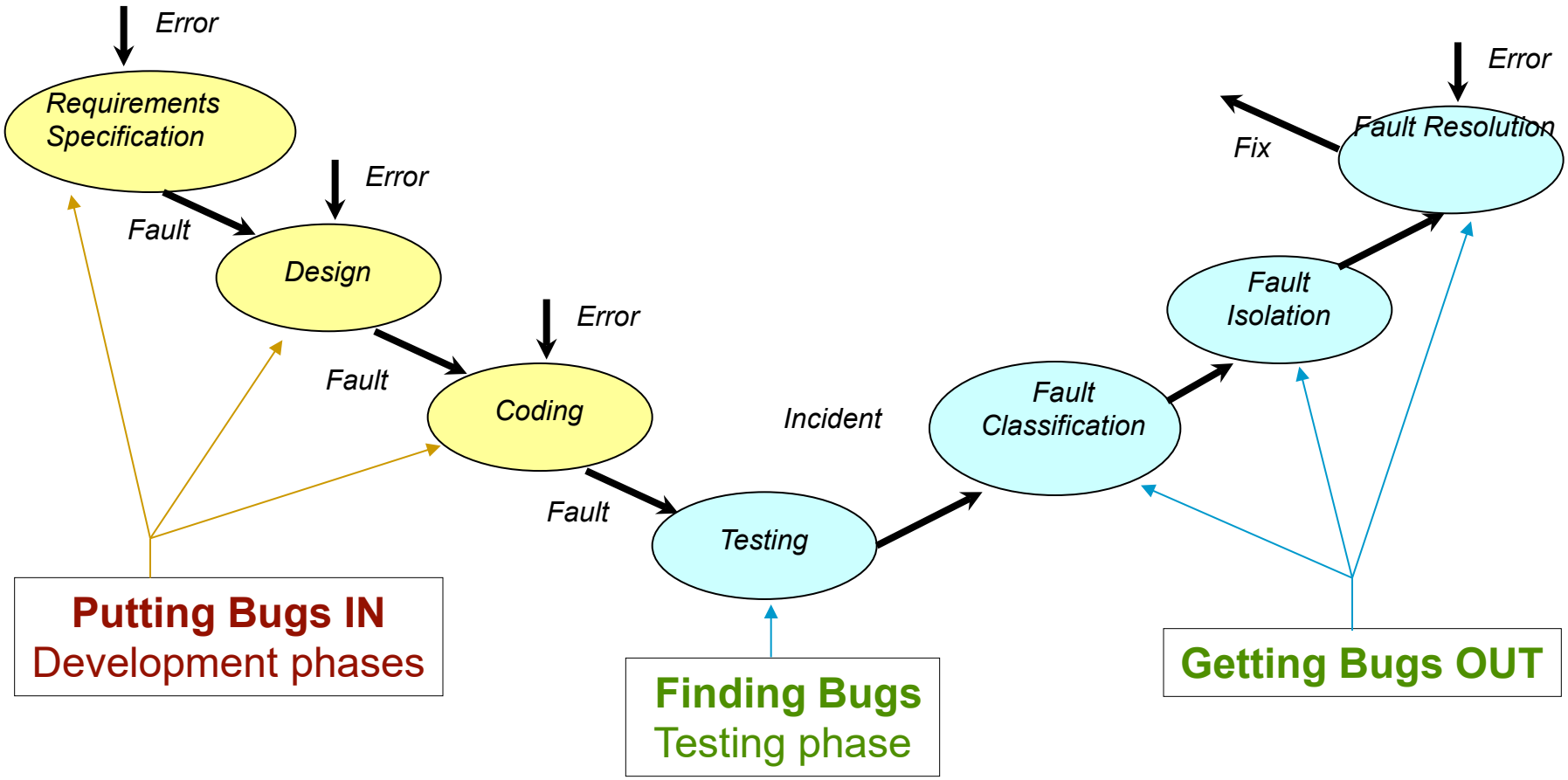
- Development team needs to work with Test team
- “Egoless Programming”

How is the testing done?

- No point in testing without a plan
- Define at what stages what types of testing will be performed



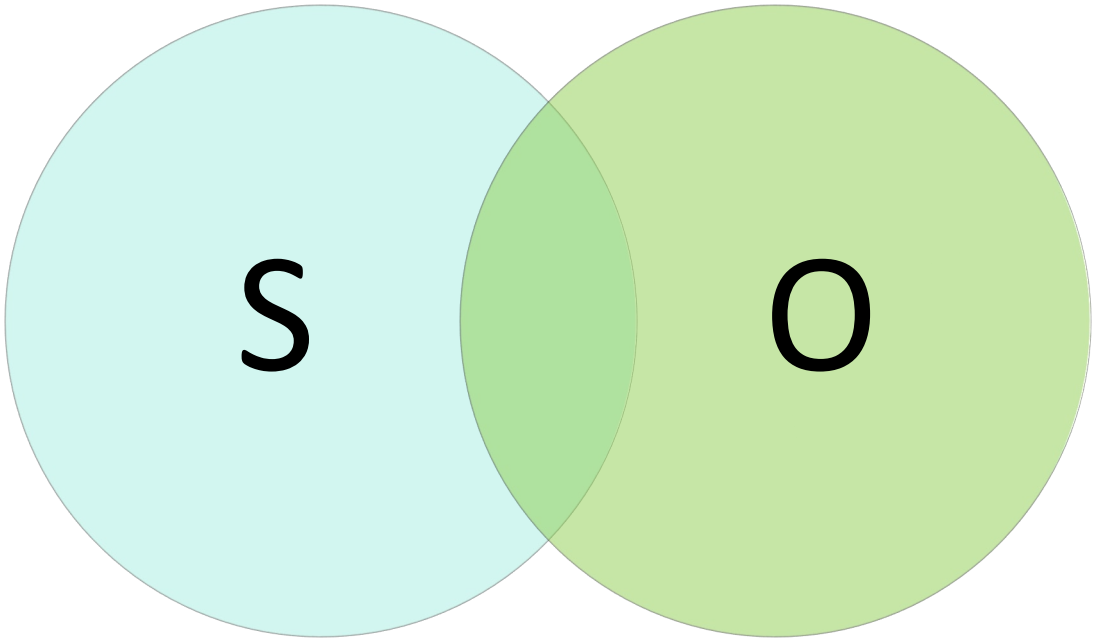
The V-model from the tester perspective



Program behavior

Specification (expected behavior)

Program (observed behavior)



Missing functionality (Faults of omission)

Extraa Functionality (Faults of commission)

Correct portion

Types of Faults

- Algorithmic: division by zero
- Computation & Precision: rounding error
- Documentation: discrepancy between documentation and code
- Stress/Overload: data-structure size (dimensions of tables, size of buffers)
- Capacity/Boundary: x devices, y parallel tasks, z interrupts
- Timing/Coordination: failure to meet real-time deadlines
- Throughout/Performance: failure in processing x requests/minute
- Recovery: power failure
- Hardware & System Software: problems with network connection
- Standards & Procedure: a release has not been properly reviewed

Course standard

Contents of a Test Case

"Boilerplate": author, date, purpose, ***test case ID***

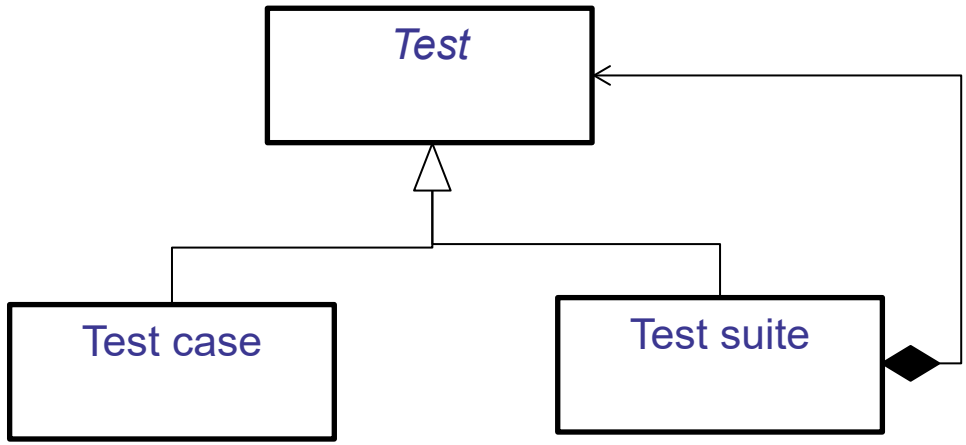
Pre-conditions (including environment)

Inputs

Expected Outputs

Observed Outputs

Pass/Fail



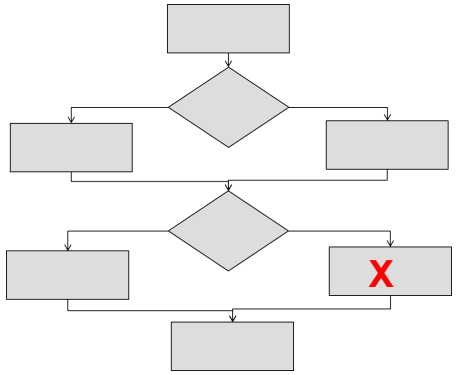
Testing Approaches

Specification

R1: Given **input**, the software shall provide **output**.

Functional (Black Box) establishes confidence

Program

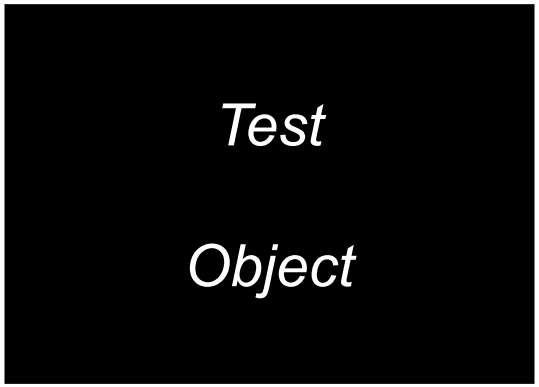


Find **input** and **output** so that **X** is executed.

Structural (White Box) seeks faults

The oracle problem

Input



Output

Oracle

Two Types of Oracles

- **Human**: an expert that can examine an input and its associated output and determine whether the program delivered the correct output for this particular input.

- **Automated**: a system capable of performing the above task.

R2: “The answer is 42.”

42

42.0

XVII

41.99999

Black-box/ closed box testing

Testing based only on specification:

1. Exhaustive testing
2. Equivalence class testing (Equivalence Partitioning)
3. Boundary value analysis
4. Test tables

1. Exhaustive testing

Definition: testing with every member of the input value space.

Input value space: the set of all possible input values to the program.

– Sum of two 16 bit integers: 2^{32} combinations

One test per ms takes about 50 days.

2. Equivalence Class Testing

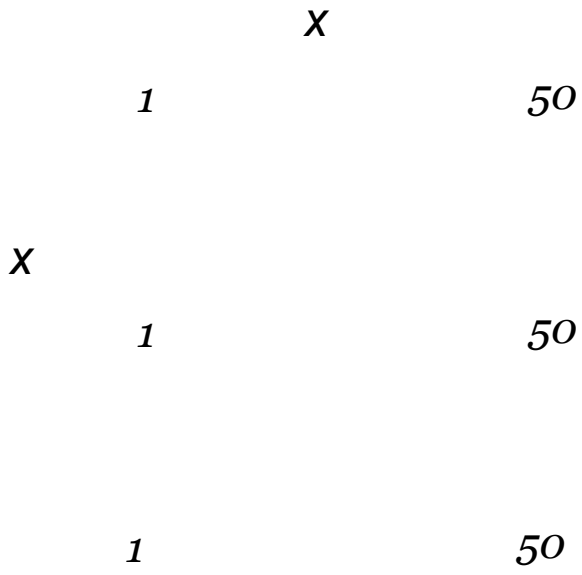
- Equivalence Class (EC) testing is a technique used to reduce the number of test cases to a manageable level while still maintaining reasonable test coverage.

- Each EC consists of a set of data that is treated the same by the module or that should produce the same result. Any data value within a class is ***equivalent***, in terms of testing, to any other value.

Identifying the Equivalence Classes

Taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups:

- Input condition
  - range of values x: 1-50
- Valid equivalence class
  - 1 <= x <= 50
- Invalid equivalence classes
  - x < 1
  - x > 50



Two-variable example

Validate loan application forms against the rule:

- If you are 18 years and older, you can borrow maximally 100.000, but not less than 10.000.
- Variable: **age**
  - EC1: age < 18
  - EC2: age >= 18
- Variable: **sum**
  - EC3: sum < 10.000
  - EC4: 10.000 <= sum <= 100.000
  - EC5: sum > 100.000

Two-variable example, test-cases

Test-case id	Age	Sum	Valid form
1	32	55.300	Yes
2	13	72.650	No
3	44	9.875	No
4	50	60.000	Yes
5	87	103.800	No

Arbitrary, valid sums

Arbitrary, valid ages

Two linked variables

Validate loan application forms against the rule:

- If you are **21 years and older**, you can borrow maximally 100.000, but not less than 10.000.
- If you are between 18 and 21 you can borrow between 10.000 and 20.000.

Now you have to think about the combination between the variables!

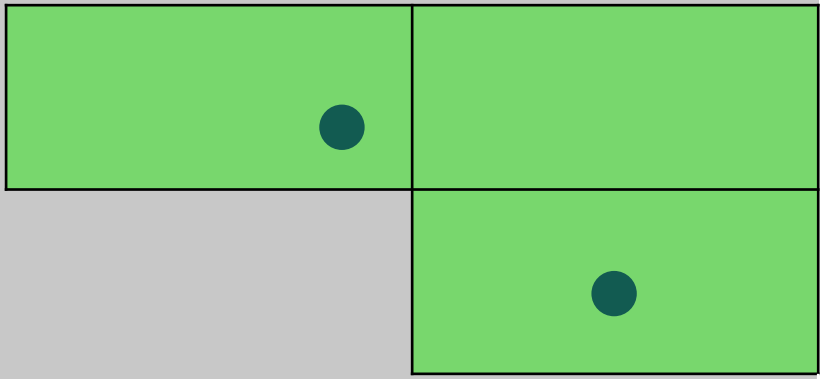
Two-variable example, updated

- Variable: **age**
  - EC1: age < 18
  - EC2: 18 <= age <=21
  - EC3: 21 < age
- Variable: **sum**
  - EC4: sum < 10.000
  - EC5: 10.000 <= sum <= 20.000
  - EC6: 20.000 < sum <= 100.000
  - EC7: sum > 100.000

Valid EC

Weak EC approach, valid EC testing:

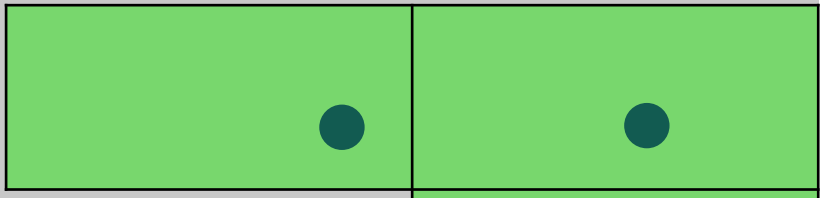
Age/Sum	Under 18	18 - 21	Over 21
Under 10.000			
10.000-20.000			
20.001-100.000			
Over 100.000			



Each valid EC/ variable covered at least once

Strong EC approach , valid EC testing :

Age/Sum	Under 18	18 - 21	Over 21
Under 10.000			
10.000-20.000			
20.001-100.000			
Over 100.000			



All valid combinations covered

Guidelines

1. If an input condition specifies a *range* of values; identify one valid EC and two invalid EC.

2. If an input condition specifies the *number* (e.g., one through 6 owners can be listed for the automobile); identify one valid EC and two invalid EC (- no owners; - more than 6 owners).

3. If an input condition specifies a set of input values and there is reason to believe that each is handled differently by the program; identify a valid EC for each and one invalid EC.

4. If an input condition specifies a “must be” situation (e.g., first character of the identifier must be a letter); identify one valid EC (it is a letter) and one invalid EC (it is not a letter)

5. If there is any reason to believe that elements in an EC are not handled in an identical manner by the program, split the equivalence class into smaller equivalence classes.

Applicability and Limitations

- Most suited to systems in which much of the input data takes on values within ranges or within sets.

- It makes the assumption that data in the same EC is, in fact, processed in the same way by the system. The simplest way to validate this assumption is to ask the programmer about their implementation.

- EC testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs or outputs that can be partitioned based on the system's requirements.

3. Boundary Value Testing

Boundary value testing focuses on the boundaries simply because that is where so many defects hide.

The defects can be in the requirements or in the code.

Technique

Course standard

1. Identify the ECs.

2. Identify the boundaries of each EC.

3. Create test cases for each boundary value by choosing one point on the boundary, one point just below the boundary, and one point just above the boundary.

Specification: the program accepts four to eight inputs which are 5 digit integers greater than or equal to 10000.

Input values

Less than 10000 | *Between 10000 and 99999* | *More than 99999*

Number of input values

Less than 4 | *Between 4 and 8* | *More than 8*

Boundary value analysis

Less than 10000

Between 10000 and 99999

More than 99999

Applicability and Limitations

Boundary value testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs that can be partitioned and boundaries that can be identified based on the system's requirements.

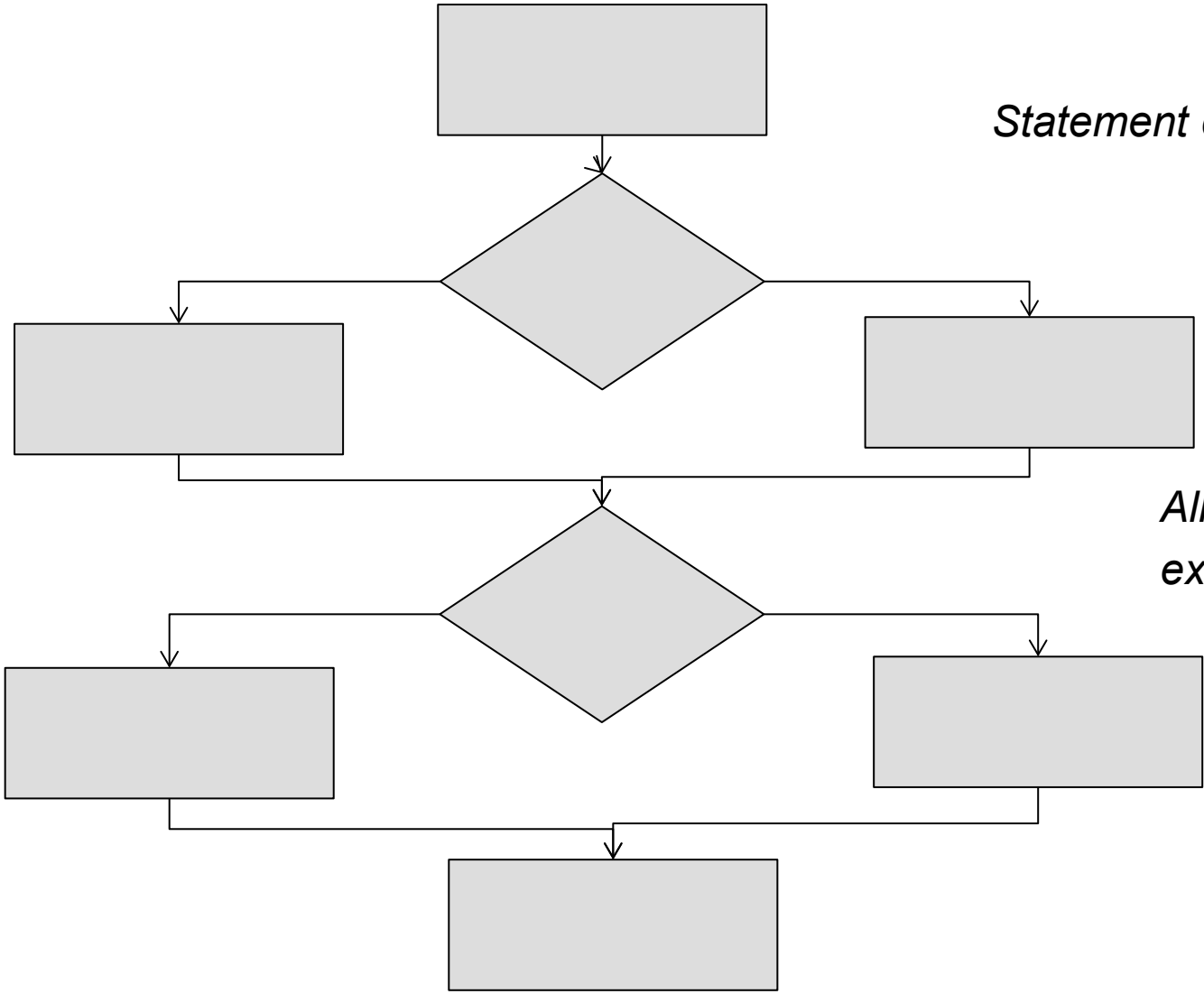
It makes the assumption that the implementation is consistent for entries of the same type!

Test table

"If a client been employed over 3 years and has a salary over 40.000 SEK or has a history of succesful loans, the loan is granted automatically, otherwise the application is sent for review"

Id	Monthly salary	Has a history of successful loans	Employment duration	Loan application
1	45.000 SEK	Yes	1 year	**Granted**
2	45.000 SEK	No	5 years	**Granted**
3	45.000 SEK	No	2 years	**Sent for review**
…	…	…	…	**…**

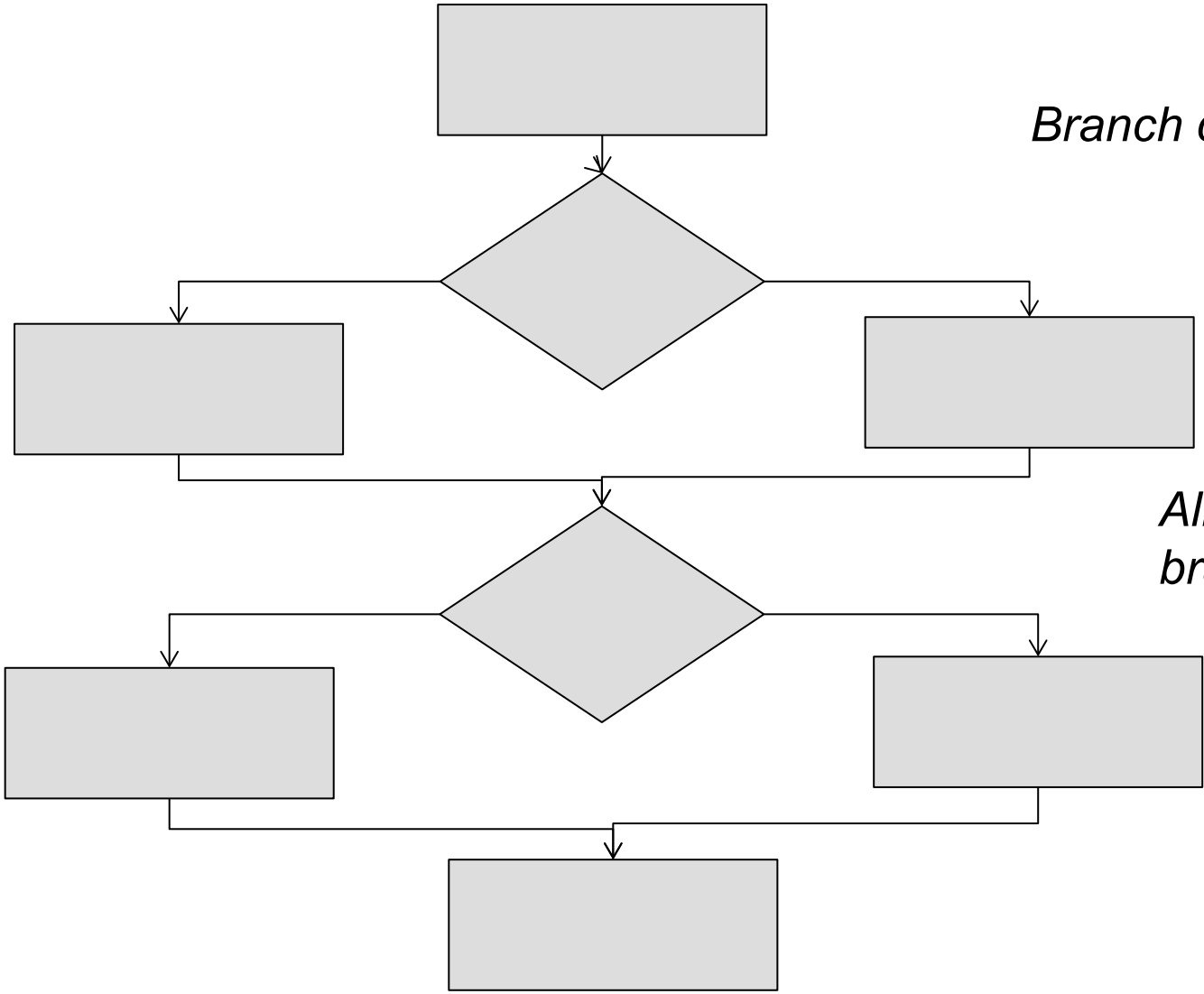
Control-flow based coverage



Statement coverage

All statements executed

Control-flow based coverage

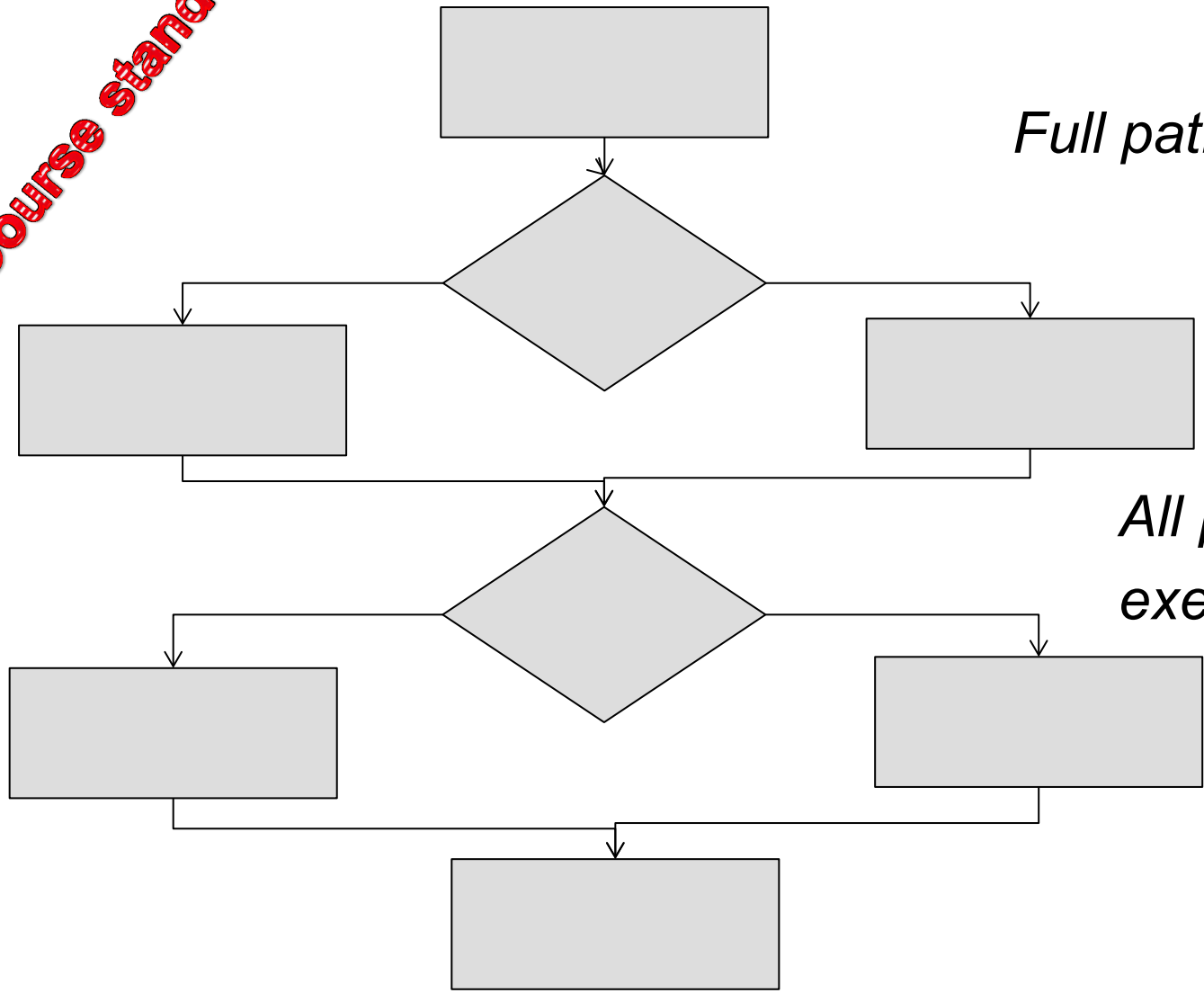


Branch coverage

All decision branches tried

Control-flow based coverage

Course standard

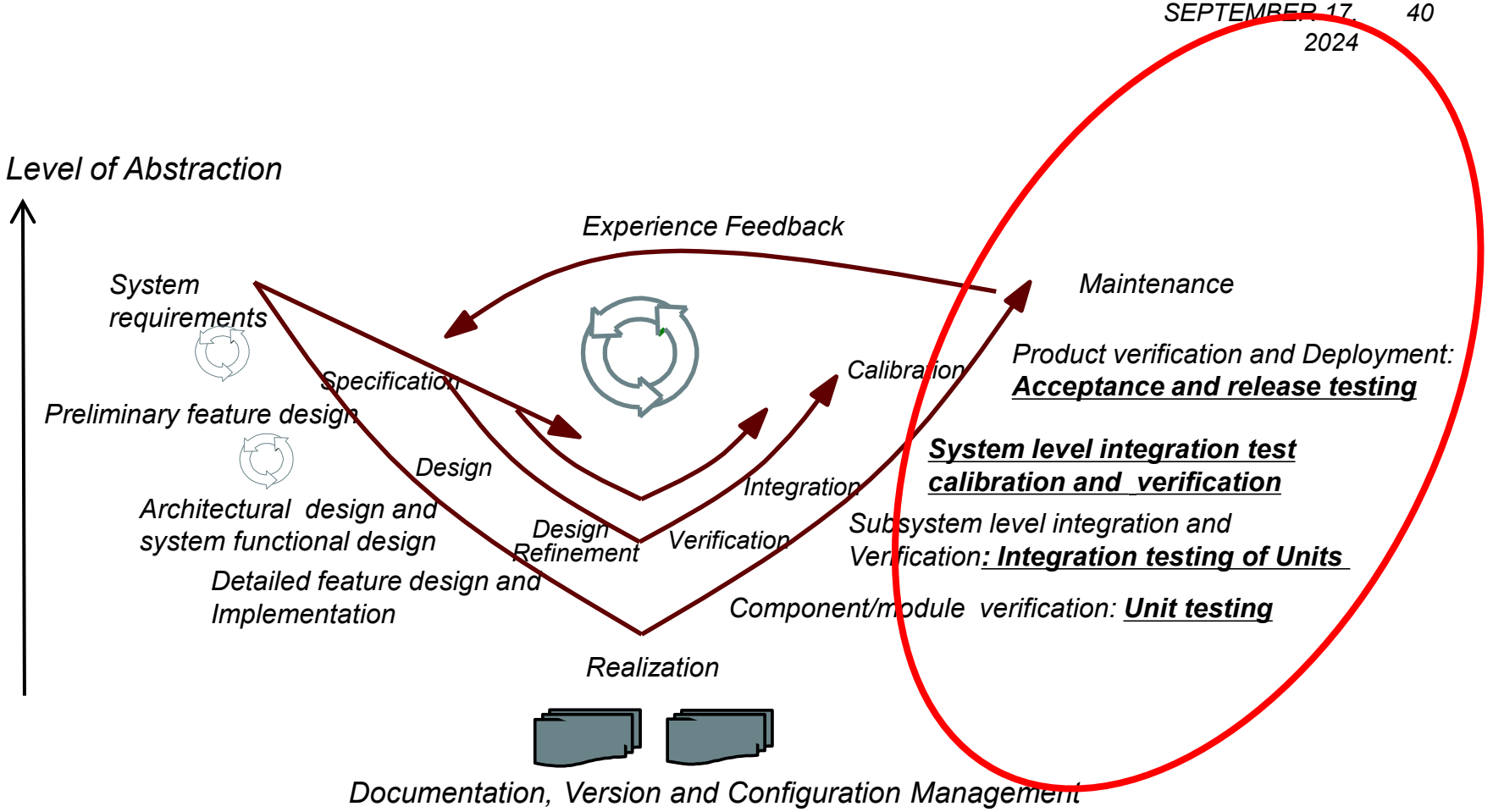


Full path coverage

All possible paths executed

Example

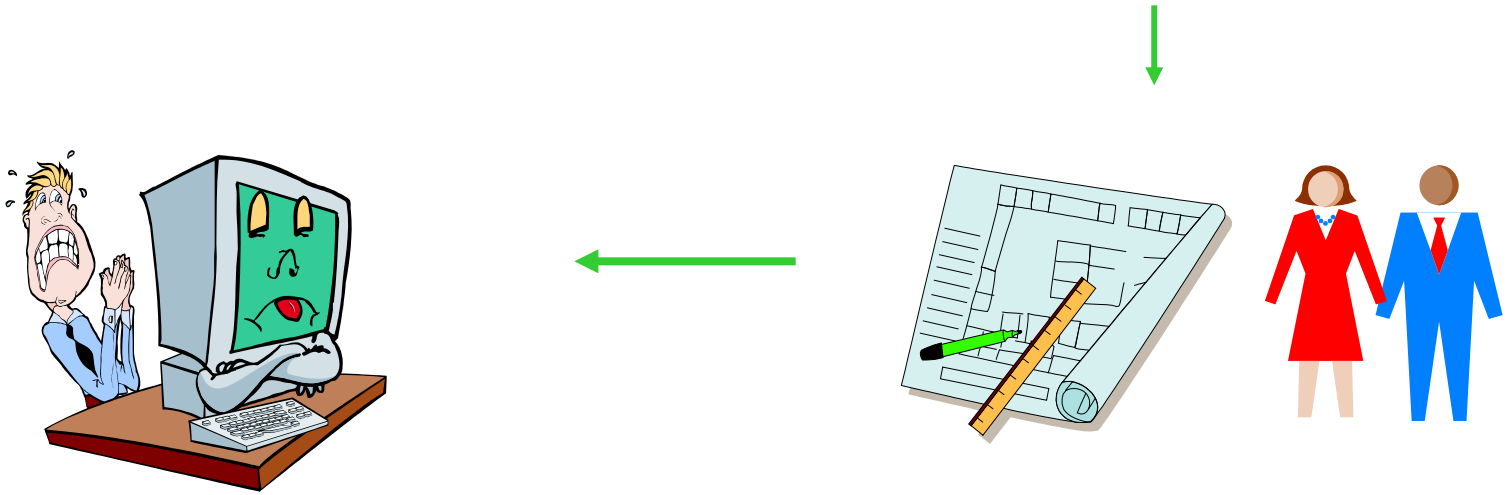
```
public int returnInput(int start, boolean c1, boolean c2, boolean c3) {
    int x = start;
    int y = 0;
    if (c1)    y = y + x;
    if (c2)    x++;
    if (c3)    y = 0;
    return y;
}
```



Levels of testing

Unit Testing

Objective: to ensure that code implemented the design properly.



Code = System

Design Specification

Often done by the programmers themselves.

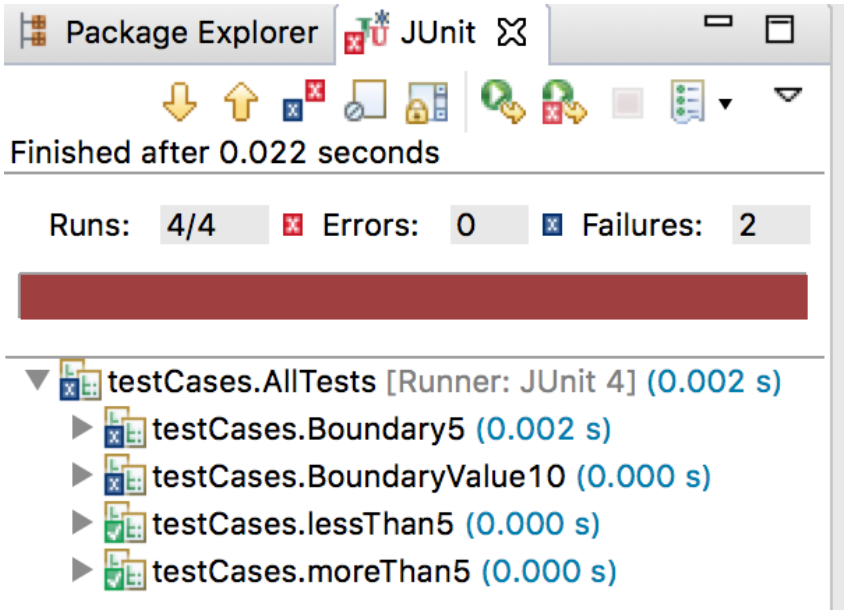
xUnit

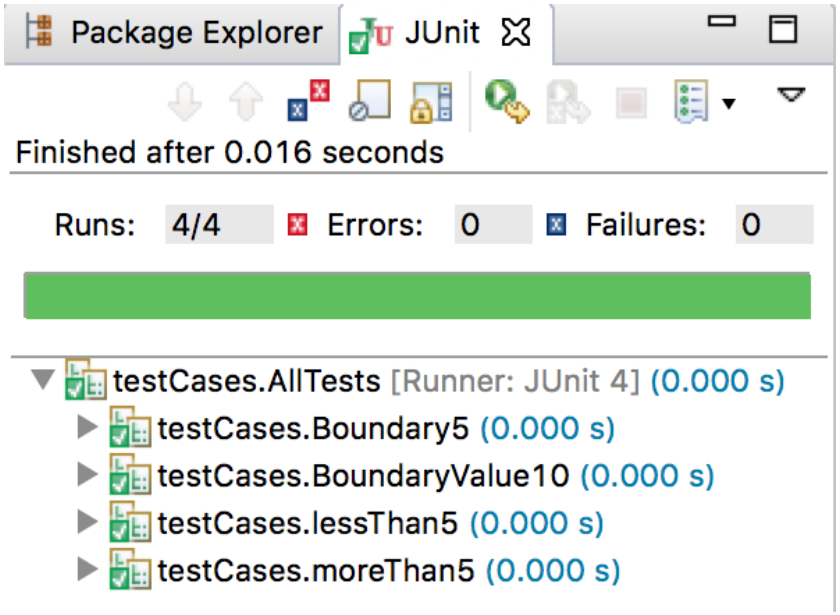
- xUnit is a set of tools for regression testing
- x denotes a programming language
- Junit, for Java is one of the earliest and most popular
- TDDC88 has a lab – do that

- Recommended primer:

http://www.it-c.dk/~lthorup/JUnitPrimer.html

JUnit interface





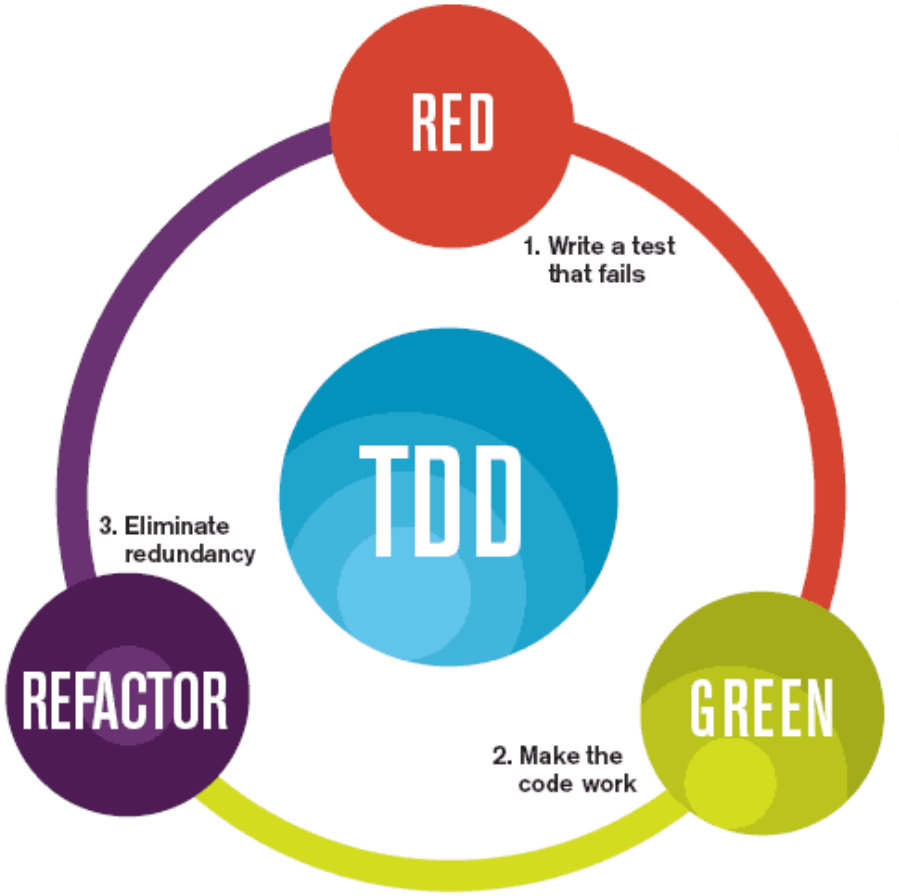
*assertEquals("Checks the boundary value 5", **true, tester.isBetween5and10(5));***

message if fail

expected

actual

Test-Driven Development (TDD)



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

source: Redmond Developer

Behaviour Driven Development (BDD)

- A refinement of TDD

The criteria for naming tests: *the software under test should do something*

```
public void shouldDoX() { // ... }
```

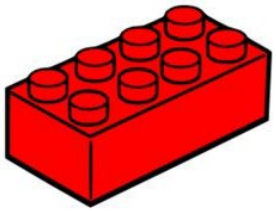
- Example: ClientDetailsValidatorTest {

```
public void testShouldFailForMissingSurname (){...}
public void testShouldFailForMissingTitle (){...} }
```

- If a test does not fit the pattern → refactor
- Choosing the next test: What is the next most important behaviour?

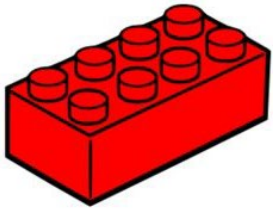
Integration testing

Component code



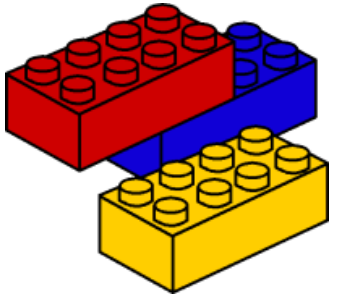
Unit test

Tested components

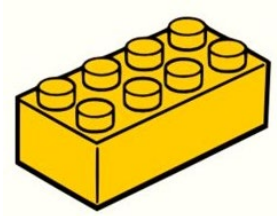


Integration test

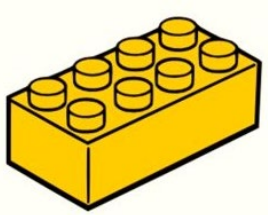
Design Specification



Integrated module



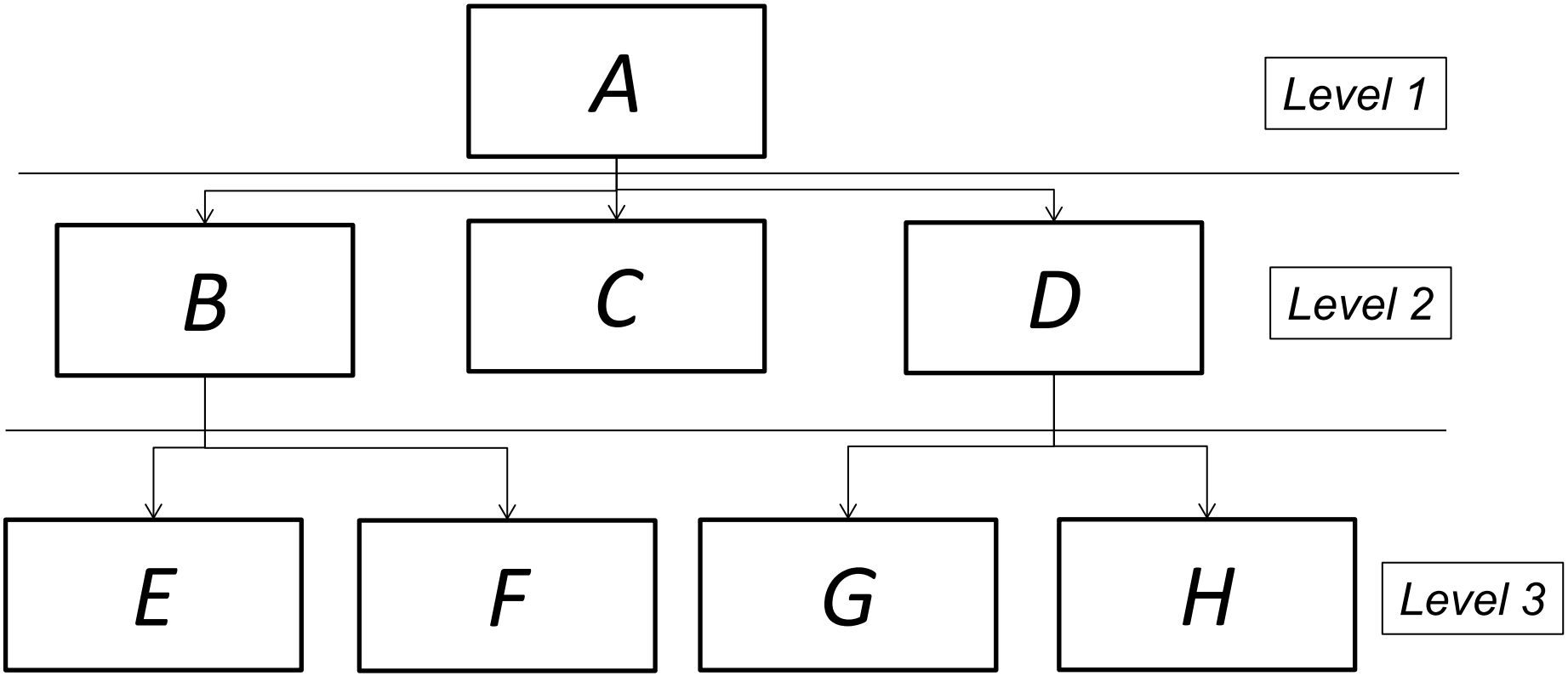
Unit test

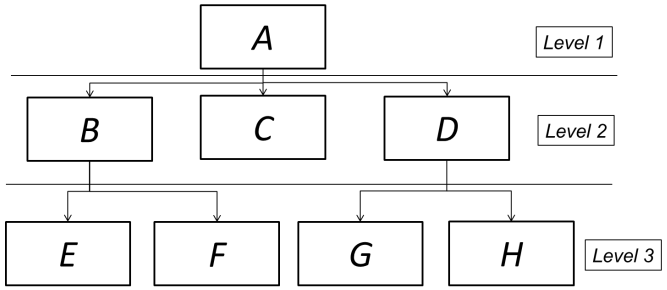


Integration Testing strategies

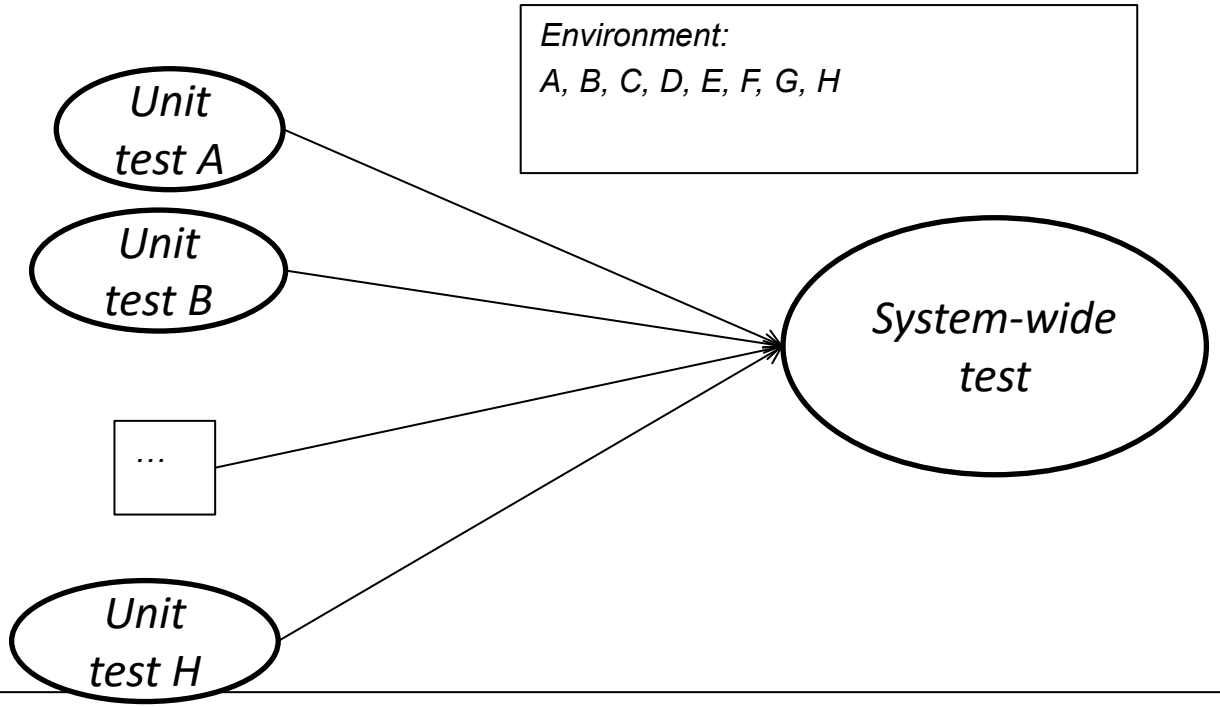
1. Big-bang
2. Bottom-up
3. Top-down
4. Sandwich

Three level functional decomposition tree



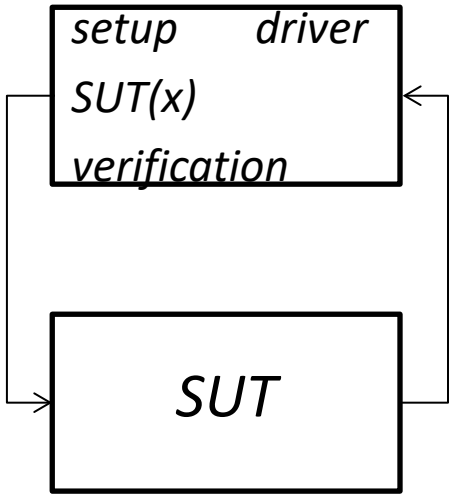


Big-Bang testing

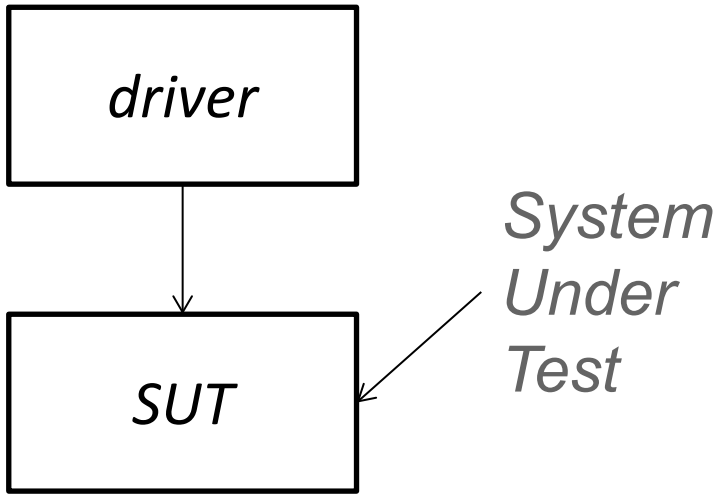


Driver

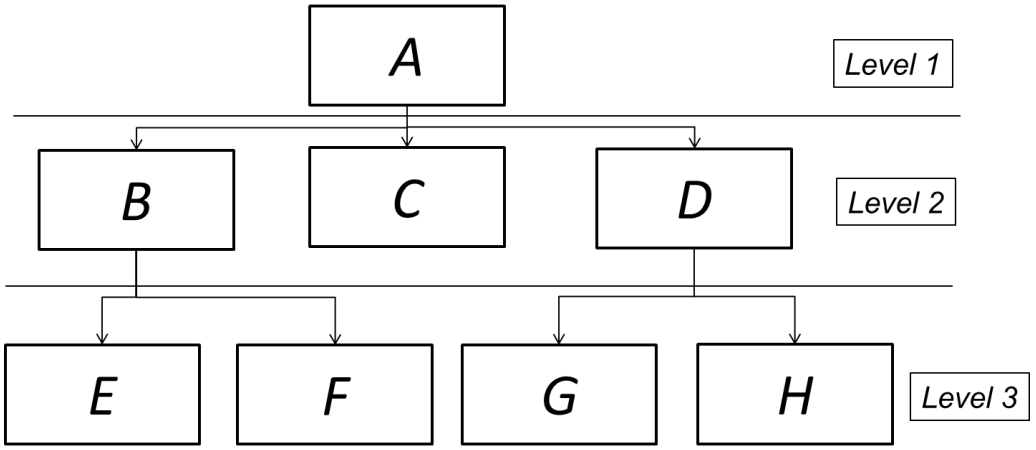
A pretend module that requires a sub-system and passes a test case to it

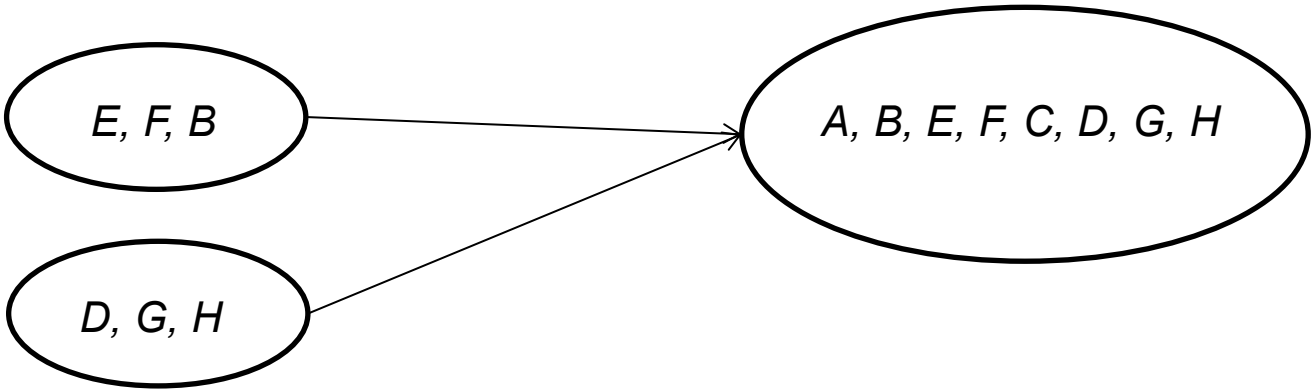


Black-box view



Bottom-up testing





Is bottom-up smart?

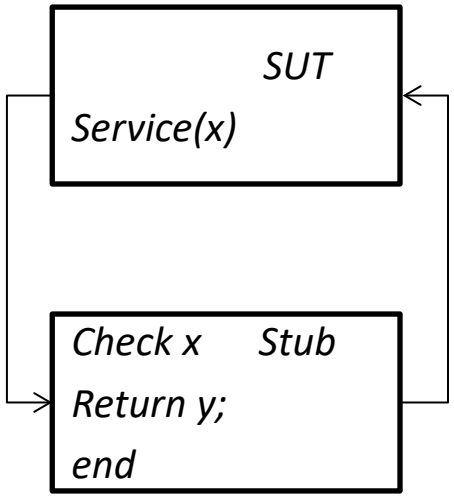
- If the basic functions are complicated, error-prone or has development risks
- If bottom-up development strategy is used
- If there are strict performance or real-time requirements

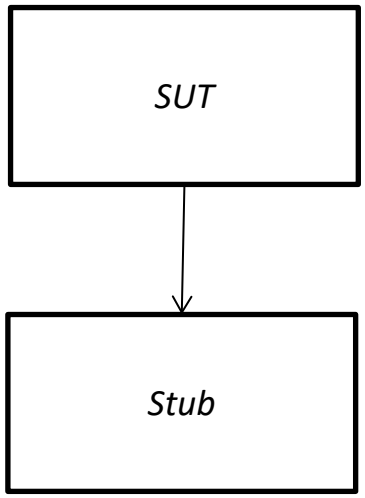
Problems:

- Lower level functions are often off-the shelf or trivial
- Complicated User Interface testing is postponed
- End-user feed-back postponed
- Effort to write drivers.

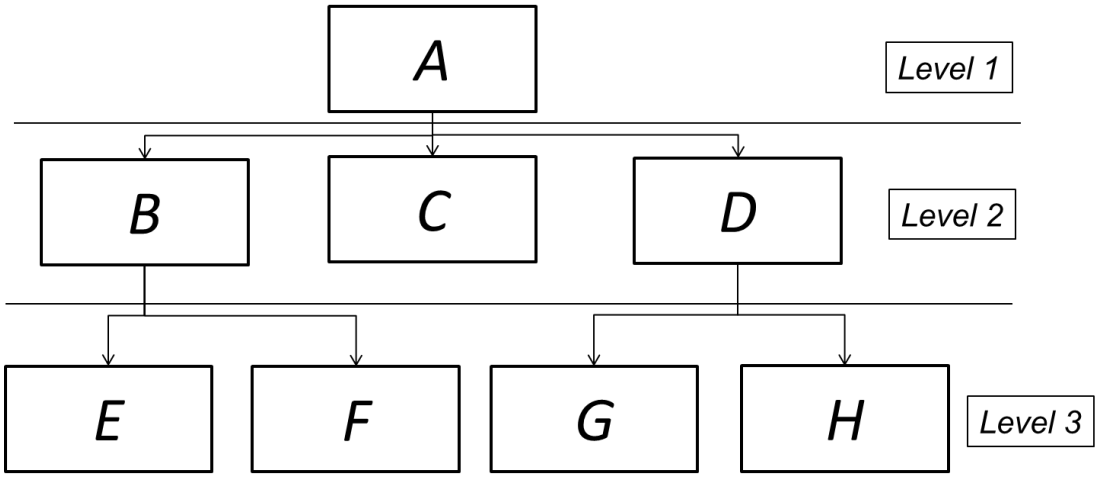
Stub

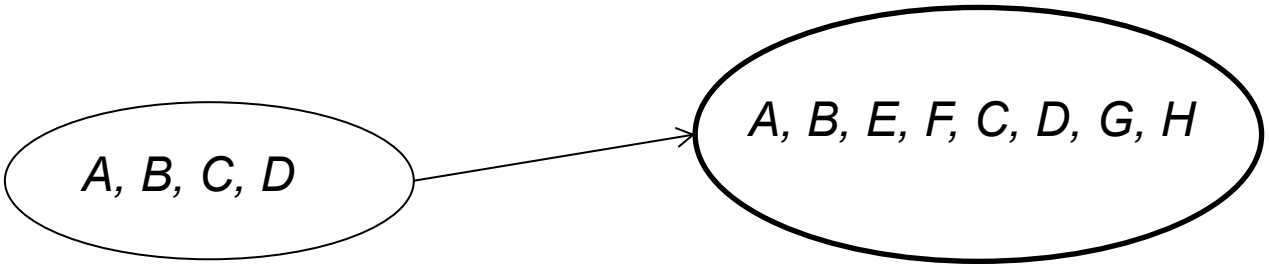
- A program or a method that **simulates the input-output functionality** of a missing sub-system by answering to the decomposition sequence of the calling sub-system and returning back simulated or ”canned” data.





Top-down testing





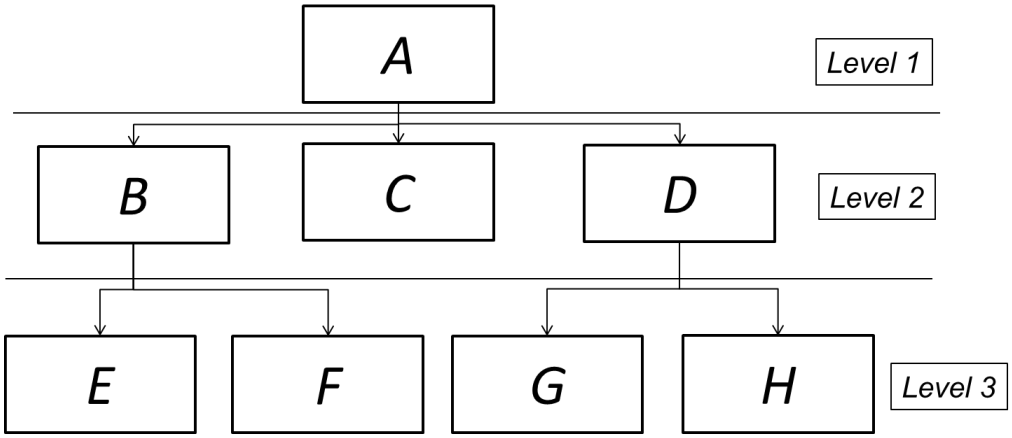
Is top-down smart?

- Test cases are defined for functional requirements of the system
- Defects in general design can be found early
- Works well with many incremental development methods
- No need for drivers

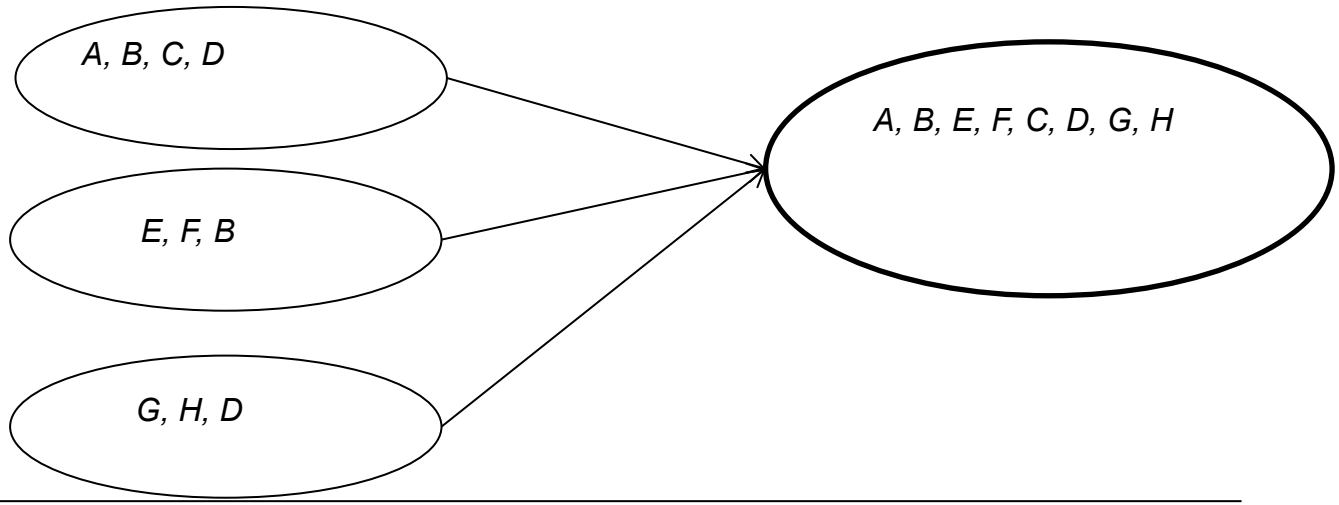
Problems:

- Technical details postponed, potential show-stoppers
- Many stubs are required
- Stubs with many conditions are hard to write

Sandwich testing



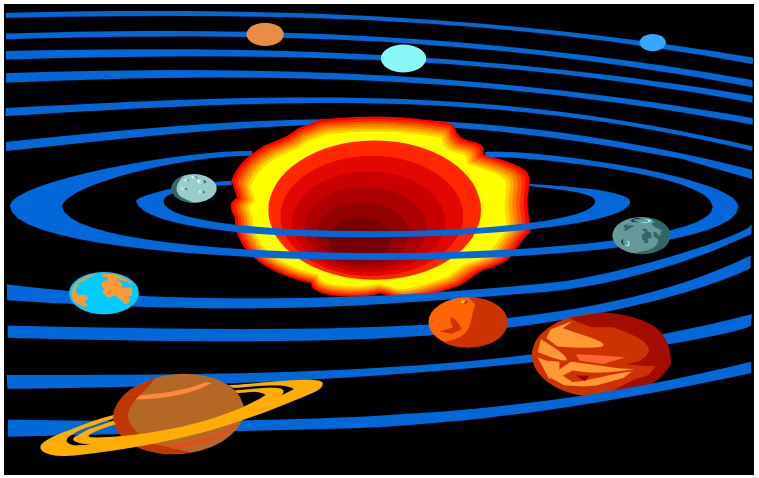
target level



Is sandwich testing smart?

- Top and Bottom Layer Tests can be done in parallel
- Problems:
- Does not test the individual subsystems on the target layer thoroughly before integration

System Testing



Function testing/Thread testing

(testing one function at a time)

functional requirements

A function test checks that the integrated system performs its function as specified in the requirement

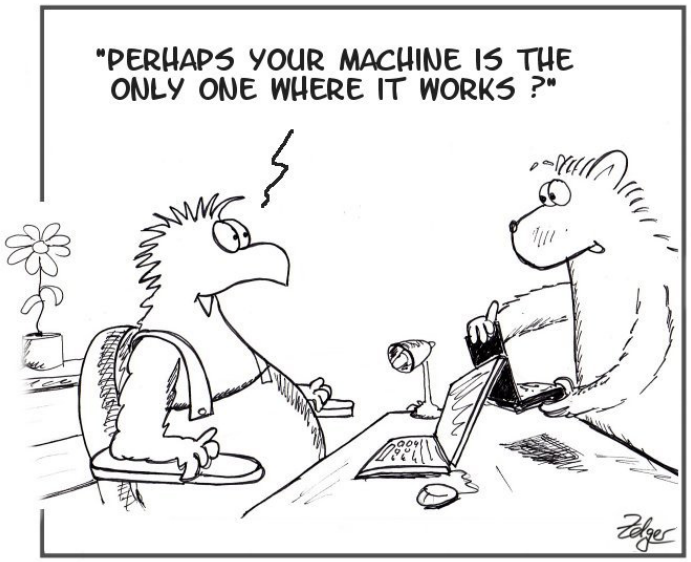
- Guidelines
  - know the expected actions and output
  - test both valid and invalid input
  - never modify the system just to make testing easier
  - have stopping criteria

Testing non-functional requirements

- Stress tests
- Timing tests
- Volume tests
- Configuration tests
- Compatibility tests
- Regression tests
- Security tests
- (physical) Environment tests
- Quality tests
- Recovery tests
- Maintenance tests
- Documentation tests
- Human factors tests / usability tests

Acceptance Testing

Customers, users needs



It works on my machine

***Benchmark test**: a set of special test cases*

***Pilot test**: everyday working*

- *Alpha test: at the developer's site, controlled environment*
- *Beta test: at one or more customer site.*

***Parallel test**: new system in parallel with previous one*

GUI Testing

- GUI application is **event driven**; users can cause any of several events in **any order** – this means testing sequences - explosion of number of tests
- Unit testing is typically at the "button level"; that is **buttons have functions**, and these can be tested in the usual unit-level sense.
- The essence of **system-level testing** for GUI applications is to exercise the event-driven nature of application
- GUI testing is more expensive/harder to automate – consistent unit testing at lower levels can help reduce cost
- Challenges: Repeatability and regression

A wide range of GUI testing tools has appeared on the market over the past few years.

TDDC88 has a lab on Selenium

Smoke test

- Important selected tests on module, or system
- Possible to run fast
- Build as large parts as possible as often as possible
- Run smoke tests to make sure you are on the right way

Termination Problem :
How decide when to stop testing

- The main problem for managers!

Termination is influenced by:

- Deadlines, e.g. release deadlines, testing deadlines;
- Test cases completed with certain percentage passed;
- Test budget has been depleted;
- Coverage of code, functionality, or requirements reaches a specified point;

Summary

- Testing context and goals
- Fault classification
- Black Box techniques
- White box techniques
- Testing at different levels
- Non functional requirement testing

The end. Thank you! Questions?

www.liu.se

LINKÖPING UNIVERSITY