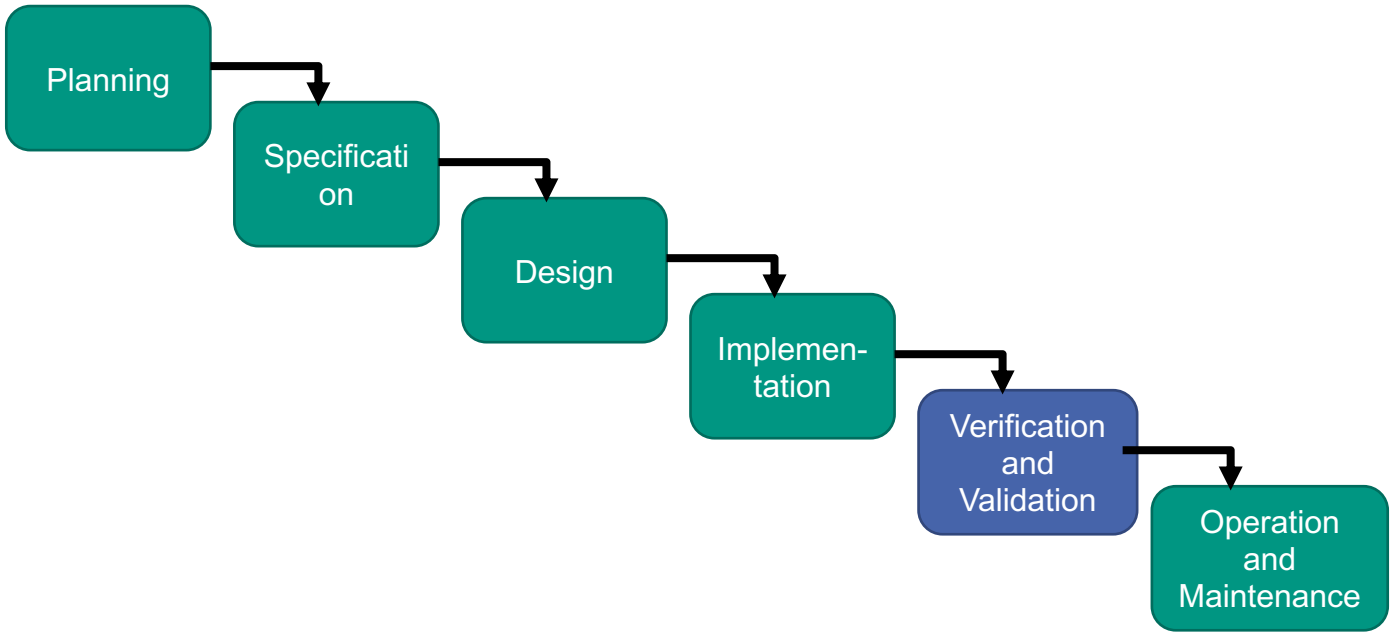
Chap. 5.1 – Test and Inspection

Walter F. Tichy

Where are we right now?



Literature

- This lecture is based on chapter 11 of

  B. Bruegge, A.H. Dutoit, **Object-Oriented Software Engineering: Using UML, Patterns and Java,** Pearson Prentice Hall, 2004 and 2010.

Motivation

- Software artifacts always contain defects.
- The later a defect is found, the more expensive it is to fix it.
- The goal is to find defects as early as possible.

Defect detection is the goal of the test procedures

"Testing shows the presence of bugs, not their absence."

(Edsger W. Dijkstra)

Defect detection is the goal of the test procedures

- Complete testing of all combinations of all input values is not possible except for trivial programs (because of an astronomical number of test cases).
- Correctness is only possible with formal proof of correctness (correspondence of specification and program); today this is only possible for small programs.
- Central question: When can you stop looking for errors in testing? (Need Test Completeness Criteria)

Defect detection is the goal of the test procedures

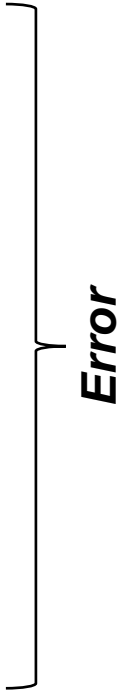
- Complete testing of all combinations of all input values is not possible except for trivial programs (because of an astronomical number of test cases).
- Correctness is ess (correspondenc s is only possible for sm
- Central questio testing? (Test Completeness

Distinguish

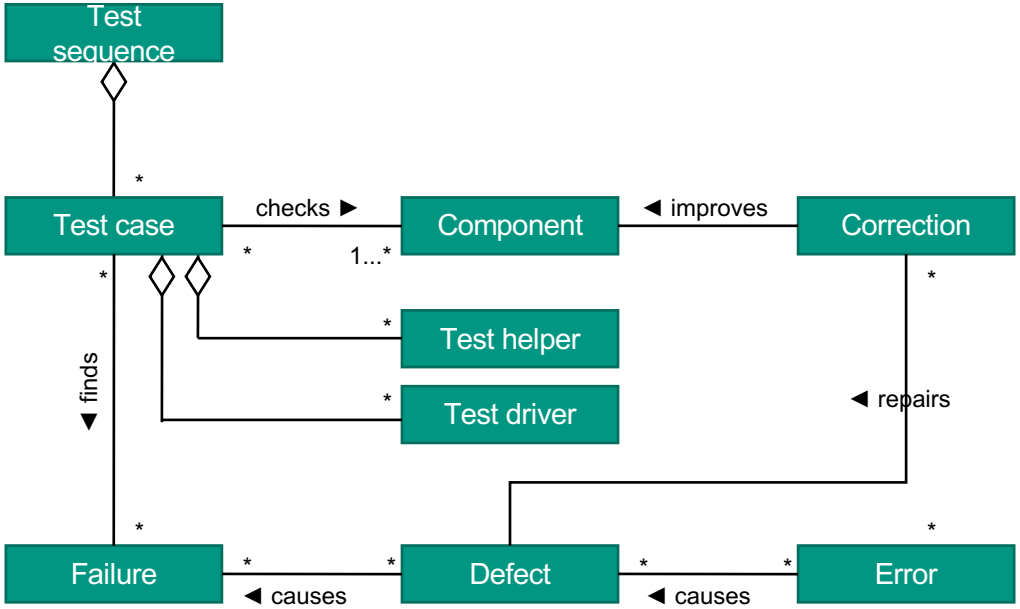
- Testing procedures → Detect defects
- Verifying procedures → Prove correctness
- Analyzing methods → Determine properties of a system component

There are 3 types of errors....

- A ***failure*** or ***fault*** is the deviation of the behavior of the software from the specification (this is an event: wrong output or crash).
- A **defect** or **bug** is a deficiency in a software product that can (but need not) lead to failure (a condition).
  A defect is said to manifest itself in a failure or breakdown.
- A ***mistake*** or ***manufacturing error*** is a human (or tool) action that causes a defect (a process).



UML modeling of the three types of errors



Types of test helpers (also called doubles)

- A ***stub*** is a rudimentarily implemented part of the software (usually doing nothing) and serves as a placeholder for functionality that has not yet been implemented. Stubs can be methods, procedures, or classes with stubs.
- A ***dummy*** simulates the implementation for test purposes. A dummy method may always return the same values or perform the same actions. A dummy class contains dummy methods.
- **A *mock object*** is a dummy with additional functionality, such as logging calls or checking the calls coming in from the "client". It can be used to test the behavior of the client.

- More details later.

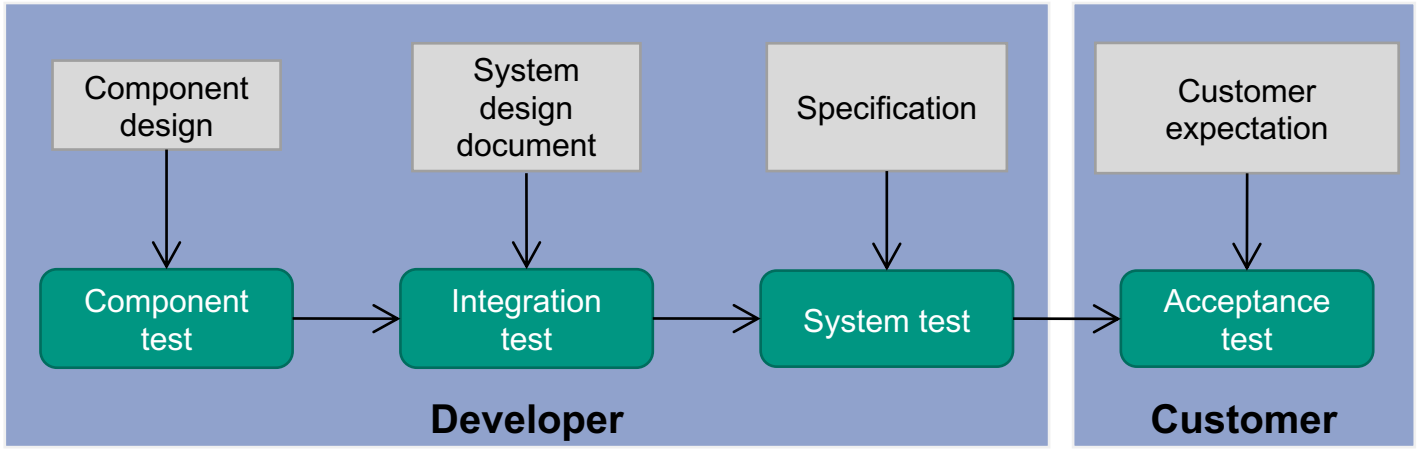
Defect classes

- Requirement defect (defect in the requirements specification)
  - Incorrect specification of user requests
  - Incomplete or unclear information on functional requirements, performance requirements, etc.
  - Inconsistency of different requirements
  - Impracticability
- Design defect (defect in the design document)
  - Incomplete or incorrect implementation of the requirements
  - Inconsistency between specification and design
- Implementation defect (defect in the program)
  - Incorrect conversion of specification or design into a program

Module/software test procedure

- A **software test**, or test for short, executes a single software component or a configuration of software components under known conditions (inputs and execution environment) and checks its behavior (outputs and reactions).
- The SW component or configuration to be tested is called test object or **component under test** (CUT).
- A **test case** consists of a set of data for the execution of a part of the test object.
- A **test driver or testing framework** provides test objects with test cases and triggers the execution of the test items (interactively or automatically).

Test phases



The **component test (unit test)** checks the function of a single module or class by observing the processing of test data.
Goal: Confirm that the component is correctly coded and carries out the intended functionality

The **integration test** checks step by step the interaction of system components that have already been tested in unit test.
Goal: Test the interfaces and interactions among the subsystems.

The **system test** is the final test of the entire system by the developer in real (or realistic) environment without customers.
Goal: Determine if the system meets the requirements (functional and nonfunctional)

The **acceptance test** is the final test of the whole system in a real environment under observation, participation, and/or direction of the customer at the customer's site.
Goal: Demonstrate that the system meets the requirements and is ready for use.

Classification of testing methods (1)

- Dynamic methods: executing code
  - Structural testing (white box/glass box testing)
    - Control flow-oriented tests
    - Data flow-oriented tests
  - Functional testing (*black box testing*)
  - Performance tests (also *black box*)
- Static methods: analyzing code
  - Manual test methods (inspection, review, walkthrough)
  - Program analysis tools

Testing

Check

Classification of testing methods (2)

- Dynamic methods
  - The translated executable program is provided with certain test cases and executed
  - The program is tested in the real (realistic) environment
  - This is a sampling method: Correctness of the program is **not proven!**
- Static methods
  - The program (component) is not executed, but the source code is analyzed

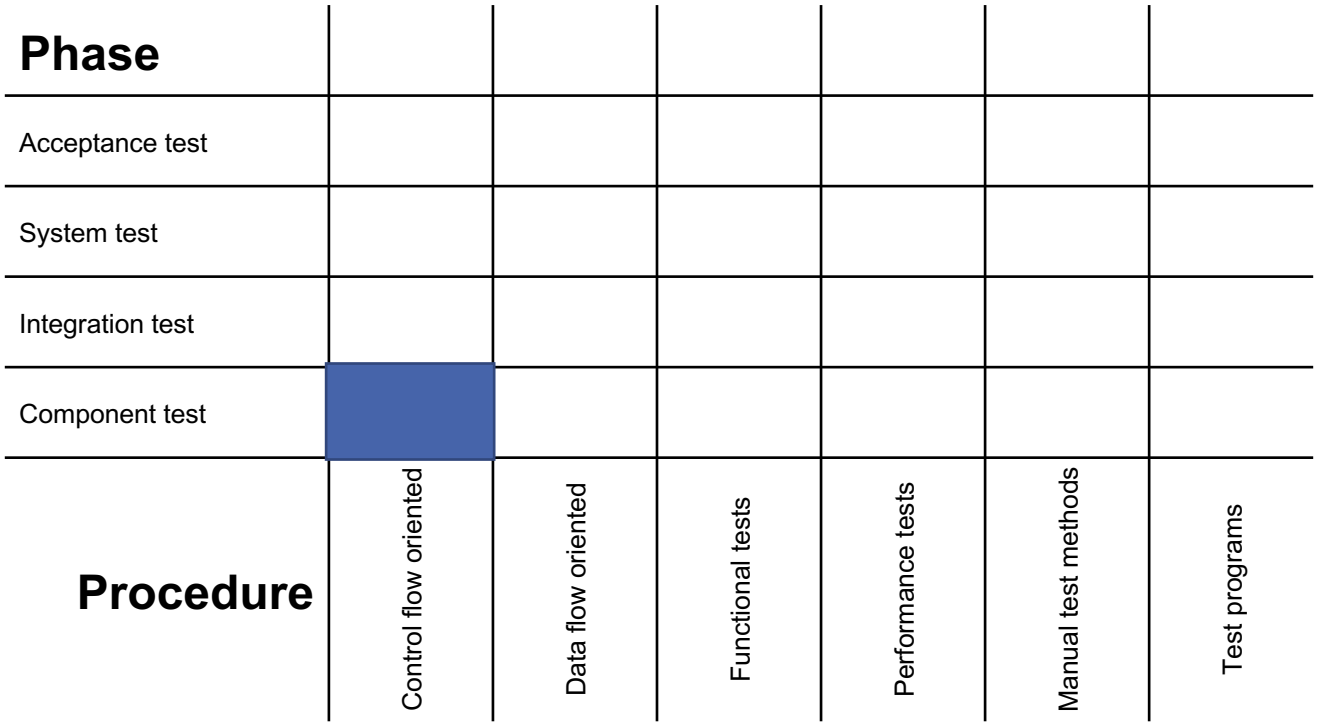
Classification of testing methods (2)

- Dynamic methods
  - The translated executable program is provided with certain test cases and executed
  - The program is tested in the real (realistic) environment
  - This is a sampling method: Correctness of the program is not proven!
- Static methods
  - The program (component) is not executed, but the source code is analyzed

White box testing: determine values **with knowledge of** control and/or data flow

Black box testing: determining values **without knowledge of** control and/or data flow; only from specification

Overview matrix: What comes next?



Control flow oriented (CFO) test procedures

- Statement coverage
- Branch coverage
- Boundary-interior test
- Path coverage
- Condition coverage (not relevant for the exam)
  - Simply
  - Multiple
  - Minimal multiple

Before that, however...

- Completeness criteria for test procedures are defined via "control flow graphs".
  - Definition of an intermediate language
  - Definition of the transformation to the intermediate language
  - Definition of the control flow graph
- Then: definition of the testing procedures and their coverage

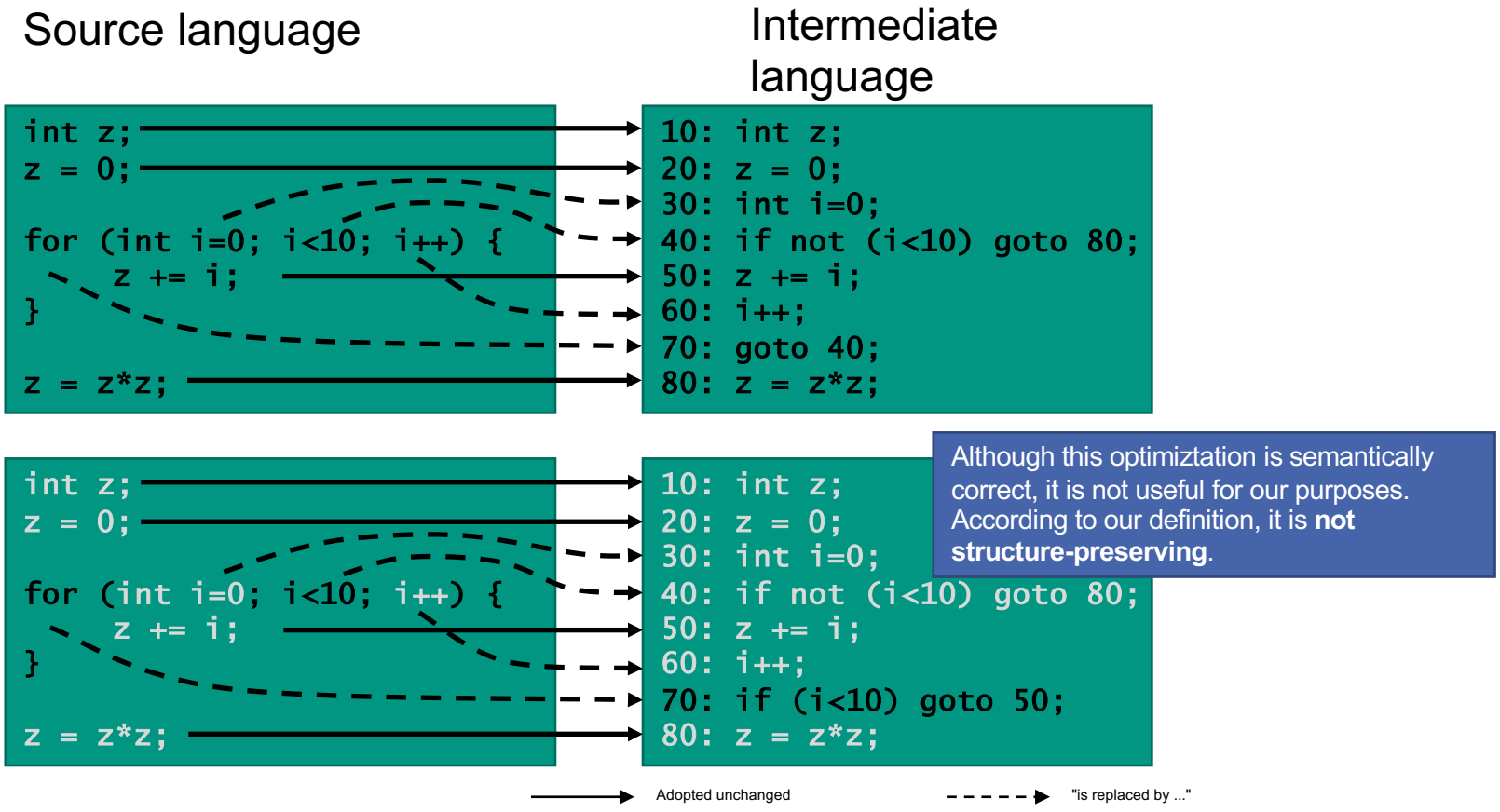
Definition: Intermediate language

- We define an **intermediate language** consisting of:
  - any statement except those that affect the execution order (conditional statements, jumps, loops, etc.),
  - conditional branches to arbitrary but fixed positions of the statement sequence and
  - unconditional branches to arbitrary but fixed positions of the statement sequence,
  - any number and type of variables.
- The intermediate language is based on what is commonly understood as "assembler code".
- The implementation (in particular, the nomenclature of the statements) of this intermediate language is irrelevant here.

Definition: Structure-preserving transformation

- We speak of a **structure-preserving transformation of** a source language (e.g. Java) into the intermediate language if
  - (exclusively) the statements affecting the execution order are replaced by statement sequences of the intermediate language, such that
    - the execution order of the remaining statements is the same with the same parameterization as in the source language!
  - all other statements are taken over unchanged
  - Transformations that replicate statement sequences or conditional jumps are avoided (no loop unrolling, no optimizations).

Example transformation



Definition: Basic Block (BB)

- A **basic block** denotes a maximally long sequence of continuous statements of the intermediate language,
  - into which the control flow enters only at the beginning and
  - which contains no branches except at the end.

Example

```
a = 10;
b = c / a;
if b > d goto basic block x;
m = 3 * b;
...
```

1 Basic block

next basic block

Definition: Control Flow Graph (CFG)

- A control flow graph of a program P is a directed graph G with

$$G = (N, E, n_{start}, n_{stopp})$$

where

- N is the set of basic blocks in P,
- $E \subseteq N \times N$ the set of edges, where the edges indicate the execution order of two basic blocks (sequential execution or jumps).
- n_{start} the starting block and
- n_{stopp} is the stop block.

Find control flow graph

Example program

```
...
int z=0;
int v=0;
char c = (char)System.in.read();
while ((c>='A') & (c<='Z'))
{
  z++;
  if ((c=='A') || (c=='E'))
  {
    v++;
  }
  c = (char)System.in.read();
}
...
```

Find control flow graph

Step 1: Transform to intermediate language

Step 2: Combine all sequences that end with a branch into one basic block each.

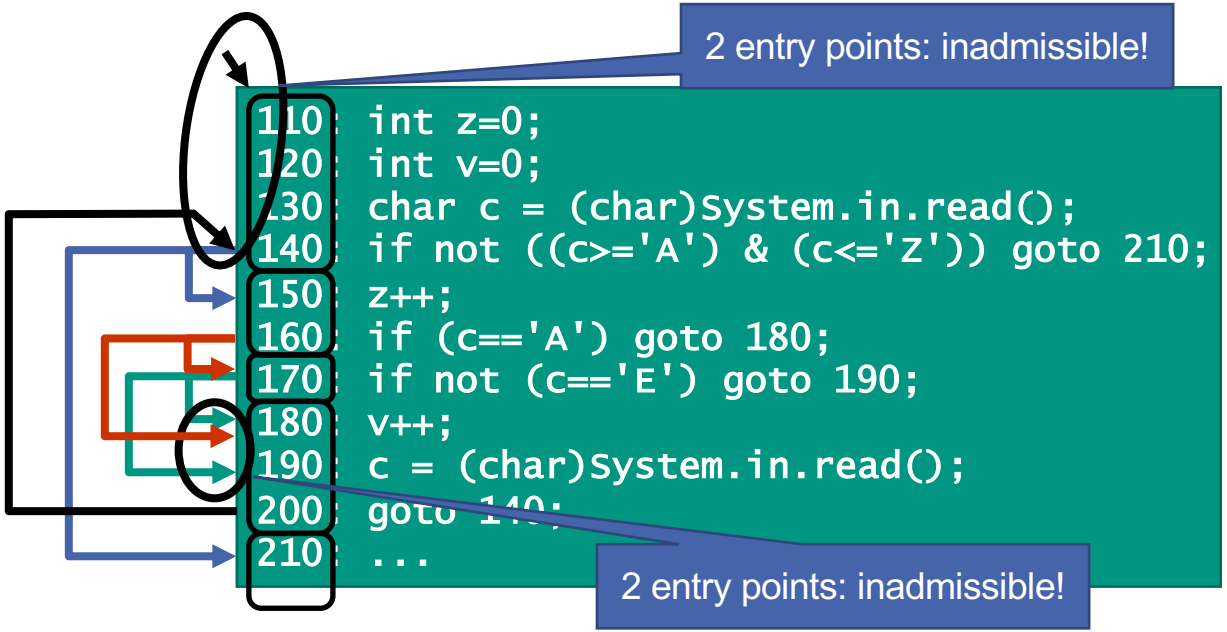
```
110: int z=0;
120: int v=0;
130: char c = (char)System.in.read();
140: if not ((c>='A') & (c<='Z')) goto 210;
150: z++;
160: if (c=='A') goto 180;
170: if not (c=='E') goto 190;
180: v++;
190: c = (char)System.in.read();
200: goto 140;
210: ...
```

Short circuit evaluation must be implemented correclty

Find control flow graph

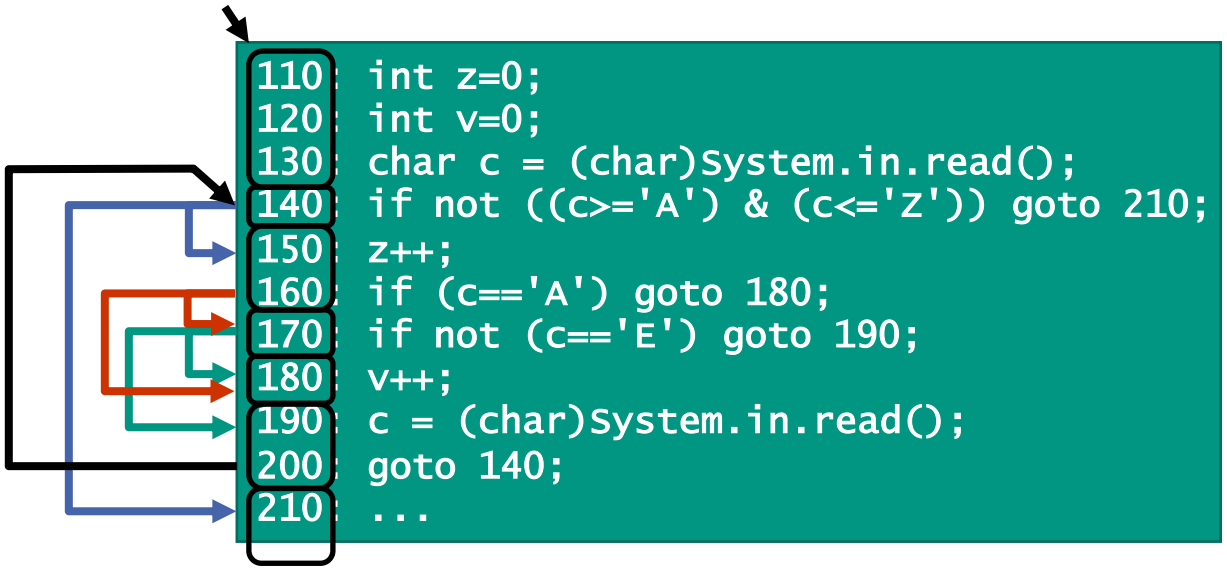
Step 3: Check if blocks entered only at the beginning

Step 4: If not, split blocks at entry points

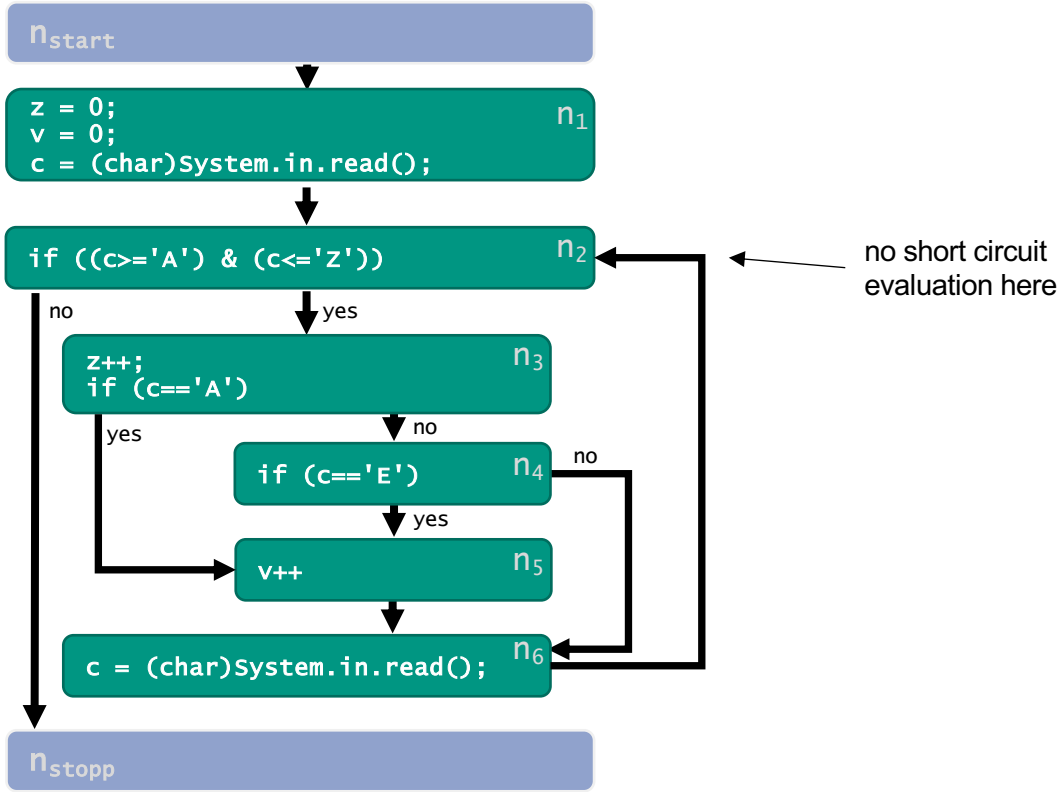


Find control flow graph

Step 3: Check if blocks entered only at the beginning
Step 4: If not, split blocks at additional entry points



Example control flow graph

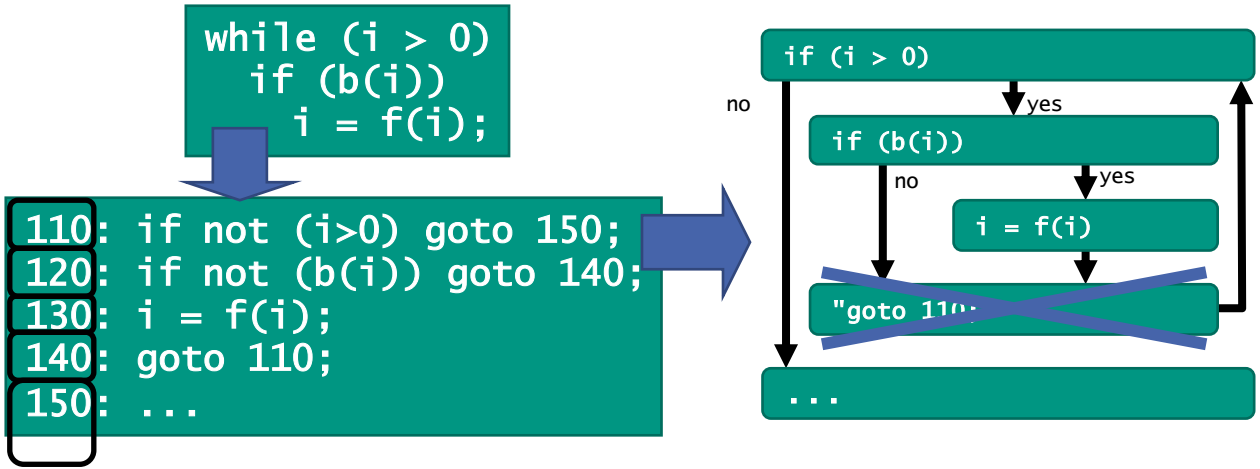


Definitions: Branch, Complete Paths

- An edge $e \in E$ in a CFG G is called a branch. Branches are directed.
- Paths in the CFG that start with the start node n_{start} and end with the stop node n_{stopp} are called **complete paths**.

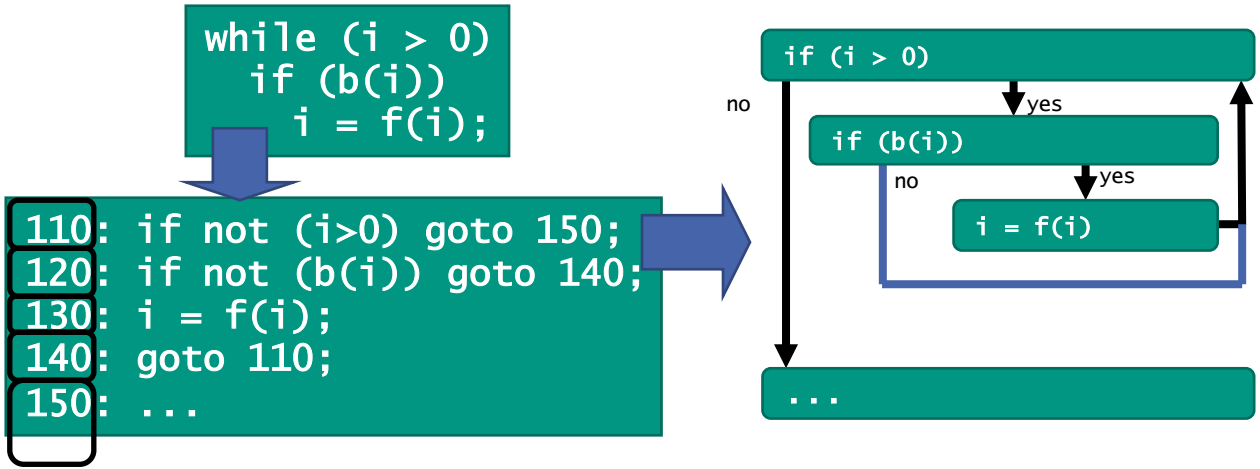
Simplifying control flow graph

- If a split of a block results in a basic block that contains only one unconditional branch, then the CFG can be reduced by removing the branch and using a single edge.



Simplifying control flow graph

- If a split of a block results in a basic block that contains only one unconditional branch, then the CFG can be reduced by removing the branch and using a single edge.



Definition: Statement Coverage

- The test strategy *statement coverage* requires the execution of all basic blocks (and thus all statements) of the program P.
  - C stands for Coverage
  - Metric called C_0

$$C_0 = \frac{\text{Number of unique blocks executed}}{\text{Number of all blocks}}$$

  - Non-reachable program parts can be found.
  - Missing program parts are not detected and not tested.

- The testers task is to find test cases such that by executing them, all blocks are executed at least once. Then the goal of $C_0 = 100\%$ is fulfilled.

Definition: Branch Coverage

- *Branch coverage* requires traversing all branches in the CFG.
  - Metric called C_1

$$C_1 = \frac{\text{Number of traversed, unique branches}}{\text{Number of all branches}}$$

  - more thorough than statement coverage
  - Branches that cannot be executed can be detected
  - Neither combination of branches (paths) nor complexity of conditions are considered
  - Loops are not sufficiently tested
  - Missing branches not detected and not tested
- The testers task is to find test cases such that by executing them, all branches are traversed at least once. Then $C_1 = 100\%$

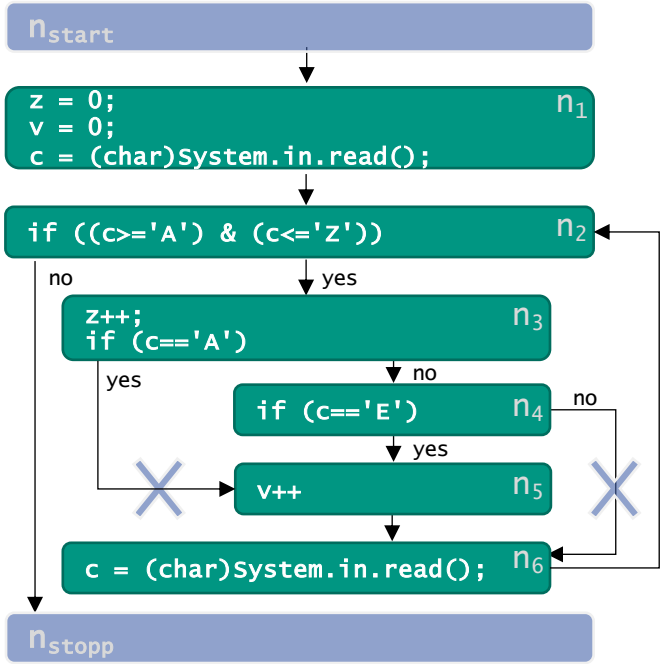
Statement- vs. Branch Coverage

Statement coverage, all Blocks

Example path for test with input “E”:

(n_{start}, n_1, n_2, n_3, n_4, n_5, n_6, n_2, n_{stopp})

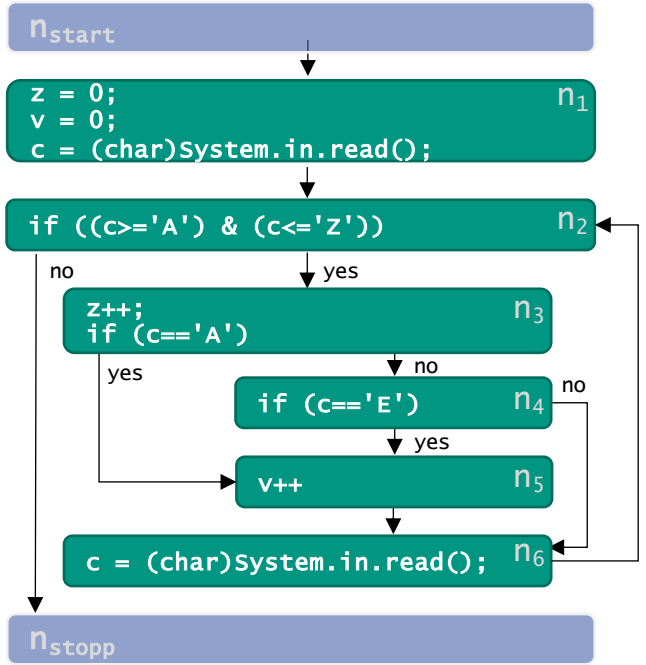
→ branches (n_3, n_5) and (n_4, n_6) not executed



Branch coverage, all Edges

Example path for test with input “EAB”

(n_{start}, n_1, n_2, n_3, n_4, n_5, n_6, n_2, n_3, n_5, n_6, n_2, n_3, n_4, n_6, n_2, n_{stopp})



Boundary-Interior Test

Boundary-interior test works like branch coverage, but treats loops more thoroughly. For each loop:

- Construct a set of test cases that traverse the loop body once.
- Cover all branches within the loop body: Add test cases until achieving branch coverage when traversing the loop body once.
- Construct test cases that traverse the loop body at least twice.
- Cover all branches in the last traversal when traversing at least twice: add test cases until you achieve branch coverage on the last traversal (the other traversals do not matter).
- Example: a loop with a single conditional statement needs test cases for 2+2 paths.

Definition: Path Coverage

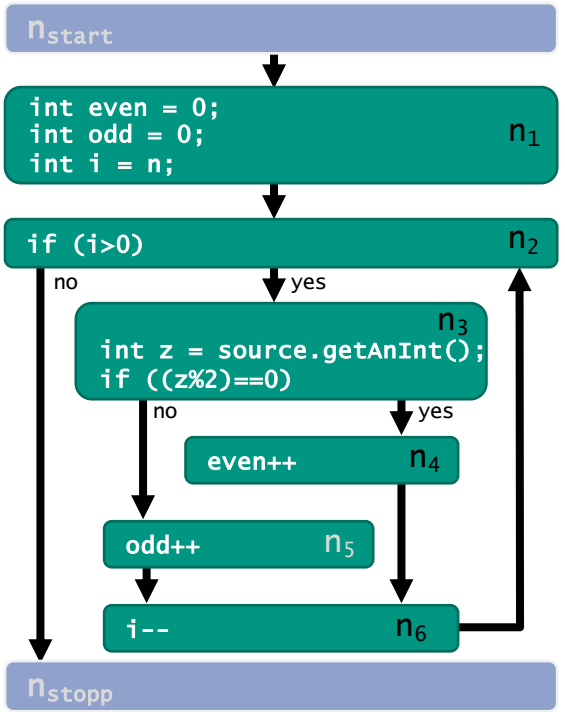
- *Path coverage* requires the execution of all complete paths in the program.
  - Path count grows dramatically with loops.
  - Some paths may not be executable due to mutually exclusive conditions
  - Most thorough CFG test strategy
  - Not practicable because of the enormous number of possible paths.

Path Coverage Effort

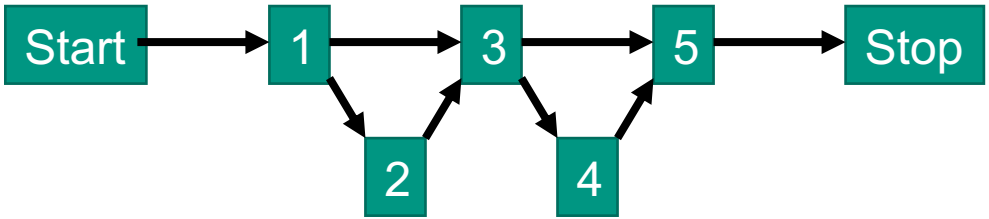
Example code

```
int even = 0;
int odd = 0;
int i = n;
while (i>0) {
  int z = source.getAnInt();
  if ((z%2)==0) even++;
  else odd++;
  i--;
}
```

n	Number of paths
0	I
1	II
2	IIII
…	…
k	2^k



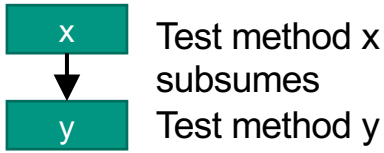
Example for
Statement, branch and path coverage



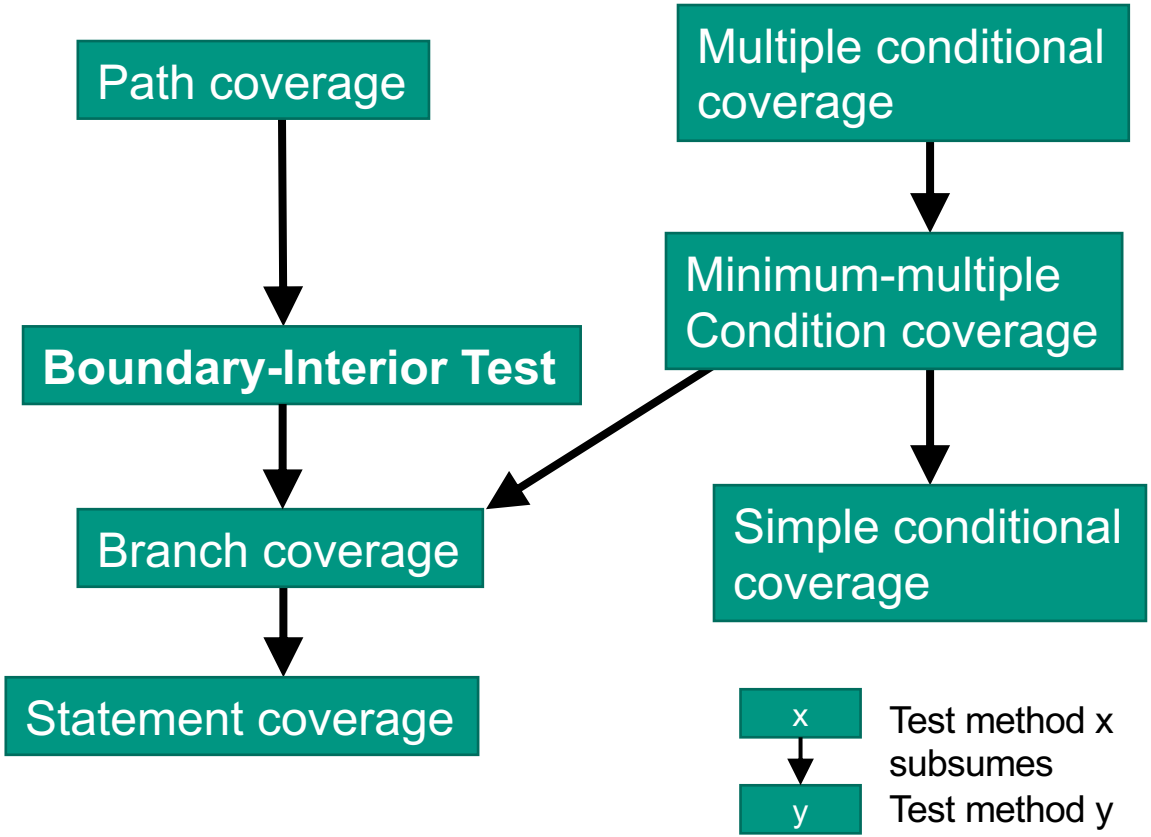
- Statement coverage
  A = { (start, 1, 2, 3, 4, 5, stop) }
- Branch coverage
  Z = A ∪ { (start, 1, 3, 5, stop) }
- Path coverage
  P = Z ∪ { (start, 1, 3, 4, 5, stop) }
  ∪ { (start, 1, 2, 3, 5, stop) }

Definition: Subsumption

- A test procedure for criterion x **subsumes** a test procedure for criterion y if every set of paths that satisfies criterion x also satisfies criterion y.



Hierarchy of CFG test strategies



Definition: Short Circuit Evaluation

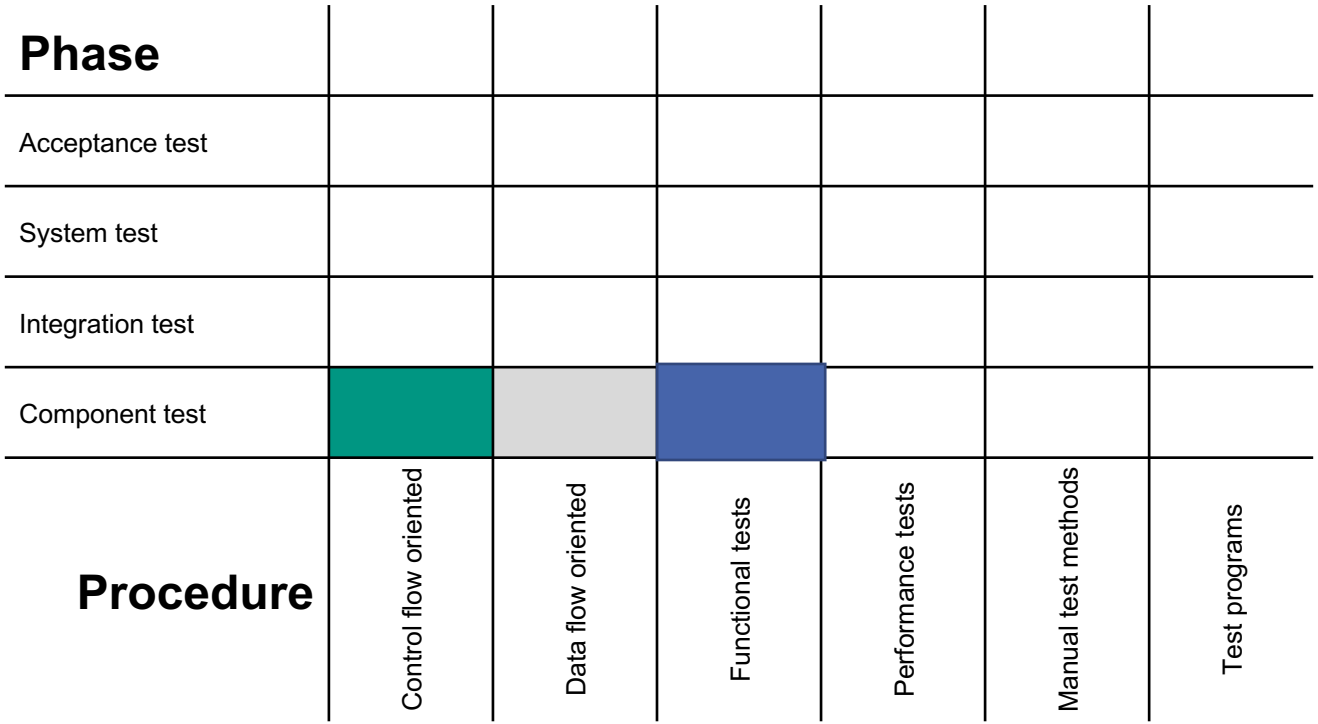
- In **short circuit evaluation,** the evaluation of a compound condition is terminated as soon as the complete result is determined.
- Example:
  - `if ( a!=0 && x/a > 1 ) ...`
    If a = 0, the evaluation is terminated. x/a >1 is not computed (not necessary)
  - `if ( a!=0 || x/a > 1 ) ...`
    If a ≠ 0, the evaluation is terminated. x/a >1 is not computed (not necessary)

  - Short circuit evaluation in Java, e.g. for "&&" and "| |" operators.
  - In Java, the operators "&" and "|" do not perform a short circuit evaluation.
- When constructing the control flow graph, make sure short-circuit evaluation is reflected correctly with an if-cascade.

Summary: CFG test strategies (1)

- **Statement coverage** is the weakest criterion. Each statement must be executed at least once to have a chance to find defects in it. (If you don't execute something at all, then you are guaranteed not to find any defects there).
- **Branch coverage** subsumes statement coverage. Requires that all branches are executed at least once to have a chance to detect defects in all branches.
- **Boundary-Interior Test** loops more extensively than branch coverage (and subsumes it)
- **Path coverage** is most complex and even for small programs with loops not practicable.
- **In general:** the different test strategies are **test completeness criteria**. Example: if C_1 =1 is reached, then the test according to C_1 is complete and you can stop testing if C1 is the desired test criterion.

Overview matrix: What comes next?



Functional Tests

- **Goal**: Test the specified functionality
  - Derive test cases from the **specification**
  - Internal structure of the CUT not considered because unknown to the tester (black box)
  - Advantages:
    - Test cases can be created independently of the implementation (before or at the same time)
    - Avoids "short-sightedness" in the selection process
  - Disadvantage:
    - possible critical paths not known & potentially not tested

Functional Tests

- **Goal**: Test the specified functionality
  - Derive test cases from the specification
  - Internal structure of the CUT not considered because unknown to the tester (black box)
  - Advantages:
    - Test cases can be cre… …he same time)
    - Avoids "short-sightedness" in…
  - Disadvantage:
    - possible critical paths not kno…

Test cases contain
- Input data
- Expected output data/response (target)

Problem:
Let the function under test be $f(x) := x^2$.
→ Obviously, it is not possible to perform a complete functional test.

⇒ As few test cases as possible, but as many test cases as necessary that the probability of finding a defect is high enough.

Functional test case determination procedures

- Functional equivalence classes
- Boundary value analysis
- Random test
- Test of state machines

Functional Equivalence Classes

- **Assumption**: A program/module works the same when processing a value from a certain range as when processing any other value from this range.
- **Example**: We expect a function that correctly computes 42^2 to also correctly compute 40^2
- **Approach**: Decompose the value range of the input parameters and the range of the output parameters into equivalence classes.
  Pick a value from each combination of equivalence classes.

Functional Equivalence Classes

- **Assumption**: A program/module works the same when processing a value from a certain range as when processing any other value from this range.
- **Example**: We expect a function that correctly computes 4^2 to also correctly compute 40^2
- **Approach**: Decompose the value range of the input parameters and the range of the output parameters into equivalence classes. Pick a value from the equivalence classes

Inputs may be outside the domain of the input variables. Out-of-domain inputs test error handling.

Functional Equivalence Classes

Formation of equivalence classes

- Partition ranges starting from a large equivalence class
  - Partitioning along domain boundaries
  - Partitioning along assumed(!) processing method boundaries
- Selection of representatives
  - Let m be the number of equiv. classes of the input parameters and n be the number of equiv. classes of the output parameters.
  - Then ≤ m * n different equiv. classes arise. An arbitrary representative is chosen for testing from each class

Equivalence Classes: Example (1)

- Function to determine the number of days in a month of a given year (according to the Gregorian calendar).
- Input values: **month, year: Integer**

Equiv.Class	Description	Possible values
$EC_{M,31}$	Months with 31 days	1,3,5,7,8,10,12
$EC_{M,30}$	Months with 30 days	4,6,9,11
$EC_{M,Feb}$	Month with 28 or 29 days	2
$EC_{M,Error}$	Invalid entries	Month < 1 or Month > 12
$EC_{Y,LY}$	Leap years	1904, 2000, 2012, ...
$EC_{Y,Not-LY}$	Non-leap years	1901, 1900, ...
$EC_{Y,Error}$	Invalid entries	Year < 0

Equivalence Classes: Example (2)

- Derivation of test cases (valid inputs):

Equivalence class	Input		Expected output
	Month	Year	
$EC_{M,31}$ and $EC_{Y,Not\text{-}LY}$	7	1900	31
$EC_{M,31}$ and $EC_{Y,LY}$	7	2000	31
$EC_{M,30}$ and $EC_{Y,Not\text{-}LY}$	6	1900	30
$EC_{M,30}$ and $EC_{Y,LY}$	6	2000	30
$EC_{M,Feb}$ and $EC_{Y,Not\text{-}LY}$	2	1900	28
$EC_{M,Feb}$ and $EC_{Y,LY}$	2	2000	29

Equivalence Classes: Example (3)

- Derivation of test cases (invalid inputs):

Equivalence class	Input		Expected output
	Month	Year	
$EC_{M,Error}$ and $EC_{Y,Not\text{-}LY}$	-1	1900	Error
$EC_{M,Error}$ and $EC_{Y,LY}$	13	2000	Error
$EC_{M,30}$ and $EC_{Y,Error}$	6	-1	Error
$EC_{M,31}$ and $EC_{Y,Error}$	7	-1	Error
$EC_{M,Feb}$ and $EC_{Y,Error}$	2	-42	Error
$EC_{M,Error}$ and $EC_{Y,Error}$	-1	-42	Error

Boundary Value Analysis

- Further development of functional equivalence classes
- Observations:
  - Off-by-one: Barely missed is still off target
  - Test cases that cover the boundaries of the equivalence classes and their immediate surroundings are particularly effective

⇒ In addition to elements from the equiv. class, also use those on and around the boundary (approach from both sides).

Example for months: 0 and 1, 12 and 13.

- Special candidates: 0, 1, MAX_INT, NaN, ...

Boundary values for intervals

- open and closed intervals:
  - (a-b): the boundaries a and b are not in the interval.
  - [a-b]: the boundaries a and b are included in the interval
- Idea: choose a test value that is in the interval and another outside the interval, close to the boundary. This means:
- For `(a...` choose a (not in the interval) and `a+1` (int) or `a+delta` (float)
- For `[a...` choose a (in the interval) and `a-1` (int) or `a-delta` (float)
- `delta` is a small value compared to a

- Analogously for the upper end of an interval

- Example: for `[5-10)` choose boundary values `4,5,9,10.`
  plus a representative for the equivalence class somewhere in the middle

Random Testing

- Testing the function with random test cases
- **Observation**: testers tend to create test cases that are considered obvious during implementation (a kind of operational blindness)
- **Advantage of random test**: non-deterministic test method that treats all test cases equally
- Useful as a supplement to other procedures
- Useful if you want to create many test cases (e.g. for sorting procedures) and the verification of correctness can be done automatically.
- Useful as an additional criterion (e.g. in addition to C_1 or equivalence classes).

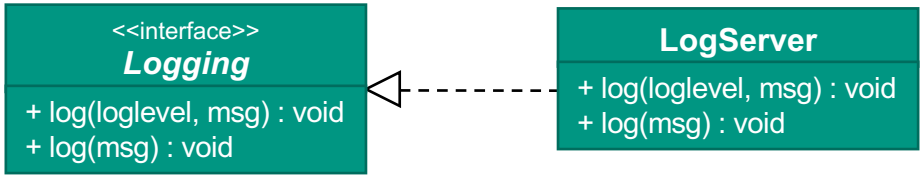
Use of test helpers (1)

- Objects do not live in isolation; objects work together in an application.
- Collaboration results in testing dependencies.
- How can individual properties be tested regarding these dependencies?
  - What exactly should be tested ?
  - What problems are to be tested ?
- How to write tests that involve dependencies ?

Use of test helpers (2)

- If classes depend on each other, classes not yet implemented can be replaced by test helpers (stubs, dummies or mockups).
- Stubs or dummies represent the implementations that are still missing.
- Stubs and dummies are gradually replaced by real implementations; mockups will continue to be useful for future testing.

Example: dummy (1)

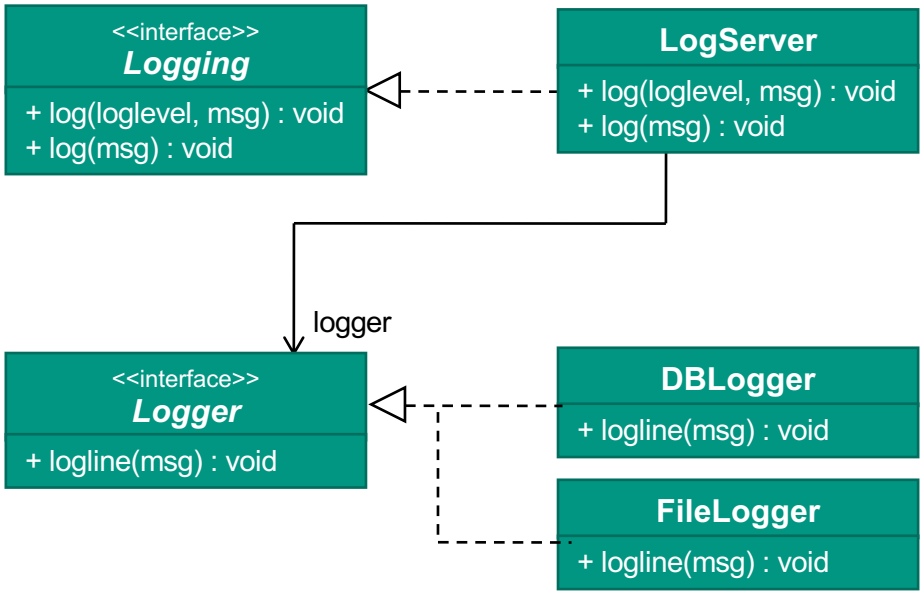


- Provide system-wide logbook management
- `log(loglevel, msg)` logs messages with different priority
- `log(msg)` logs messages with default priority "2".

Example: Dummy (2)

- **Problem**: Where does the LogServer log to ?
  - For initial purposes, the file system is provided
  - Relational database could come later
  - Both options not optimal for testing the LogServer.
- **Solution**:
  - Hide logging medium behind interface `logger`

Example: Dummy (3)

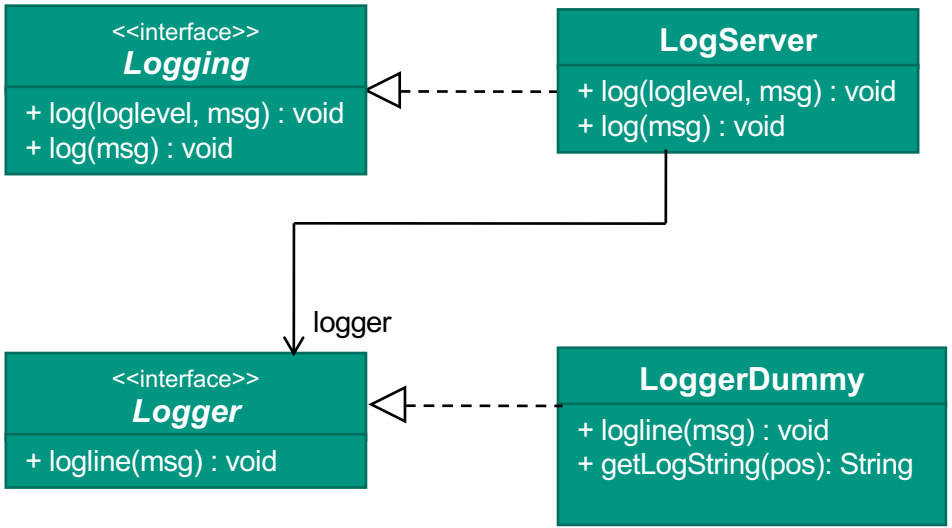


Example: Dummy (4)

- The most important feature of the logger is explicit:
  - line-by-line logging
- The target medium is hidden behind interface: `DBLogger`, `FileLogger`
- `LogServer` contains one instance of the logger.

- What does this do for testing now ?

Example: Dummy (5)



Example: Dummy (6)

- Implements the logger interface for `LogServer` and
- `getLogString` for the test
  - `getLogString(pos)` provides the message number pos.

- Now we can write a test of the LogServer:

```
public void testSimpleLogging() {
   LoggerDummy logger = new LoggerDummy();
   Logging logServer = new LogServer(logger);

   logServer.log(0, "First line");
   logServer.log(1, "Second line");
   logServer.log("Third line");

   assertEquals("(0): First line",
       logger.getLogString(0));
   assertEquals("(1): Second line",
       logger.getLogString(1));
   assertEquals("(2): Third line",
       logger.getLogString(2));
}
```

Example: Dummy (7)

- LogServer, which is independent of logging medium.
- Interface for testing our LogServer
- Checking the `LogServer` without access to the implementation.
- **Question**: What can be improved here ?
- **Answer**: Test code can move to `LoggerDummy`. It thus becomes a **mock object**

Example: Mock object (Imitation) (1)

```
public class LoggerImitation implements Logger {
   private List expectedLogs = new ArrayList();
   private List actualLogs = new ArrayList();

   public void addExpectedLine(String logString) {
     expectedLogs.add(logString);
   }
   public void logLine(String logLine) {
     actualLogs.add(logLine);
   }
   public void verify() {
     if (actualLogs.size() ! = expectedLogs.size()) {
       Assert.fail("Expected"  + expectedLogs.size()
         +"log entries but encountered "+actualLogs.size());
     }
     for (int i = 0; i < expectedLogs.size(); i++){
       String expLine = (String) expectedLogs.get(i);
       String actLine = (String) actualLogs.get(i);
       Assert.assertEquals(expLine, actLine);
     }
   }
}
```

- Sets the input expected by LogServer.
- Called at the beginning of a test.

- Checks the number of log messages and their content.
- Called at the end of a test.

Example: Mock object (3)

- Test code:

```
LoggerImitation logger;
@Test
public void testSimpleLogging() {
  logger.addExpectedLine("(0): First line");
  logger.addExpectedLine("(1): Second line");
  logger.addExpectedLine("(2): Third line");
  logServer.log(0, "First line");
  logServer.log(1, "Second line");
  logServer.log("Third line");
  logger.verify();
}
```

- Failures are reported only when `verify()` is called
- With more complicated behavior, checking becomes more complex

Example: Mock object (4)

- Modified `loggerImitation,` with checking moved to `logLine`

```
public void addExpectedLine(String logString) {
    expectedLogs.add(logString);
  }


public void logLine(String logLine) {
  Assert.assertNotNull(logLine);
  if (actualLogs.size() >= expectedLogs.size()) {
    Assert.fail("Too many log entries");
  }

  int currentIndex = actualLogs.size();
  String expectedLine =
    (String) expectedLogs.get(currentIndex);
  Assert.assertEquals(expectedLine, logLine); // check against expected line
  actualLogs.add(logLine);
}


public void verify() {
  if (actualLogs.size() < expectedLogs.size()) { //check of size needed only in verify
    Assert.fail("Expected " + expectedLogs.size()
     + "log entries but encountered " + actualLogs.size());
  }
}
```

Example: Mock object (5)

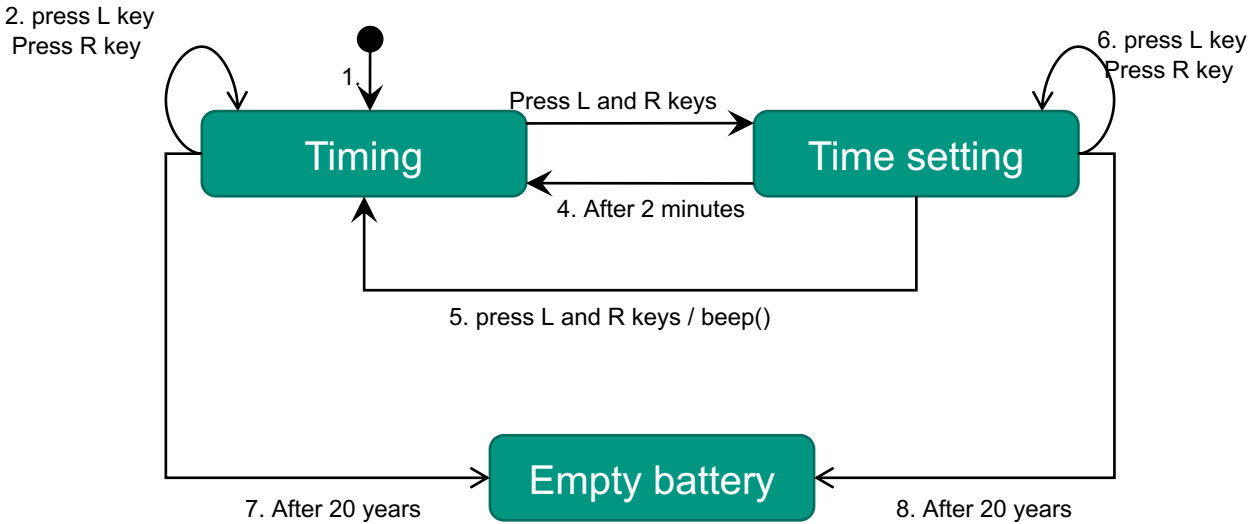
- mock object provides a `setExpected` method for each input, or `addExpected` if multiple calls are expected
- mock object contains test code in `verify`
- mok object checks parameters, as well as sequence of method calls (protocol analysis)
- **Attention**: It is not the mock object that is being tested, but the use of the mock object.

Test of state machines

- If a component has an internal state, test cases can be derived from the state transitions
- **Goal**: pass through all transitions at least once
- **Attention**:
  - Covering all transitions does not guarantee a complete test (cf. branch covering)
  - Instrumentation of the component may be necessary to monitor the test sequences

Test of state machines:
Example: Clock with two buttons

- Given: Clock with two keys "L" and "R

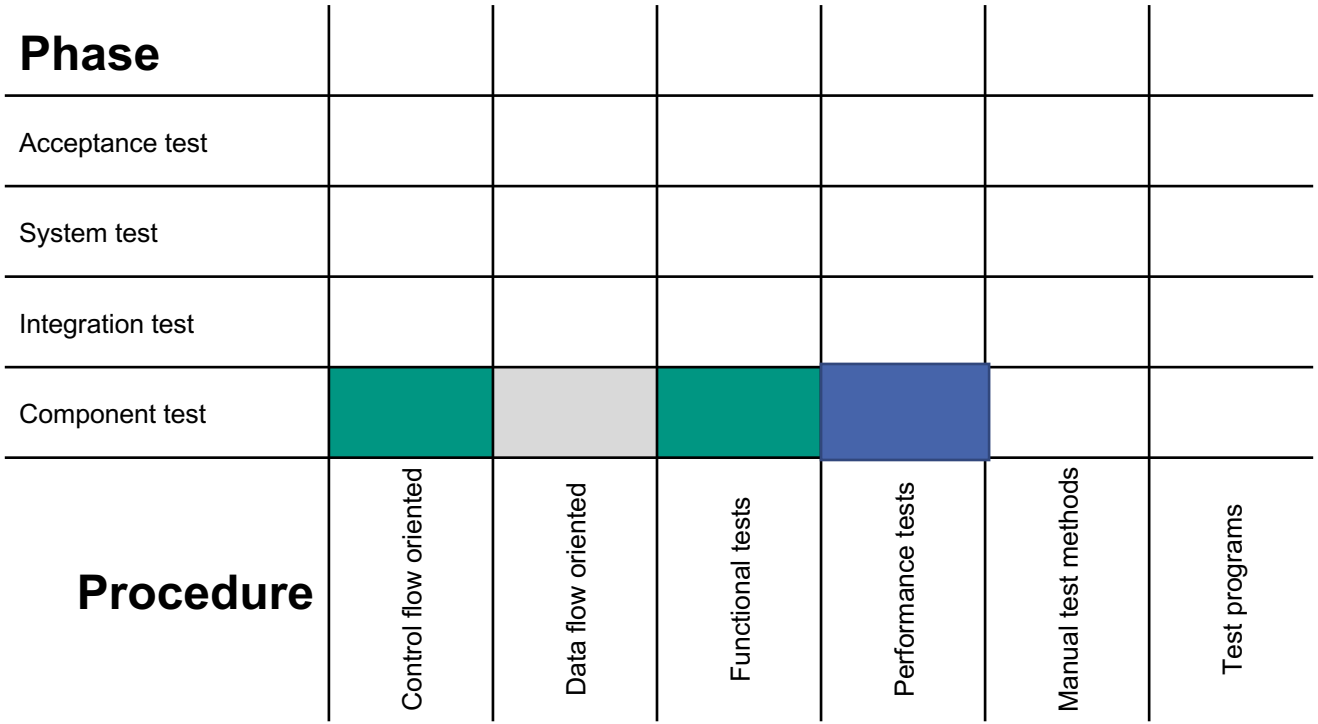


Test of state machines: Example: Clock with two buttons

Test case	Tested transitions	Expected condition
Initial state	1.	Timing
Press L key	2.	Timing
Press L and R keys simultaneously	3.	Time setting
Wait 2 minutes	4. timeout	Timing
Press L and R keys simultaneously	3.	Time setting
Press L and R keys simultaneously	5.	Timing
Press L and R keys simultaneously	3.	Time setting
...	...	...

There are more test cases.

Overview matrix: What comes next?



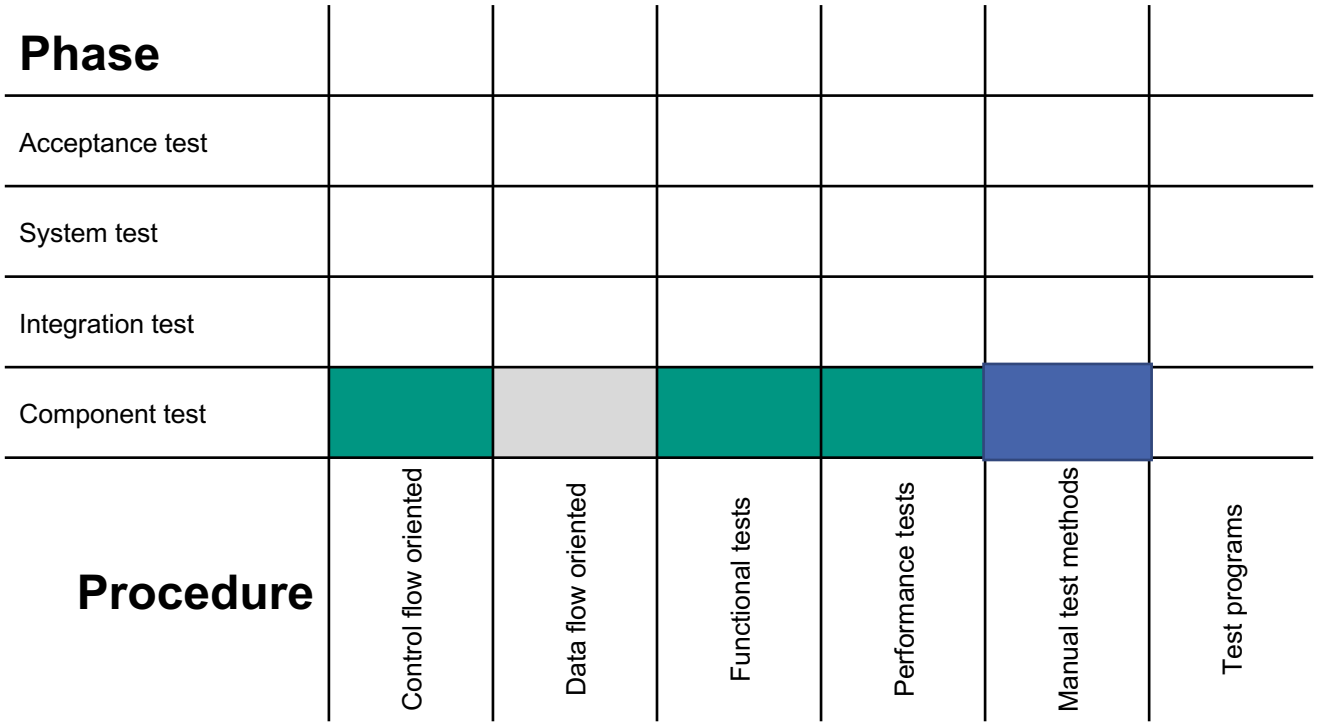
Performance tests: Load tests

- Tests the system/component for reliability and compliance with the specification near the **specified limits.**
  - Can the system serve the required number of users?
  - Can required response times be met?
  - Can a load be handled for any length of time?
  - What is the utilization of the expected bottlenecks?
  - Are there any unexpected bottlenecks?

Performance tests: Stress test

- Tests the behavior of the system **when exceeding the** defined limits
  - What is the performance behavior under overload?
    - Thrashing (page flutter in virtual memory system)
    - Exponential increase in response time
    - Standstill
  - Does the system return to the defined behavior after the overload has decreased?
    - How long does it take?

Overview matrix: What comes next?



Manual Testing Methods

- Only way to check the semantics
- Expensive (up to 20% of development costs)
- Time must be planned for examination
- Psychological pressure when an individual's work is reviewed by a group

→ Consideration of such social conflict potential in the inspection process.

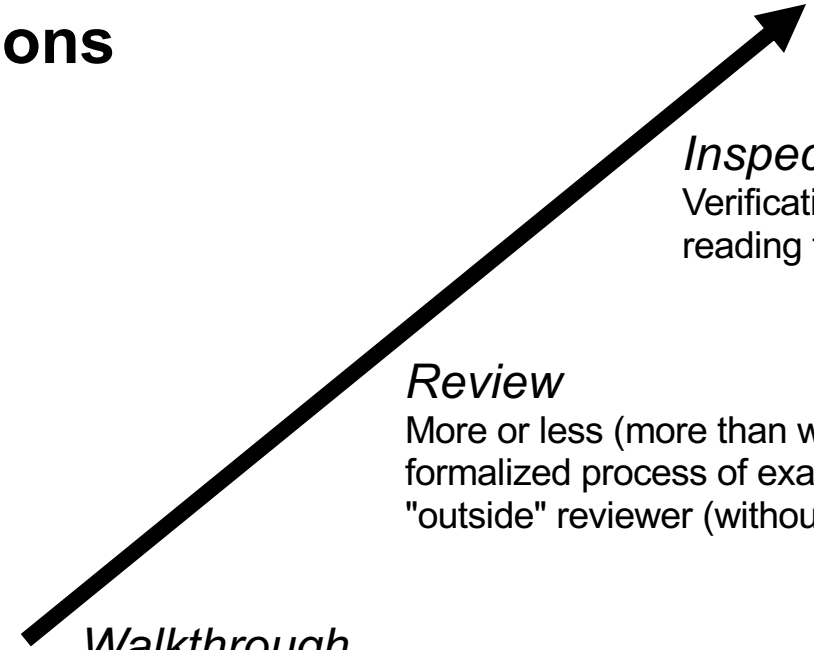
  - The work, not the person, is assessed
  - Regularly review the work of all employees
  - If possible, no customers/higher managers present

Software Inspections (Overview)

- Multiple inspectors (2 to 4) independently examine the same artifact (software document)
- Defects found are written down and discussed in a meeting
- Focus is to identify problems; solution takes place later
- Structured process
  - Roles and forms
  - Checklists and perspectives

} Adaptation to document type

Definitions



Inspection
Verification by means of checklists or reading techniques

Review
More or less (more than walkthrough, less than inspection) formalized process of examining of written documents by an "outside" reviewer (without a checklist)

Walkthrough
The developer walks one or more colleagues through a piece of code or design that he or she has written. Colleagues ask questions and make comments about style, defects, adherence to development standards, and other problems. (according to ANSI/IEEE 729-1983)

Definition: Inspection

- Inspection is a formal quality assurance technique in which requirements, design, or code are examined in depth by a person or group of persons different from the author. The purpose is to find errors, violations of development standards, and other problems.

(according to ANSI/IEEE Standard 729-1983)

Advantages and Disadvantages of Inspections

- Advantages
  - Applicable to all software documents: requirements, specifications, drafts, code, test cases, ...
  - Can be carried out at any time and at an early stage
  - Very effective in industrial practice
- Disadvantages
  - manual process
  - consume time of several employees
  - thus, expensive at first
  - "static" (as opposed to testing).

IBM
Hewlett Packard
Motorola
Siemens
NASA

Figures on Benefits and Costs

- Usually more than 50 percent of all detected defects are found in inspections (up to 90 percent). Rule of thumb: 50-75% for design defects (empirical).
- Ratio of troubleshooting costs through inspection versus defects identified through testing often between 1:10 and 1:30
- Return on investment (ROI) is often quoted at well over 500 percent
- Further study: Fagan, M., *Design and Code Inspections to Reduce Errors in Program Development*, IBM Systems Journal 15, 3 (1976): 182-211.

Why inspections are effective

- Observation: “Four eyes see more than two."
- Inspectors have distance from the software document
- Inspectors provide experience
- Joint discussion of the identified problem points (engl. *issues*)
  - Is it a defect?
  - Is it a weakness?
  - Is it merely “inelegant"?
  - Discussion leads to common insight, trains participants and newbies

Phases of an inspection

1. Preparation
2. Individual inspections
3. Group session
4. Follow-up
5. Process improvement

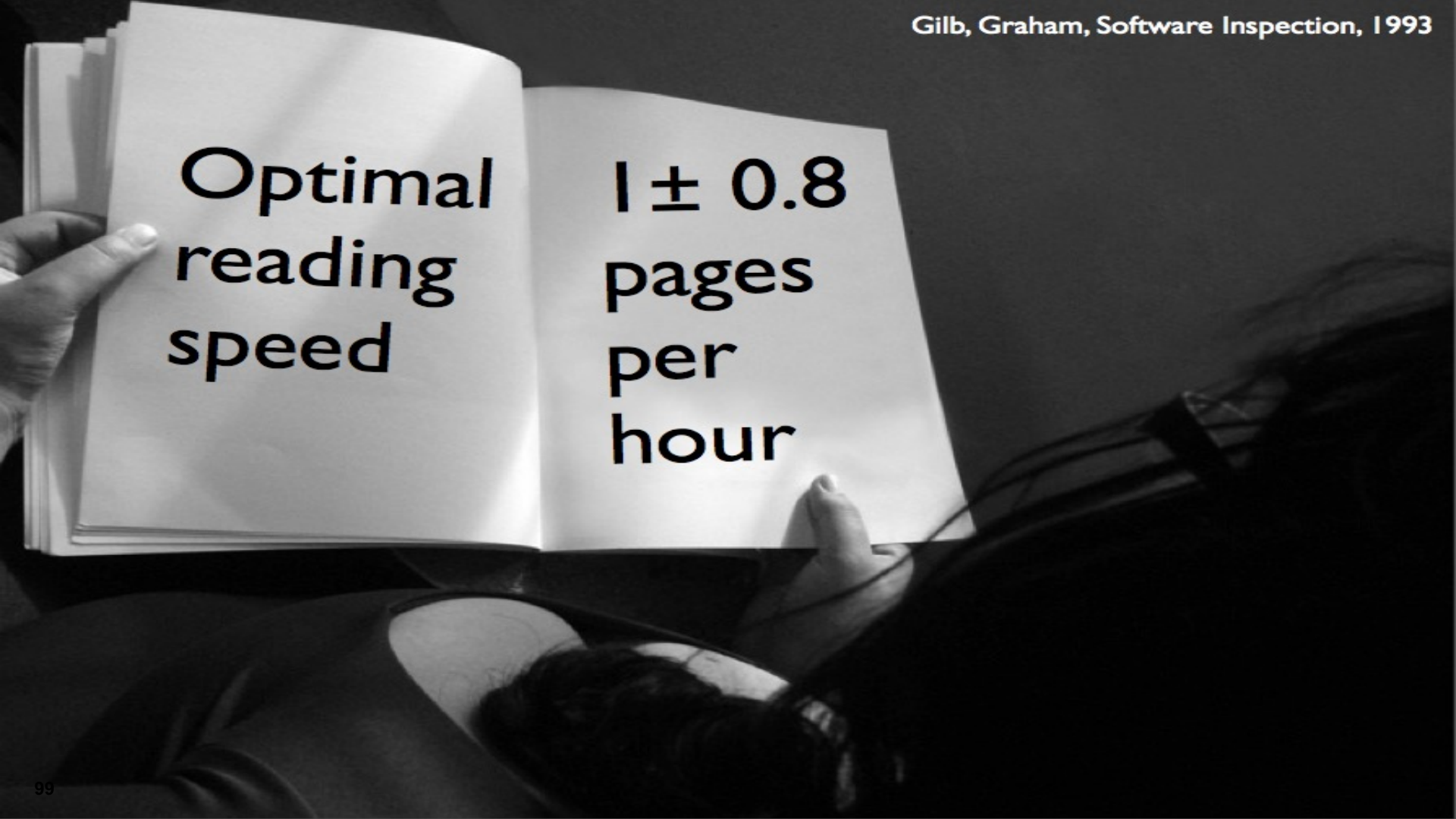
1. Preparation

- Determine participants and their roles
- Prepare documents and forms
- Determine reading techniques or checklists
- Plan the schedule
  - Schedule individual inspections and group session
  - If necessary, partition the document: A group session should not last longer than 2 hours (concentration on the important, active participation).

2. Individual Inspections

- Each inspector checks the document alone
  - about 1 page per hour per inspector
  - Results must be able to be reviewed in 1 group session
    ⇒ max. 1-4 pages net per inspection
- Use agreed-upon reading technique or checklists
- Inspectors write down all issues and their exact locations in the document
- Issues: (potential) defects, suggestions for improvement, questions.

Gilb, Graham, Software Inspection, 1993



3. Group Session (Duration: 2h)

- Collect individually found issues
- Discuss every single issue
- Clarify questions about the document
- Collect other problem points found during the discussion
- Collect suggestions for improvement in the implementation of inspections

Alternative opinion: **Do not discuss!**

4. Follow-up

- List with all issues is forwarded to the author of the document
- Editor (author) identifies actual defects and classifies them
- Editor initiates change of the document
- All issues are processed
- Their completion is checked
- Estimate residual defects
  - rule of thumb: # undetected defects ≈ # detected defects
  - 1/6 of the corrections is faulty or causes a new defect
- Document will be released if #estimated defects < N

5. Process improvement (only occasionally)

- Adapt checklists and perspectives
- Develop standards for documents
- Adapt defect classification scheme
- Improve forms
- Improve planning and implementation

Roles

- **Inspection manager**: manages all phases of the inspection
- **Facilitator**: leads the group meeting (usually the head of inspection).
- **Inspectors**: check the document
- **Secretary**: records defects in the group meeting.
- **Editor**: classifies and fixes the defects (mostly the author)
- **Author**: has written the document

Defect Classification

- *major* or *minor*
- *Defect*, *suggestion*, or *question*
- Defect type, e.g. according to NASA SEL (*orthogonal defect classification*).
  - A: ambiguous information
  - E: extraneous information
  - II: inconsistent information
  - IF: incorrect fact
  - MI: missing information
  - MD: miscellaneous defect

Reading Techniques - Checklists: Example (1)

- A possible checklist for Java code could be structured according to the following defect classes:
  1. Variable, attribute, and constant declarations
  2. Method definitions
  3. Class definitions
  4. Data references
  5. Calculations
  6. Comparisons
  7. Control flow
  8. Input/Output
  9. Module interface
  10. Comments, documentation
  11. Code formatting
  12. Modularity
  13. Memory usage
  14. Execution speed

Checklist by Christopher Fox, 1999.

Reading Techniques - Checklists: Example (2)

1. Variable, attribute and constant declarations
   1. Are the variable, attribute and constant names meaningful?
   2. Do the identifiers comply with the programming guidelines?
   3. Are there variables or attributes with similar names (risk of confusion)?
   4. Has each variable and attribute been typed correctly?
   5. Has each variable and attribute been initialized correctly and before use?
   6. [...]
2. Method definitions
   1. Are the method names meaningful?
   2. Do the method names comply with the programming guidelines?
   3. Is each method parameter checked before use?
   4. Does the method return the correct value with each `return`?
   5. Are there static methods which should not be static (and vice versa)?
   6. [...]

Additional test questions can be added.

Reading Techniques - Checklists: Example (3)

3. Class definitions
    1. Does each class have appropriate constructors?
    2. Does a class have instance variables that fit better into a superclass?
    3. Can the class hierarchy be simplified?
    4. Are classes too big? (no so-called God classes)
    5. [...]
4. Data references
    1. For each reference to a field: is each index within the allowed limits?
    2. For each object reference or reference to a field: is the value guaranteed never `NULL`?
    3. [...]

Reading Techniques - Checklists: Example (4)

5. Calculations
    1. Are there calculations with different data types (integer/floating point number conversion)?
    2. For expressions with more than one operator: Is the order of evaluation correct?
    3. [...]
6. Compare
    1. For each Boolean comparison: is the correct condition checked?
    2. Are the correct relational operators used?
    3. Has every Boolean expression been simplified by factoring the negations into the expression?
    4. Is every Boolean expression correct?
    5. [...]

Reading Techniques - Checklists: Example (5)

7. Control flow
   1. For each loop: was the best loop construct chosen?
   2. Do all loops terminate?
   3. If there are multiple exists from a loop: Are they necessary and are they handled correctly?
   4. Does every `switch` statement have a default case?
   5. [...]
8. Input/Output
   1. Are all files opened before use?
   2. Are all files closed after use?
   3. Are all exceptions (`IOException`) handled correctly?
   4. [...]

Reading Techniques - Checklists: Example (6)

9. Module interfaces
    1. Do the number, order, typing, and value of parameters match the method definition for each method call?
    2. Are the units of the parameters (kilometers vs. miles) correct?
    3. [...]
10. Comments, Documentation
    1. Does each method, class, file have an appropriate comment?
    2. Does every attribute, variable and constant have a comment?
    3. Do the comments match the commented code?
    4. Do the comments help to understand the code?
    5. Javadoc documentation complete and understandable?
    6. [...]

Reading Techniques - Checklists: Example (7)

11. Code formatting
    1. Is uniform indentation and code formatting used ?
    2. For each method: Is it no more than 60 lines long?
    3. For each class: Is it no longer than 600 lines?
    4. [...]
12. Modularity
    1. Is there a low coupling between different classes?
    2. Is there a high level of cohesion within each class?
    3. Have Java class libraries been used where it makes sense?
    4. [...]

Reading Techniques - Checklists: Example (8)

13. Memory usage
    1. Are all fields large enough?
    2. Are object and field references set to `NULL` when they are no longer used?
    3. [...]
14. Execution speed
    1. Can better data structures or more efficient algorithms be used?
    2. Must a value be recalculated each time or is it worthwhile to store the value temporarily?
    3. Can a calculation be pulled out of a loop?
    4. Should a loop be unrolled?
    5. [...]

Reading techniques - perspectives or scenarios (1)

- Software can also be inspected from a specific perspective instead of using checklists. In this case, the aspects of this perspective are specifically addressed.
- Typical perspectives
  - Maintainance
  - Code analysis
  - Tester
  - Designer
  - Quality assurance
- Perspective-based reading techniques consist of:
  - Explanation of the perspective and its goals
  - Instructions for working through the document
  - List of questions about the document

Reading techniques - perspectives - example (1)

Maintenance perspective example

Assume you are inspecting an operation from the perspective of a maintainer. The main goal of a maintainer is to ensure that the collaboration diagram is written in a way that can be easily changed and maintained. High quality, therefore, means the conformance to specified design guidelines (low coupling, high cohesion) and the minimization of complexity.

Locate the collaboration diagram and the pseudocode description for the operation. Examine the diagram and the descriptions to identify points that diverge from good design practice.

While following the instructions answer the following questions:

1. Are there any ways in which the number of objects or the number of messages could be reduced?
2. Are there any cycles of messages in the collaboration diagram?
3. Is there any way in which the control structure of the operation could be simplified?
4. Do the messages entering an object indicate the possibility of low cohesion (unrelated messages)?
5. Is there a particularly high number of messages between a pair of objects?

Reading techniques - perspectives - example (2)

Code Analysis Perspective Example

Assume you have the role of a code analyst. As a code analyst you have to ensure that the right functionality is implemented in the code.

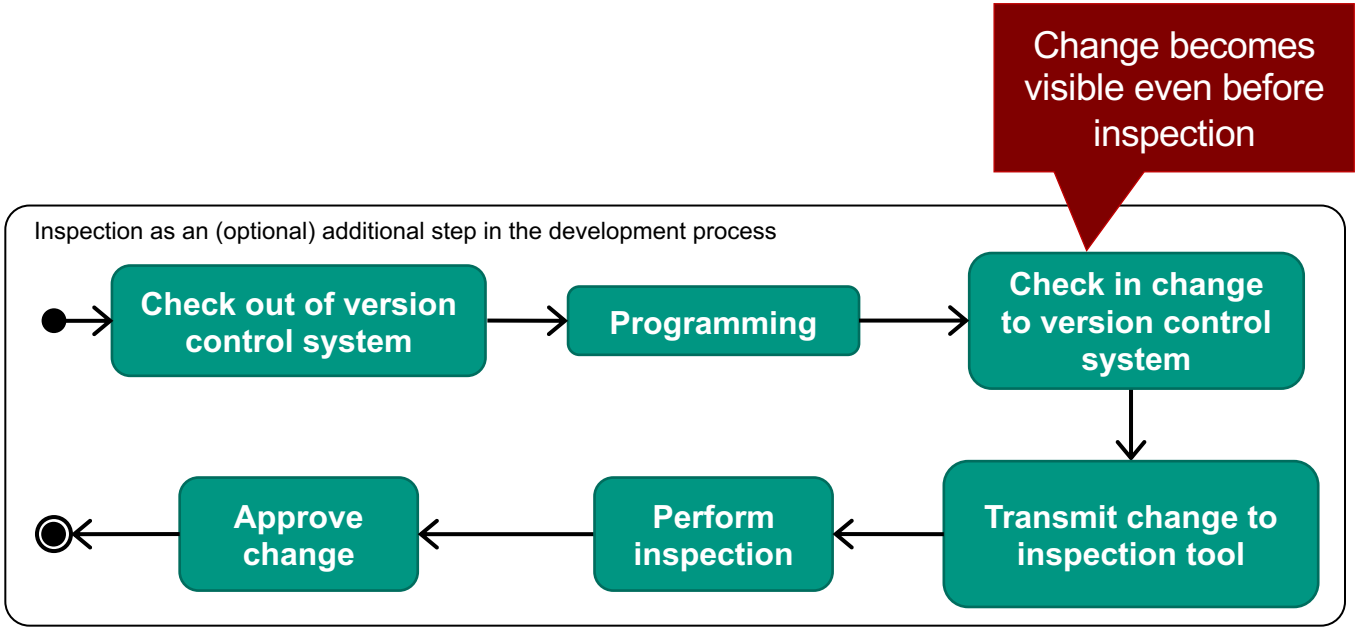
In doing so, take the code document and determine the functions that are implemented in this code module.
Determine the dependencies among these functions and document them in the form of a call graph.
Starting with the functions at the leaves of the call graph, determine the implemented operation of each function in the following manner. Identify (sequences of) assignment operations and highlight them. Determine the meaning of these (sequences of) assignment operations. Combine the (sequences of) assignment operations by taking into account conditions and loops. Determine the meaning of the larger structures.
Repeat this until you have determined the operation that is implemented in a function. Document the operation of each function.
Check for each function, whether your description matches the description that is given in code comments and the description in the specification. If differences exist, check whether there is a defect. Document each defect you detect on the defect report form.

While following the instructions, ask yourself the following questions:

1. Does the operation described in the code match the one described in the specification?
2. Are there operations described in the specification that are not implemented?
3. Is data (i.e., constants and variables) used in a correct manner ?
4. Are all the calculations performed in a correct manner?
5. Are interfaces between functions used correctly?

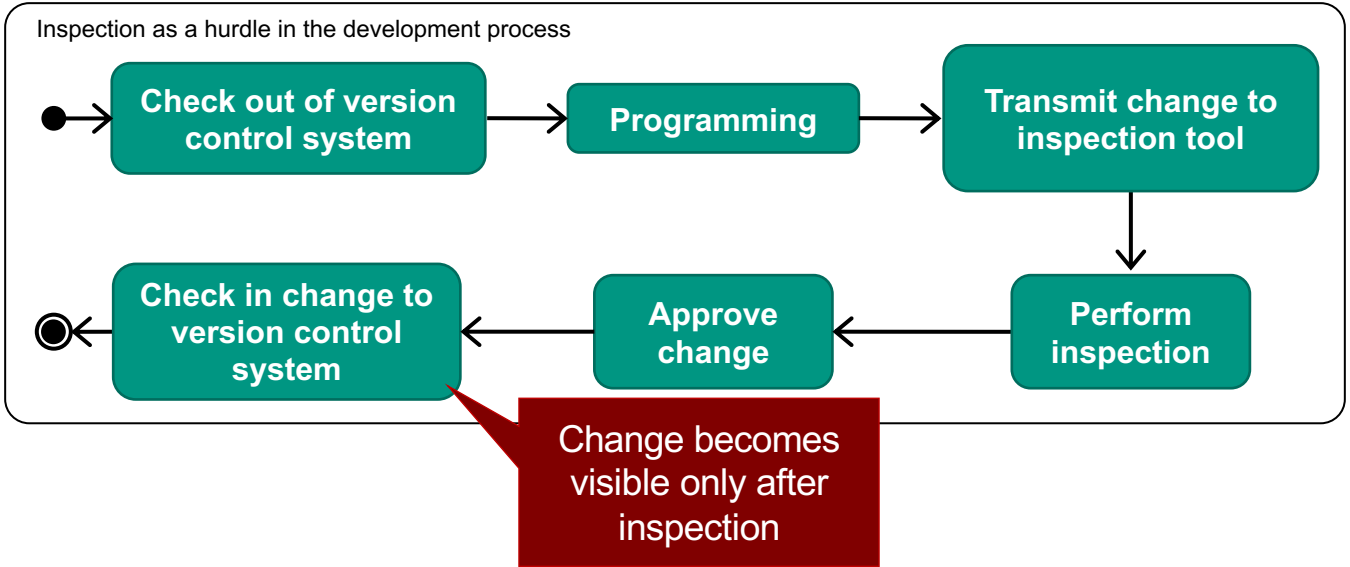
Inspection support tools (1)

- Inspection as an optional step
  - Agile Review
  - Jupiter
  - Phabricator
  - Review Board
  - RhodeCode

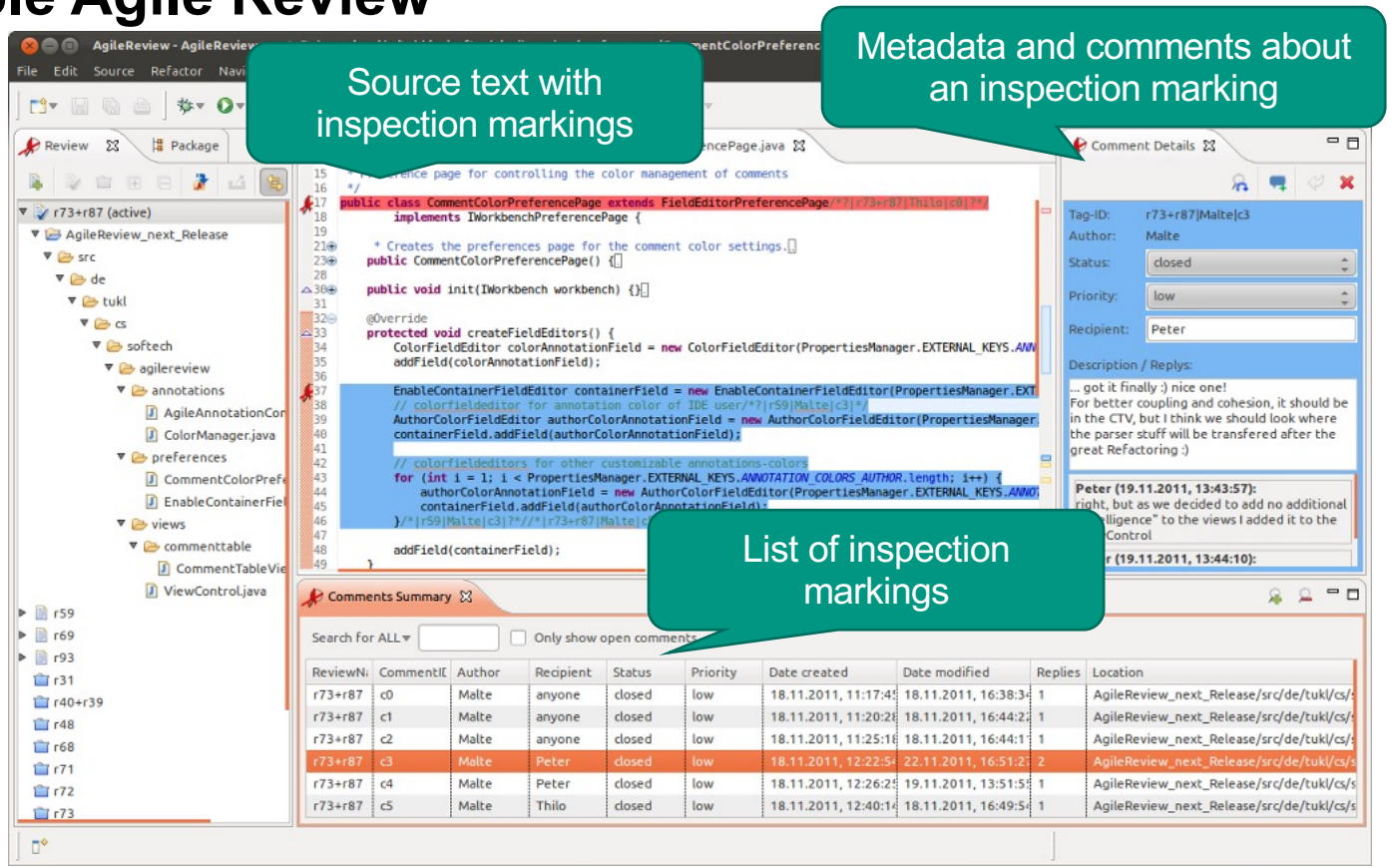


Inspection support tools (2)

- Inspection as a (technically) mandatory part of the development process
  - Gerrit (e.g. https://git.eclipse.org/r/#/c/14311/)
  - Rietveld (e.g. https://codereview.appspot.com/10695044/)



Inspection support tools (3)
Example Agile Review

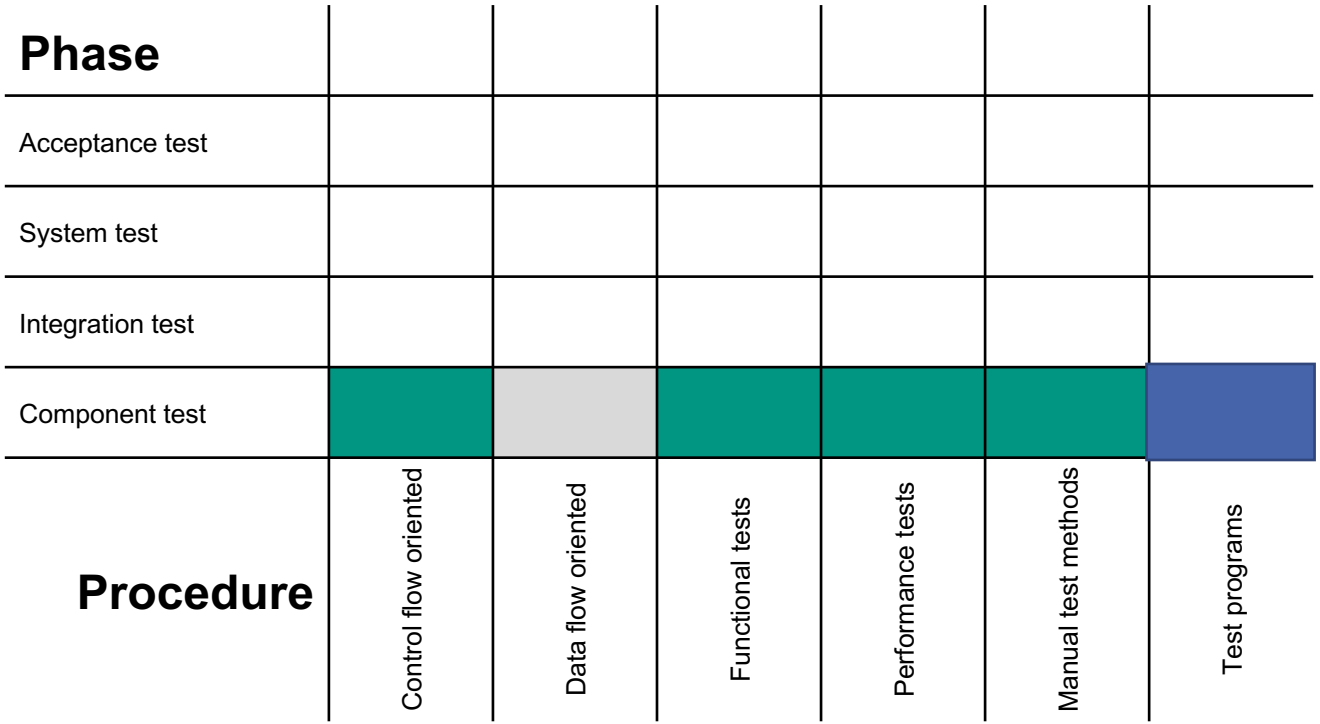


Inspection support tools (4)
Example Agile Review

- Eclipse plugin
- Each inspection is project-based and consists of a set of source code components for which there are comments and responses.
- An inspection is maintained in the version control system like the inspected project.
- An inspection evolves on the version control system timeline, as does the project.

- Other inspection tools (similar functionality): Gerrit, Jupiter, Phabricator, RhodeCode, ReviewBoard

Overview matrix: What comes next?



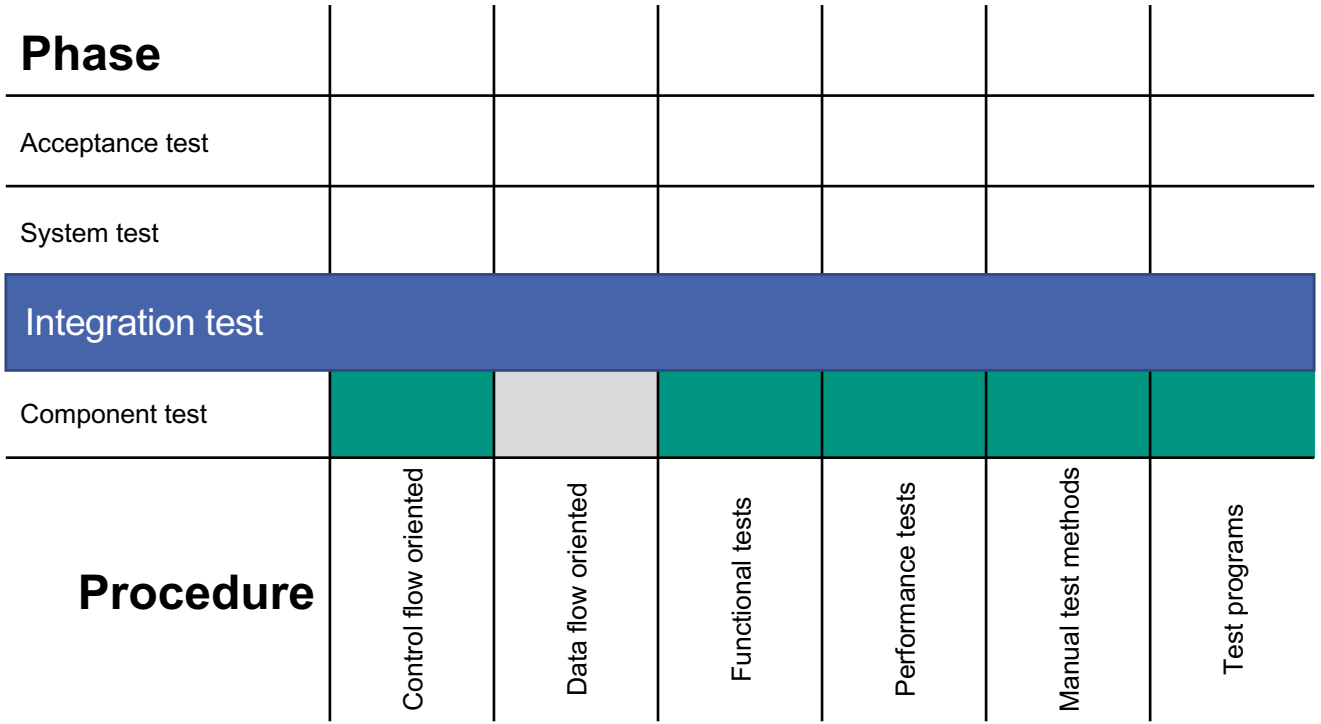
Static Analysis Tools

- Static analysis of a software component takes place during the translation of the source code or by special tools
- For static analysis of the source code there are special applications and plug-ins for many development environments.

Static Analysis Tools

- Warnings and errors
  - Are displayed by the development environment. These include, for example:
    - Possibly not initialized variables
    - Unreachable instructions
    - Unnecessary instructions
- Check programming style
  - Indentation inconsistent or forgotten or too deep/flat
  - JavaDoc comments forgotten
  - Parameters not declared as final
- Find errors using error patterns
  - With the help of a static analysis, errors can be found based on certain patterns
- Inserts for development environments and tools that provide such functionalities are discussed in chapter 5.2.

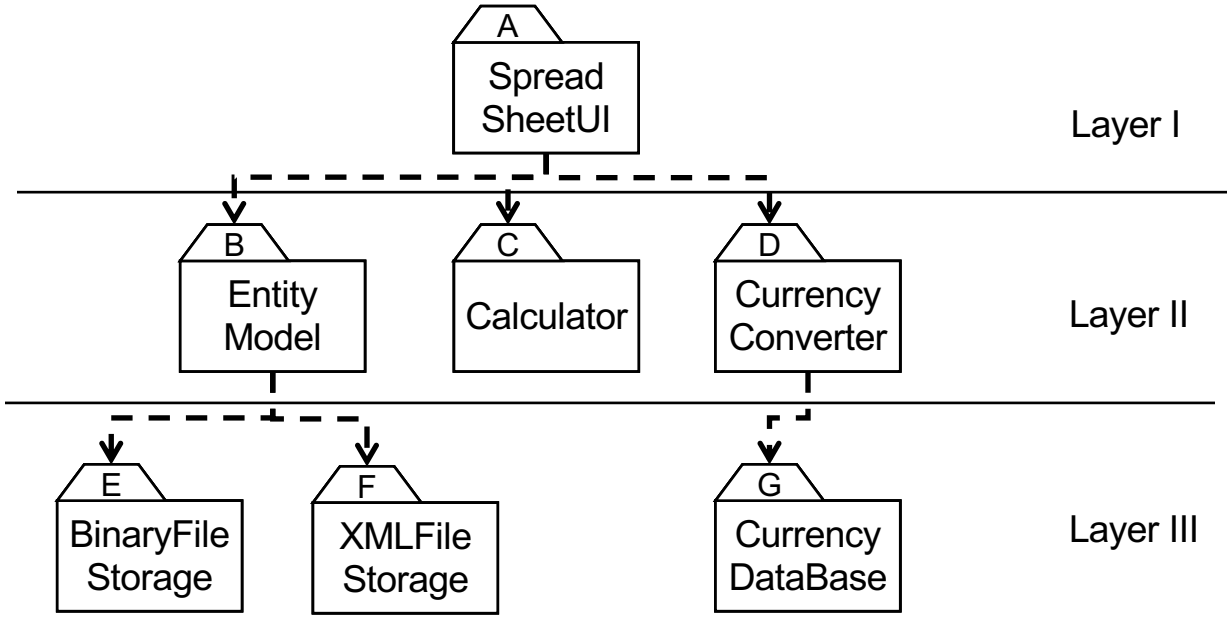
Overview matrix: What comes next?



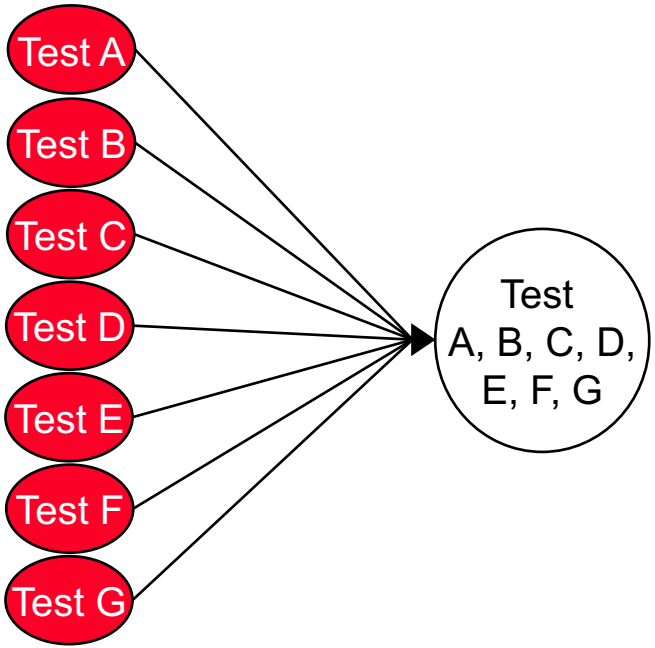
Integration Testing

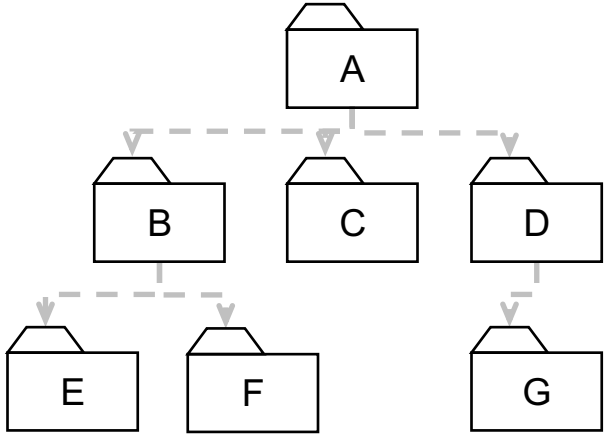
- The entire system is viewed as a collection of subsystems (sets of classes) determined during design
- Goal: Test all interfaces between subsystems and the interaction of subsystems
- The integration testing strategy determines the order in which the subsystems are selected for testing and integration
  - Big Bang integration
  - Bottom-Up testing
  - Top-Down Testing
  - Vertical Integration.

Example: Integration Testing for a 3-Layer-Design



Big-Bang Approach





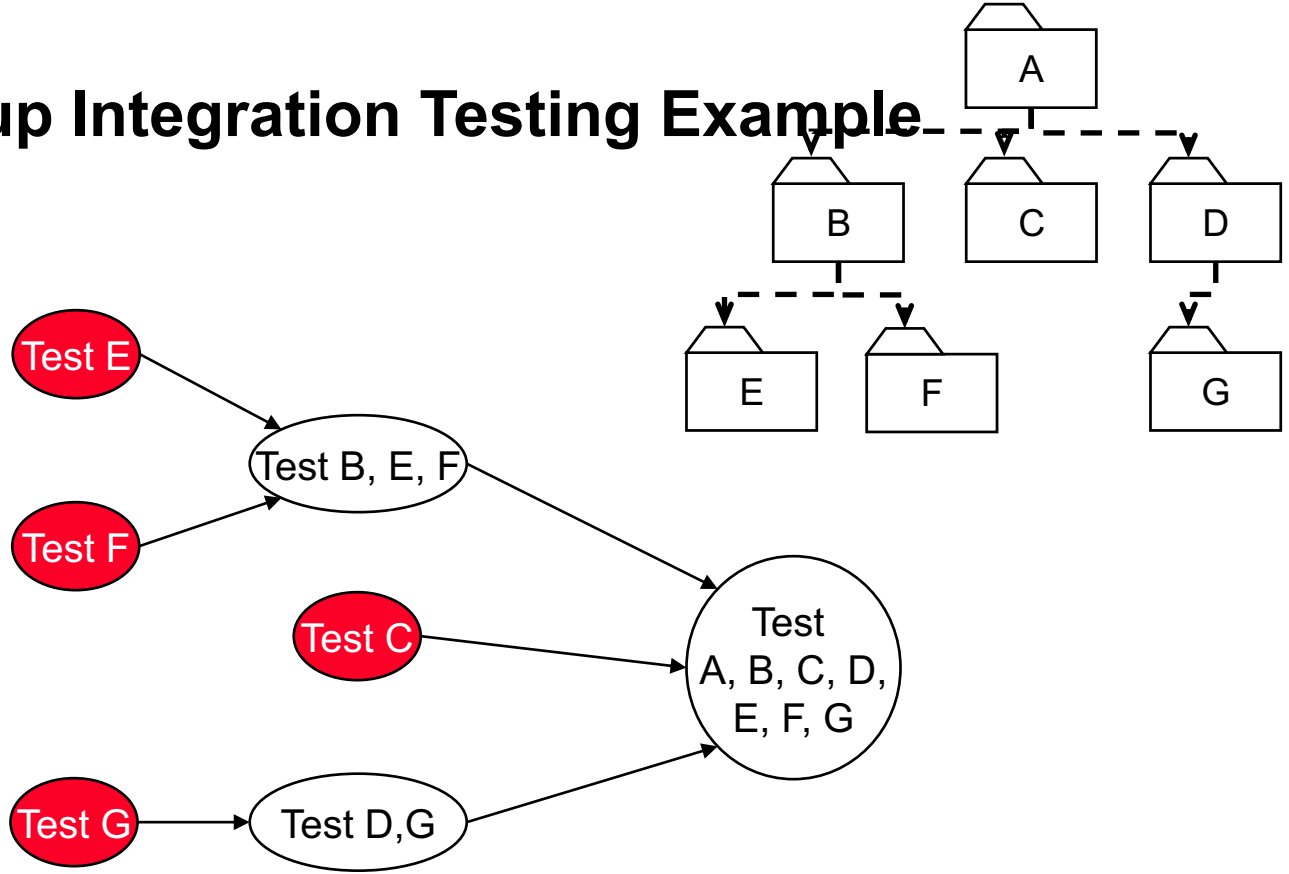
Big Bang: Nothing works, until everything works.

- Construction of test cases difficult
- Defect search difficult
- Interfaces not tested separately
- the worst possible strategy
- (head-in-the-sand)

Bottom-up Testing Strategy

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

Bottom-up Integration Testing Example



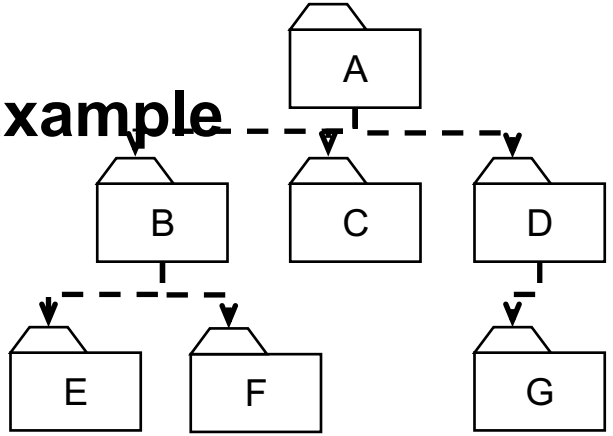
Pros and Cons: Bottom-Up Integration Testing

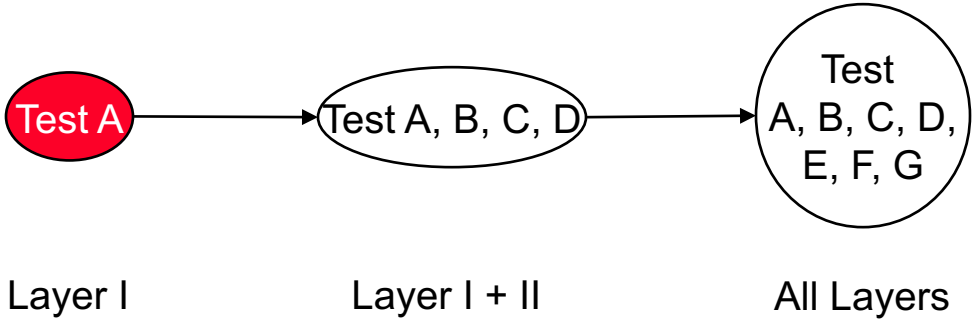
- Pros
  - No stubs and dummies needed
  - Useful for integration testing of the following systems
    - Object-oriented systems
    - Systems with strict performance requirements, e.g. real-time systems
- Cons:
  - Tests an important subsystem (the user interface) last
  - Test drivers are needed (but available through JUnit)

Top-down Testing Strategy

- Test the subsystems in the top layer 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 tests.

Top-down Integration Testing Example





Pros and Cons: Top-Down Integration Testing

Pros:

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

Cons:

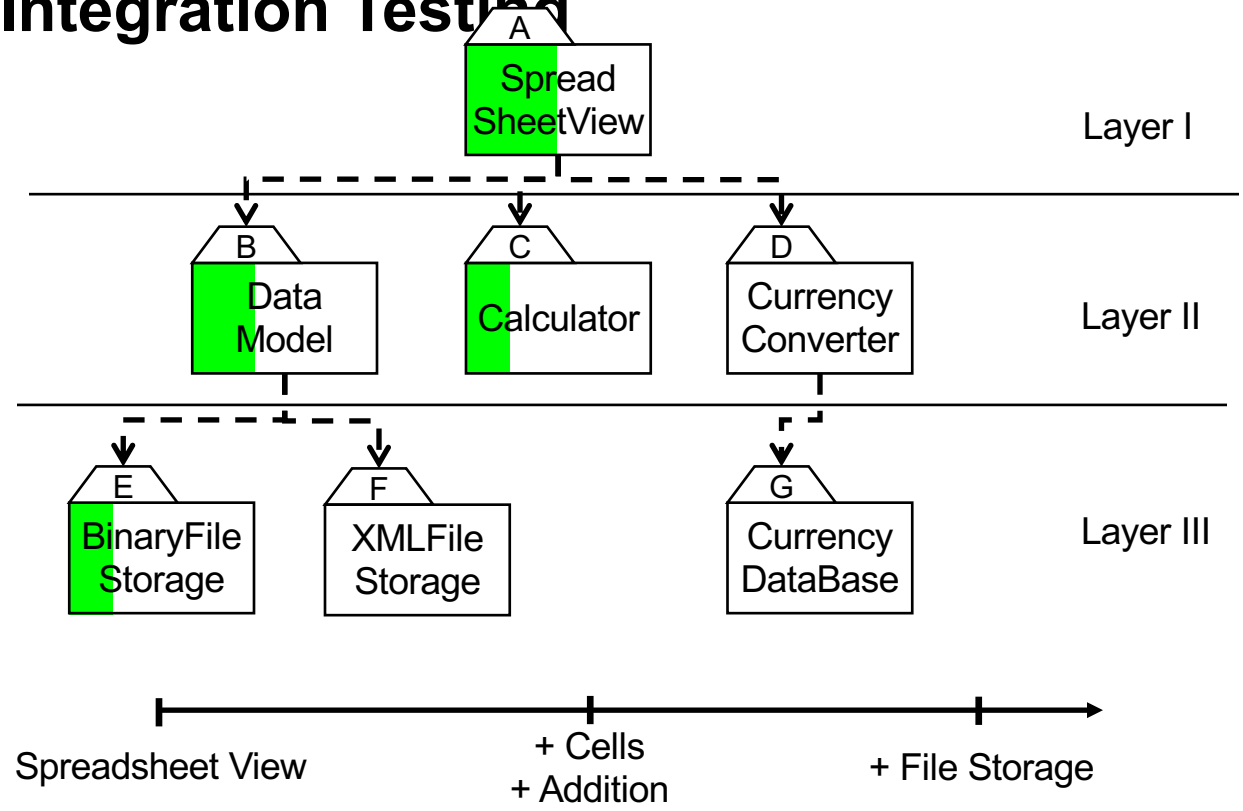
- Stubs and dummies (helpers) are needed
- Writing helpers is difficult: helpers must allow all possible conditions to be tested
- Large number of helpers may be required, especially if the lowest level of the system contains many methods
- Some interfaces are not tested separately

- One solution to avoid too many helpers: *Modified top-down testing strategy*
  - Test each layer of the system decomposition individually before merging the layers
  - Disadvantage of modified top-down testing: Both, helpers and test drivers are needed

Horizontal Integration Testing Risks

- Risk #1: The higher the complexity of the software system, the more difficult is the integration of its components
- Risk #2: The later integration occurs in a project, the bigger is the risk that unexpected failures occur
- Horizontal integration strategies (Bottom-up, top-down, sandwich testing) don't do well with risk #2
- Vertical integration addresses these risks by building as early and frequently as possible
  - Used in Scenario-driven design: Scenarios are used to drive the integration
  - Used in Scrum: User stories are used to drive the integration. Potential deliverable product increment
- Advantages of vertical integration:
  - There is always an executable version of the system
  - All the team members have a good overview of the project status.

Vertical Integration Testing



Horizontal vs Vertical Integration Testing

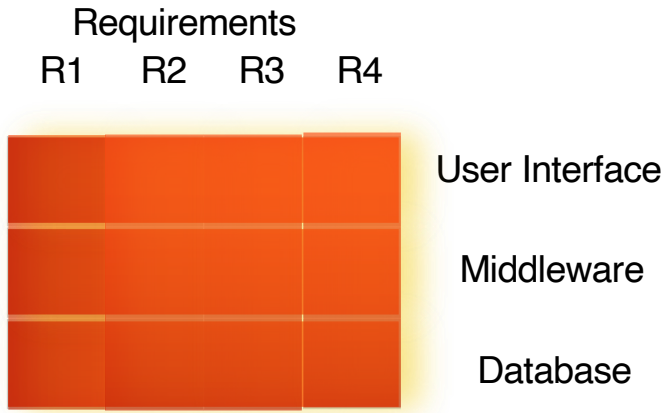
Test Cases

Bottom-up integration tests

Top-down integration tests

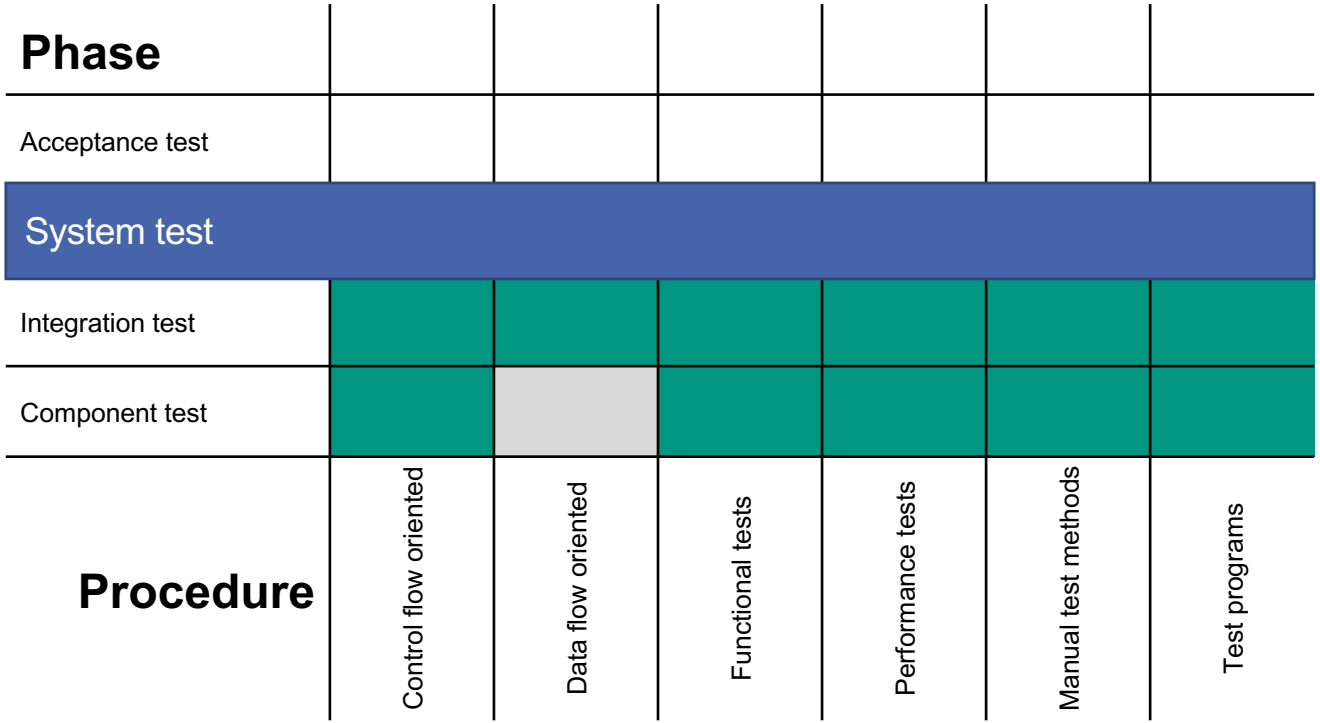
Sandwich integration tests[1]

Vertical integration tests



[1]Not covered in today's class. Find the definition in the textbook

Overview matrix: What comes next?



System test

- Testing of the complete system against the product specification
- System as "*black box*" - only the external system view (user interface, external interfaces)
- Real (realistic) environment, e.g., embedded system, application, control panel, etc.

Classification of system tests

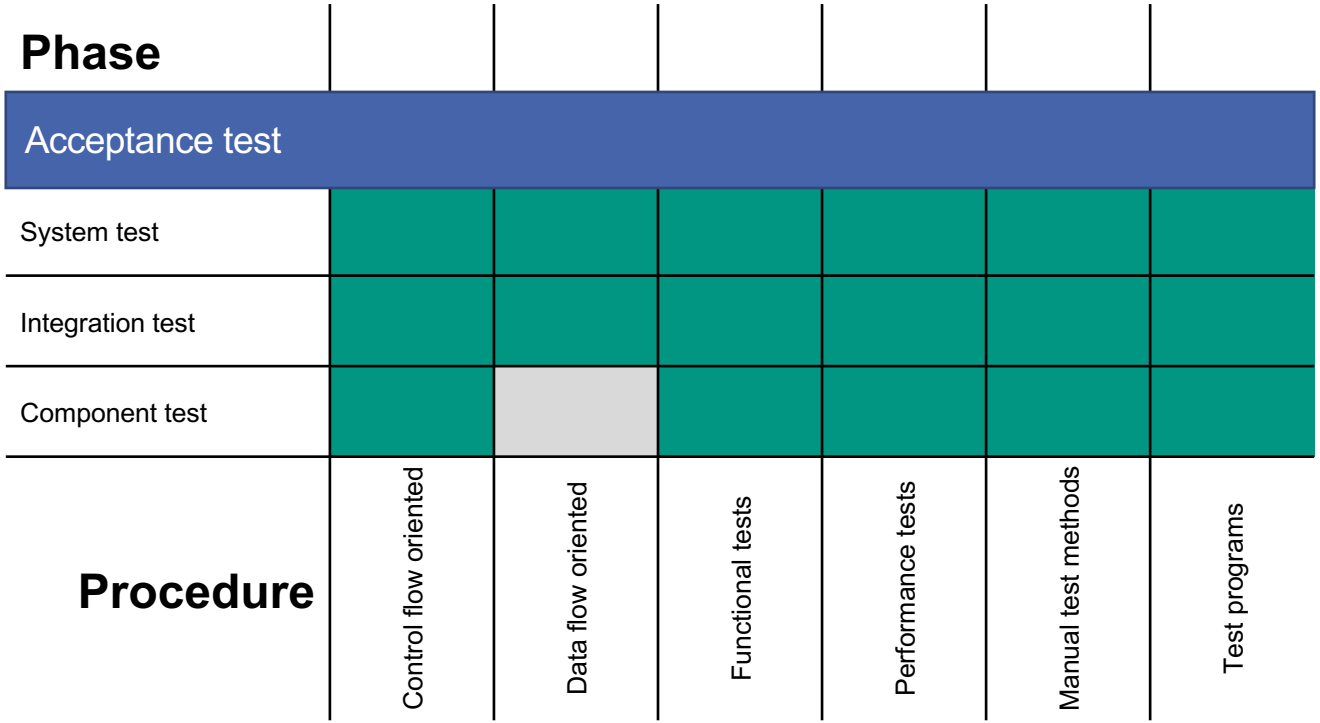
- System testing is divided into functional and non-functional system testing.
  - **Functional system test:** Verification of the functional quality characteristics correctness and completeness.
  - **Non-functional system test:** Verification of non-functional quality characteristics, such as.
    - Security
    - Usability
    - Interoperability
    - Documentation
    - Fail-safety

...and of course Performance tests!

Regression Test

- A regression test is the repetition of a system test because of maintenance, modification, or correction of the system under consideration.
- Purpose: to ensure that the system has not reverted (“regressed“) to a worse state than before.
- The results of the regression test are compared with the results of the previous test.

Overview matrix: What comes next?



Acceptance Tests

- Special system test in which
  - the customer observes, participates and/or directs the test,
  - the real operating environment is used at the customer site
  - and if possible real data of the client are used
- The client may adapt or modify system test cases and/or perform its own test scenarios, but ...
- ... formal acceptance (for which the acceptance test is the basis) is the binding declaration of acceptance (in the legal sense) of a product by the client.

Literature

- [Balzert98] H. Balzert. *Textbook of software engineering*, 2nd vol. Spectrum, 1998.
- [BrDu04] B. Bruegge, A.H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns and Java*, Pearson Prentice Hall, 2004, p. 435ff.