Software Construction and User Interfaces (SE/ComS 319)

Ali Jannesari

Department of Computer Science

Iowa State University, Fall 2018

TESTING

Administrative stuff

Final Exam:

- **Tuesday Dec 11, 2018, 12:00 PM, MARSTON 2155 (lecture room)**
- https://www.registrar.iastate.edu/students/exams/fallexams

Outline

- Testing (review)
- Statement Coverage
- Branch Coverage
- Path coverage
- Functional testing (self-study, not relevant for the exam)
- Performance testing (self-study, not relevant for the exam)
- Manual testing methods (self-study, not relevant for the exam)

Testing – Motivation

- Software artifacts always contain errors
- The later a mistake is found, the more expensive it is to fix it
- The goal is to find mistakes as early as possible

Error detection is the goal of testing (1)

> *„Testing shows the presence of bugs, not their absence.“*
>
> (Edsger W. Dijkstra)

- Why do we need testing?
  - Catch bugs (defect testing)
  - Check if we follow all the requirements (Validation testing)
  - **Providing “documentation” in XP process**
- How much resources are spent in testing?

Error detection is the goal of testing (2)

- Complete testing of all **combinations** of all **input values** is **not possible** except for trivial programs (astronomical number of test cases).
- Correctness is possible only with **formal** correctness proof (correspondence of specification and program); this is only possible for small programs today.
- Central question: When can one stop testing to look for errors? (**Test completeness criteria**)
  - When is sufficient?
  - Exhaustive testing

Error detection is the goal of testing (3)

- Attention:

Differences:

- Testing procedure → Detect error
- Verifying procedure → Prove correctness
- Analyzing procedure → Determine properties of a system component

There are 3 types of errors ...

- A **failure** (*fault)* is the deviation of the behavior of the software from the specification (an event).
- A **defect** (*bug)* is a deficiency in a software product, which can lead to failure (a condition).
- It is said that a defect manifests itself in failure.
- A **mistake** is a human action that causes a defect (a process).

Error

Test doubles (Types of test auxiliary/assistant)

- A ***stub*** is a rudimentary part of the implemented software and serves as a placeholder for yet unreacted functionality.
- A ***dummy*** simulates the implementation for test purposes
- A ***mock*** *object* is a dummy with additional functionality, such as setting the reaction of the imitation to certain inputs or checking the behavior of the "client"
- More details later (if we get time…).

Error classes (1)

- **Requirement errors** (defect in the requirements)
  - Incorrect information of user requests
  - Incomplete information about functional requirements, performance requirements, etc.
  - Inconsistency of different requirements
  - impracticability (not feasible)
- **Design errors** (defect in the specification)
  - Incomplete or incorrect implementation of the requirement
  - Inconsistency of specification or design
  - Inconsistency between requirement, specification and design

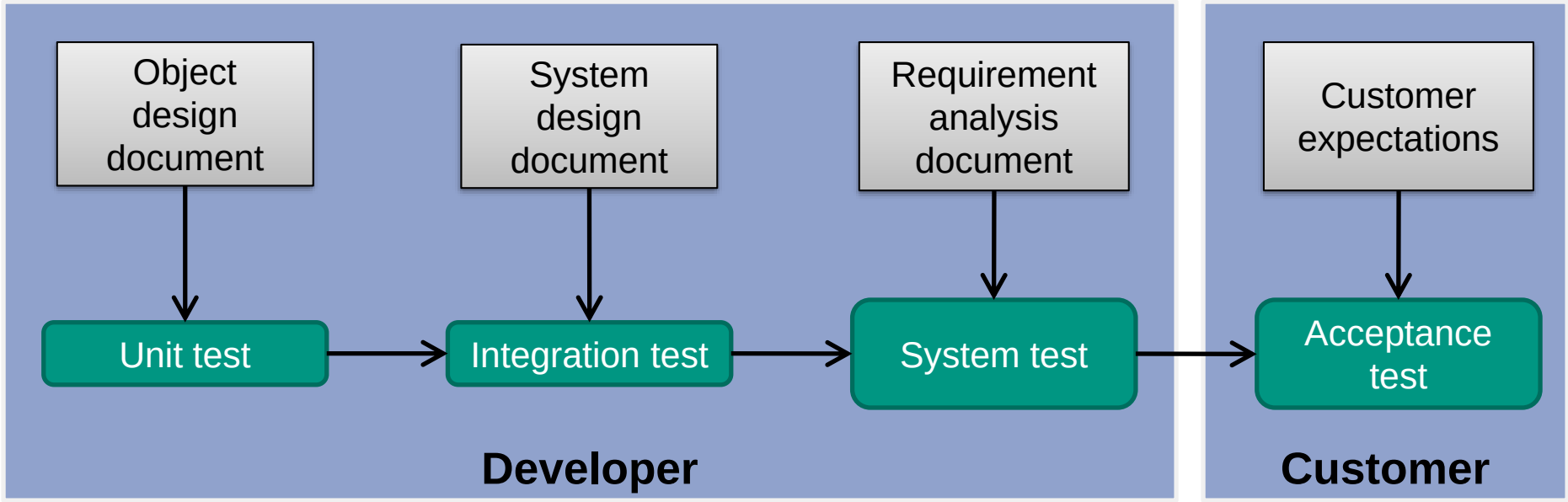
Error classes (2)

- **Implementation error** (defect in the program)
  - Incorrect implementation of the specification in a program

Module / Software testing process

- A **software test**, test for short, executes a single software component or a configuration of software components under known conditions (inputs and execution environment) and verifies their behavior (outputs and responses).
- The to be tested SW-component or configuration is called **test object** (component under test, CUT).
- A **test case** consists of a set of data for the execution of a part or the whole test object.
- A **test driver** (**test framework)** supplies test objects with test cases and initiates the execution of the test objects (interactively or automatically).

What are the different stages of testing?



The **unit test** checks the function of a single module/function by observing the processing of test data.

The **integration test** progressively checks the error-free interaction of system components that have already been individually tested.

The **system test** is the contractor's final test in a real (or realistic) environment without customers.

The **acceptance test** is the final test in real environment under observation, participation and / or leadership of the customer.

Classification of testing methods (1)

- Dynamic methods — **Testing**
  - Structural tests (white / glass box testing)
    - Control flow-oriented tests
    - Data flow-oriented tests
  - Functional tests (black box testing)
  - Performance tests (also black box)
- Static methods — **checking**
  - Manual test methods (inspection, review, walkthrough)
  - Test programs (static analysis of programs)

Classification of testing methods (2)

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

Classification of testing methods (2)

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

Classification of testing methods (2)

- Dynamic methods
  - The compiled, executable program is tagged and executed with certain test cases
  - The program is tested in the real (realistic) environment
  - Sampling method: Correctness of the program is **not proven!**
- Static methods

White Box Testing:
Determining the values **with knowledge** of control and / or data flow

Black Box Testing:
Determining the values **without knowledge** of control and / or data flow; just out of specification

Overview Matrix

Phase						
Acceptance test						
System test						
Integration test						
Unit test						
Method	Control flow oriented	Data flow oriented	Functional Tests	Performance tests	Manual testing methods	Testing programs

Control flow oriented (CFO) test methods

- **Statement Coverage**
- **Branch Coverage**
- **Path coverage**
  - Completed
  - Structured
- Condition coverage
  - Simple
  - Multiple
  - Minimal multiple
- Boundary-Interior Test

How to judge the quality of a test suite? – Test coverage criteria

- Completeness criteria for CFO test method are defined by "control flow graphs"
  - Definition of an intermediate language (IL)
  - Definition of the transformation into the intermediate language
  - Definition of the control flow graph
- Then: definition of the testing method and their corresponding coverage

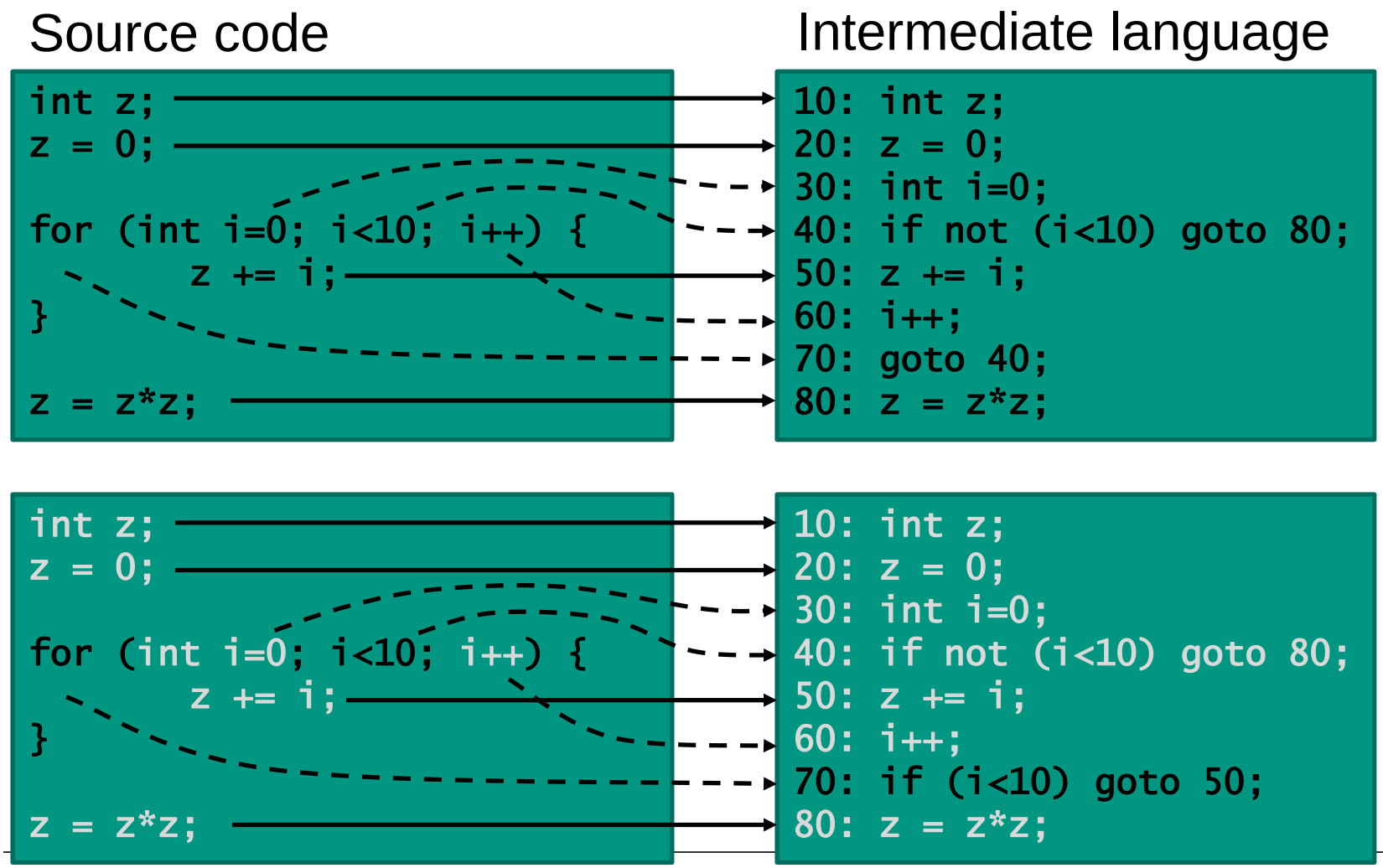
Definition: Intermediate Language (IL)

- We define an **intermediate language** consisting of:
  - arbitrary commands except those that affect the execution order (such as conditional statements jumps, loops, etc.),
  - conditional/ unconditional jump instructions (**goto**) to arbitrary but fixed positions of the instruction sequence,
  - any number of variables.
- The intermediate language is based on what is commonly understood by "assembler code".
- The implemenation (especially the glossary of the commands) of this intermediate language is irrelevant here.

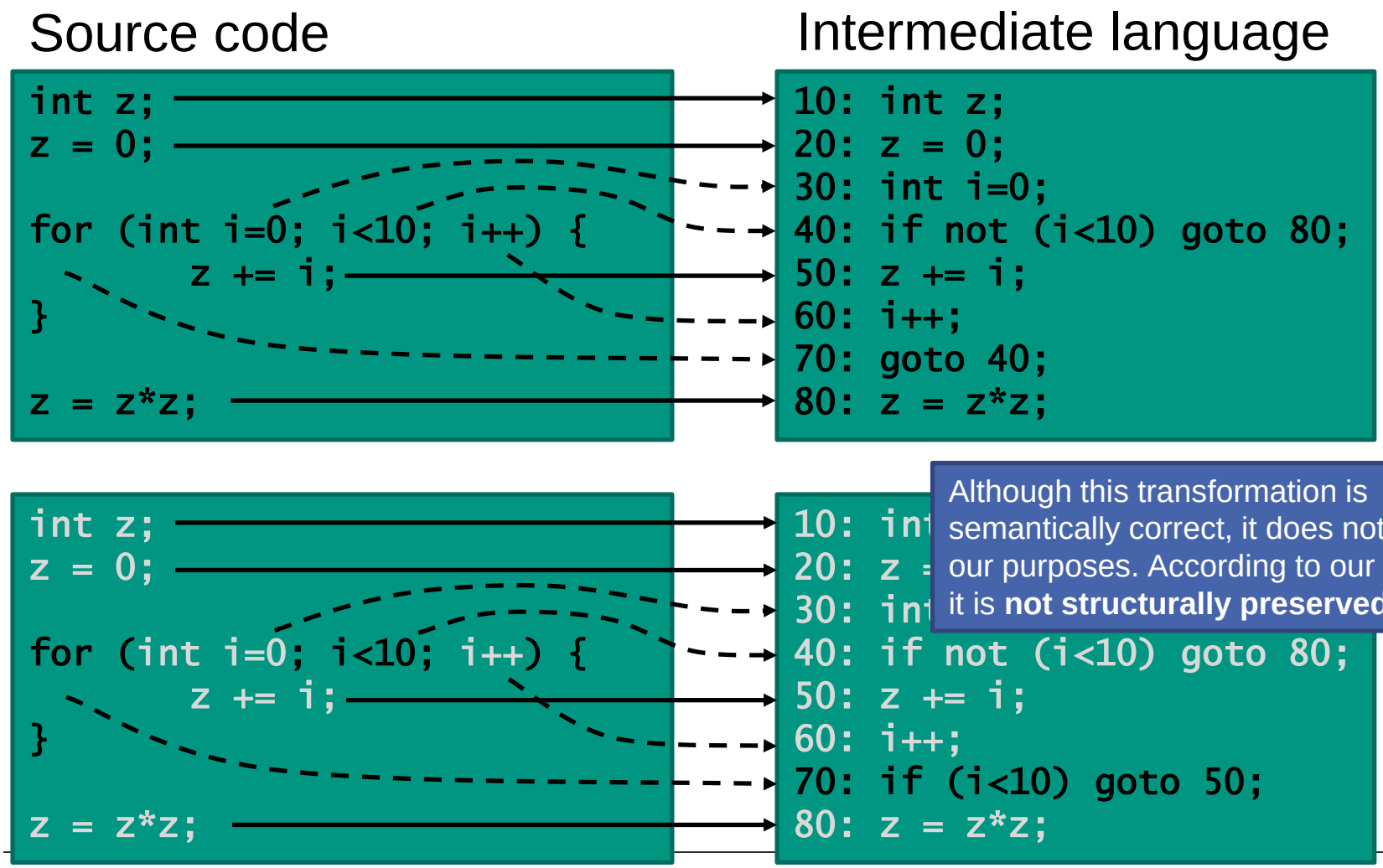
Definition: Structure-preserving transformation

- We speak of a **structure-preserving transformation** of a source language (e.g. Java) into the intermediate language, if
  - (exclusively) the commands affecting the execution order are replaced by intermediate language command sequences, where
    - the order of execution of the other commands remains the same with the same parameterization with that in the source language!
- All other commands are taken over unchanged
- Transforms are avoided where statement sequences or conditional jumps are replicated (no loop unrolling, no optimizations.)

Transformation – Example



Transformation – Example



Definition: Basic block (BB)

- A **basic block** denotes a maximum length of consecutive statements of the intermediate language,
  - in which the control flow occurs only at the beginning and
  - which contains no jump instructions except at the end.
  - In other words, has only one entry point and one exit point!

Example:

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

1 basic block

next basic block

Definition: Control flow graph

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

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

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_{stop} the stop block.

Find control flow graph – Example

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

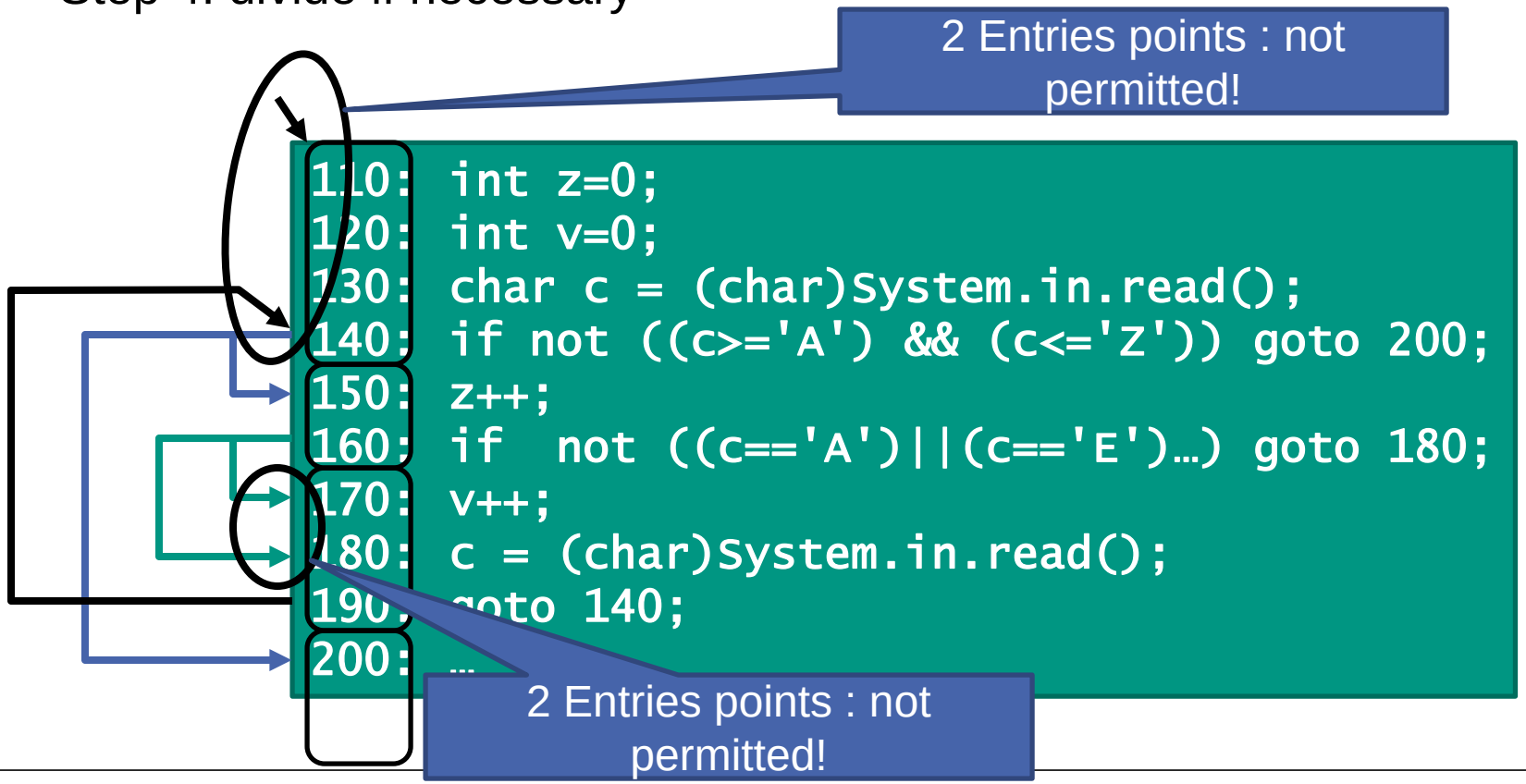
Find control flow graph – Example

- Step 1: Transform in intermediate language
- Step 2: Summarize all sequences ending in a jump into a basic block

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

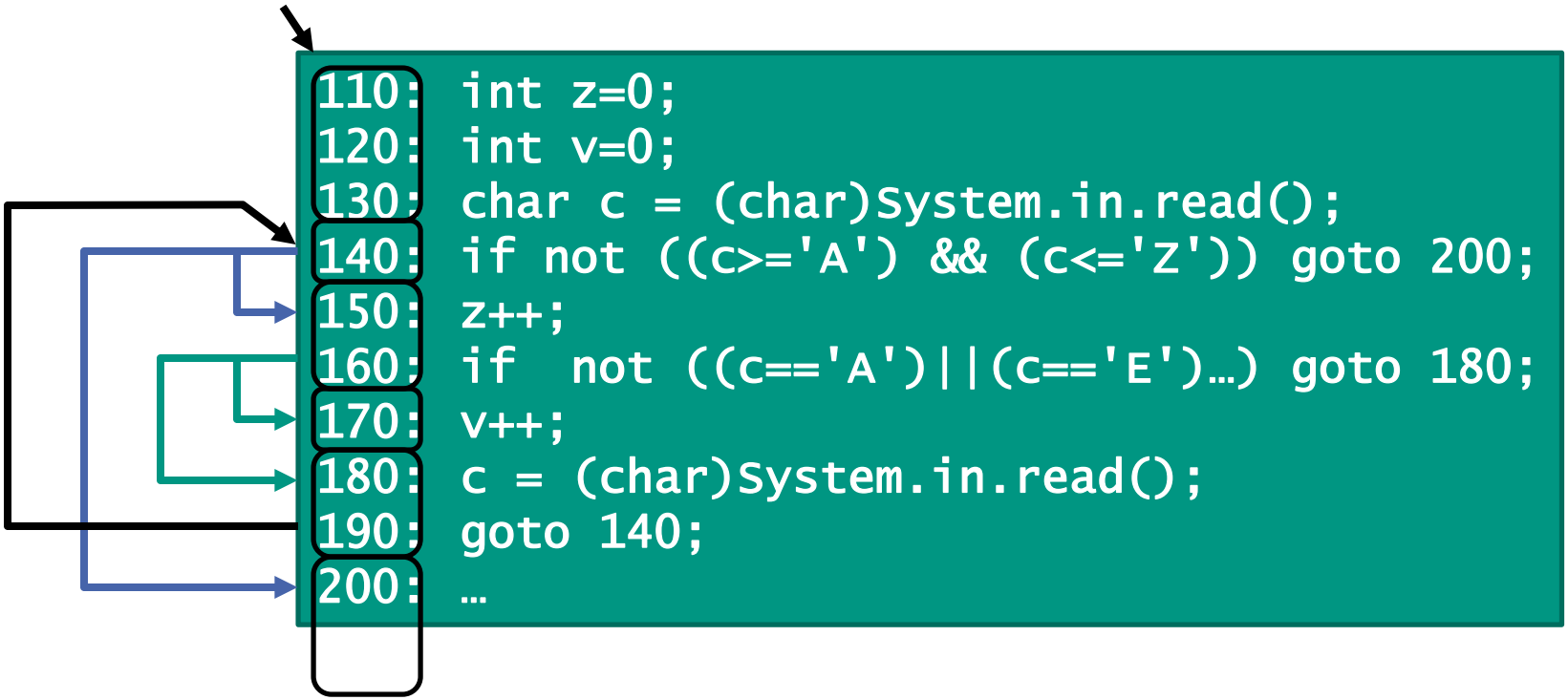
Find control flow graph – Example

- Step 3: Check if entry is only at the beginning
- Step 4: divide if necessary



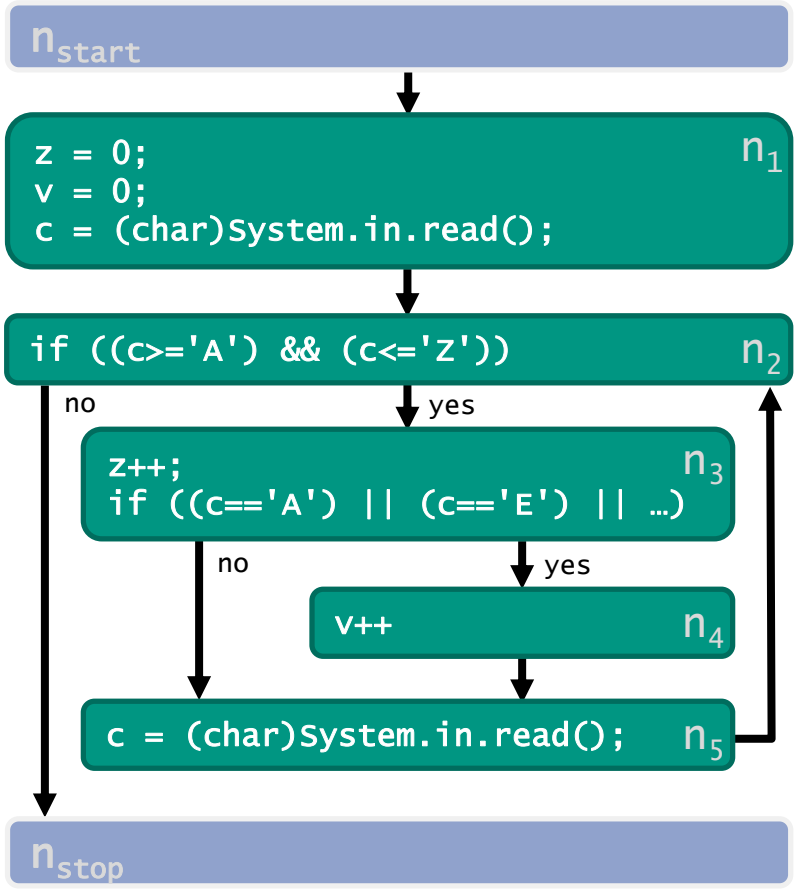
Find control flow graph – Example

- Step 3: Check if entry is only at the beginning
- Step 4: divide if necessary



Find control flow graph – Example

- Control flow graph (CFG):

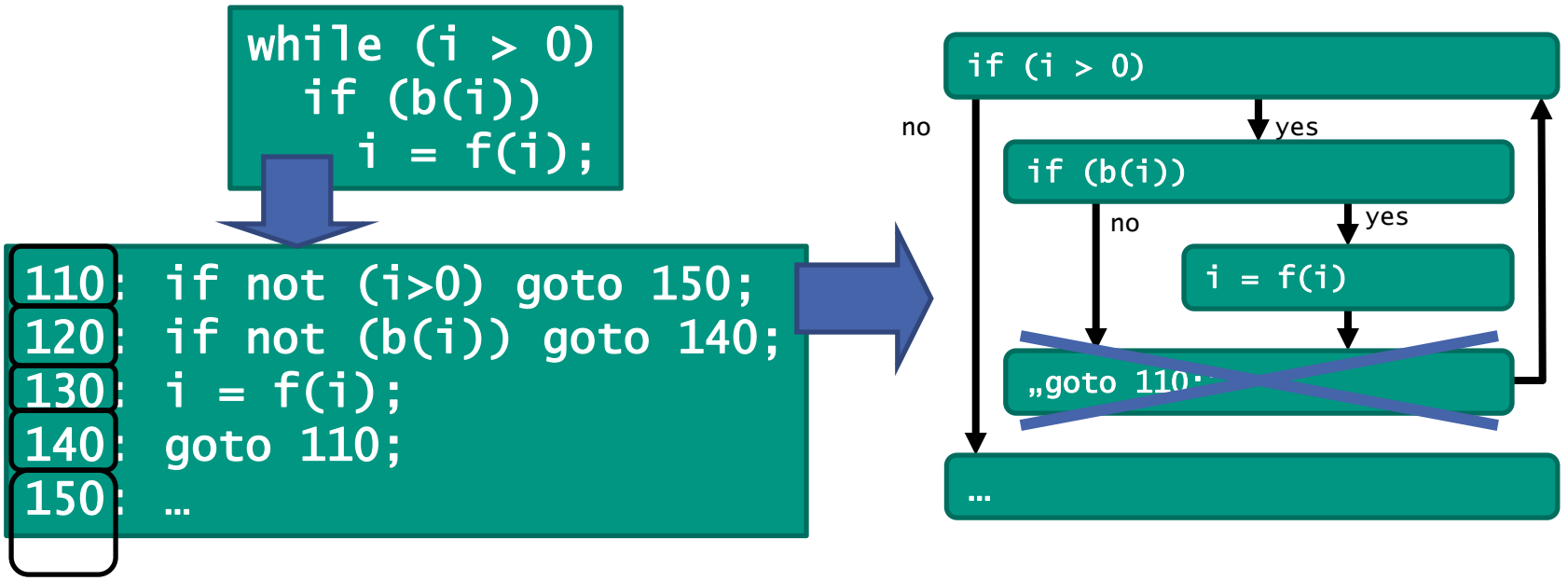


Definitions: branch, full paths

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

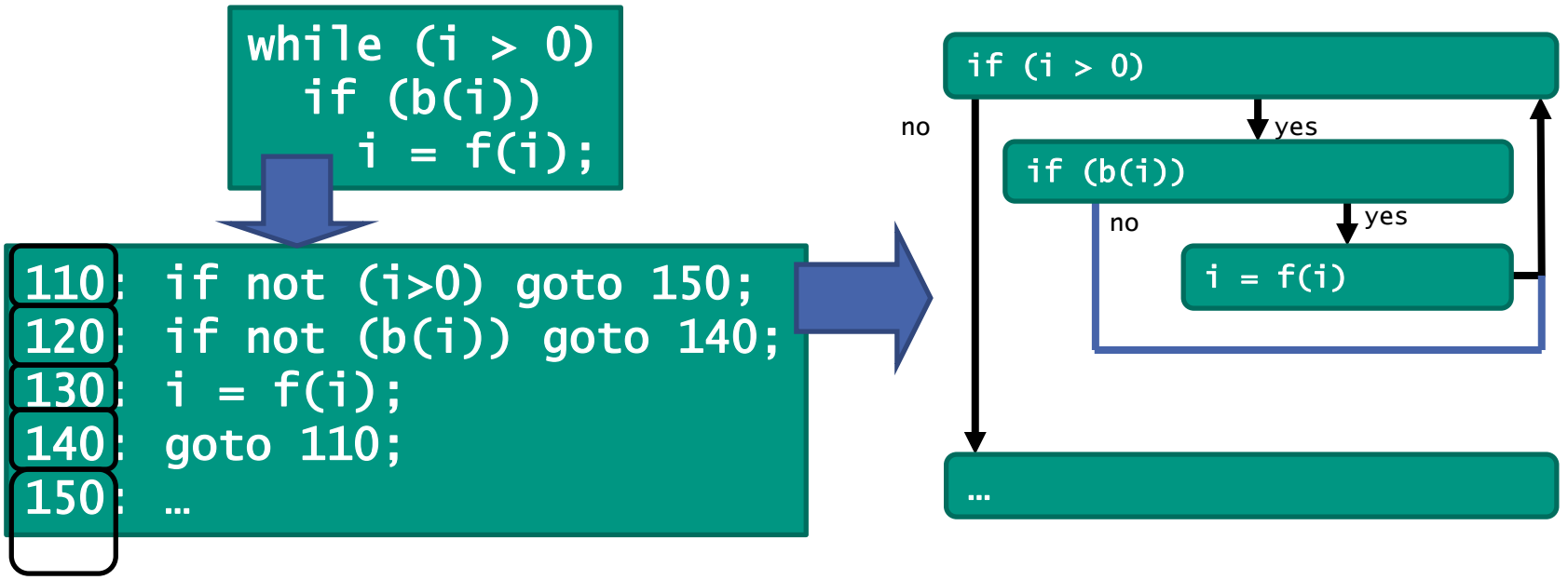
Simplify control flow graph

- Perhaps:
- If a basic block, which only contains an **unconditional jump** command, is created by splitting it can be removed (reduction of the BB).



Simplify control flow graph

- Perhaps:
- If a basic block, which only contains an unconditional jump command, is created by splitting it can be removed (reduction of the BB).



Definition: Statement coverage

- The test strategy **statement coverage** $C_{Statement}$ requires the execution of **all basic blocks** of the program P.
  - C stands for coverage
  - Metric, also called C_0

$$C_{Statement} = \frac{\text{Number of instructions passed}}{\text{Number of all instructions}}$$

  - Insufficient test criterion
  - Non-executable program parts can be found.
  - Missing program parts are not detected.
- Also called statement capture (a test completion criterion)

Definition: Branch coverage

- **Branch coverage** C_{branch} requires the traversal of **all branches** in the CFG.
- Metric, also called C_1

$$C_{branch} = \frac{\text{Number of traversed branches}}{\text{Number of all branches}}$$

  - Branches that are not executed can be detected
  - Neither combination of branches (paths) nor complex conditions considered
  - Loops are not tested sufficiently
  - Missing branches not testable, not detected.
- Also called branch capture.

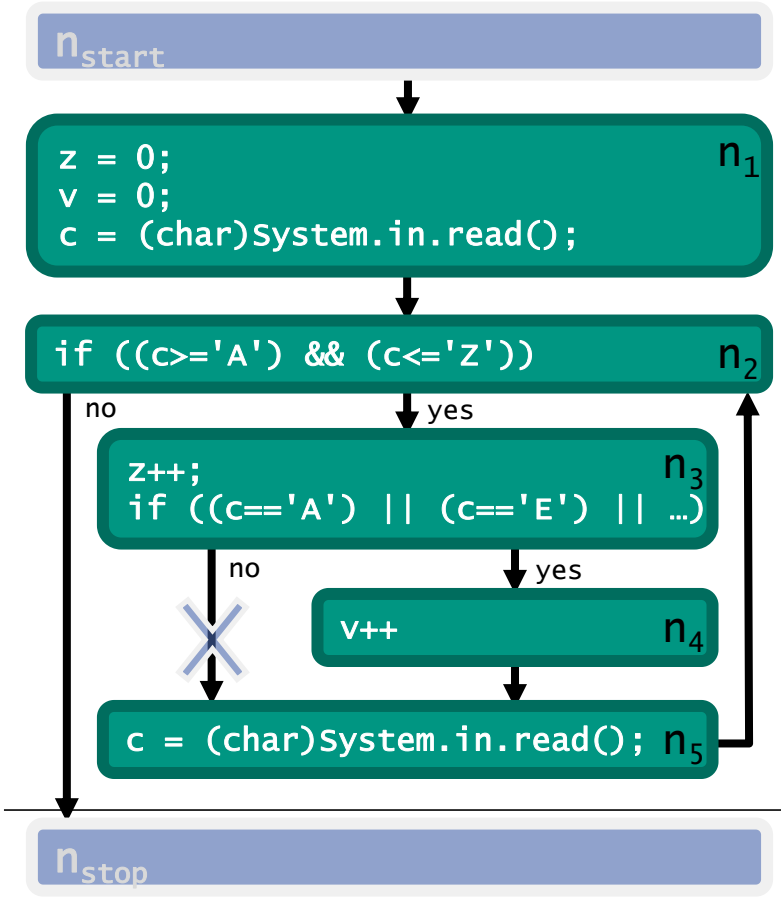
Statement coverage vs. Branch coverage

Statement coverage, all nodes

Example sequence:

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

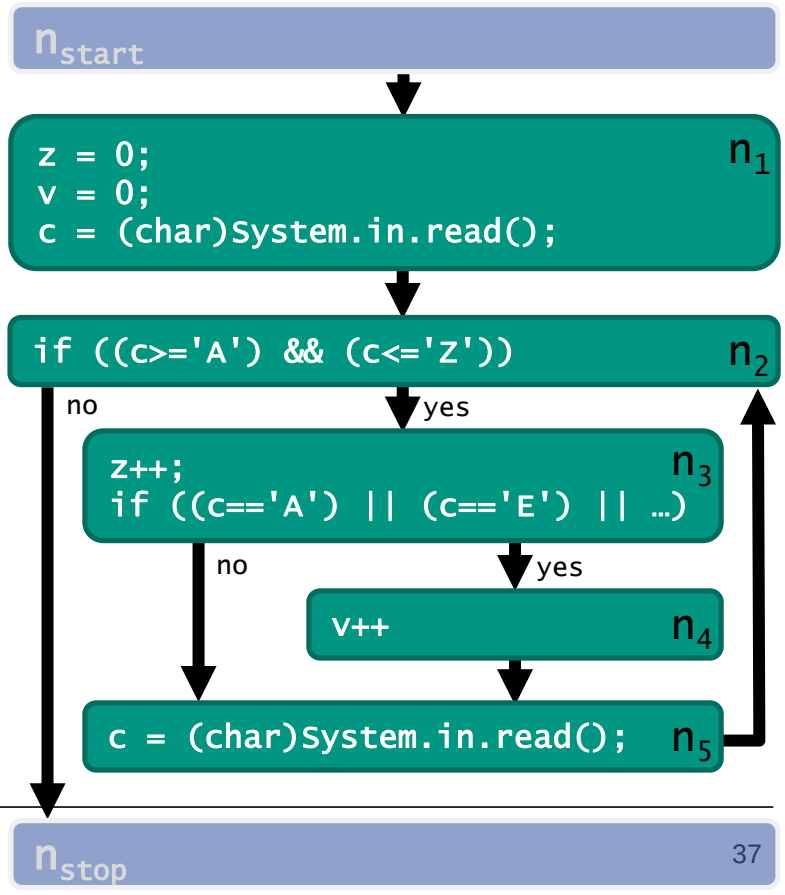
→ branch (n_3, n_5) is not executed



Branch coverage, all edges

Example sequence:

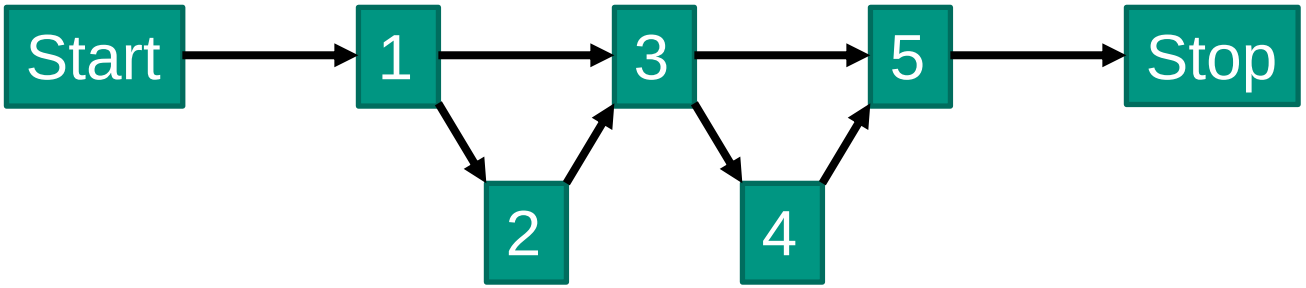
(n_{start}, n_1, n_2, n_3, n_4, n_5, n_2, n_3, n_5, n_2, n_{stopp})



Definition: Path coverage

- The **path coverage** calls for the execution of all different, full paths in the program.
  - Path number grows dramatically in loops.
  - Some paths may not be executable, due to mutually exclusive conditions
  - Most powerful CFG test strategy
  - Not practicable/feasible

Example of statement, branch, and path coverage



- Statement Coverage

  A = {(Start, 1, 2, 3, 4, 5, Stop)}

- Branch coverage

  Z = A ∪ {(Start, 1, 3, 5, Stop)}

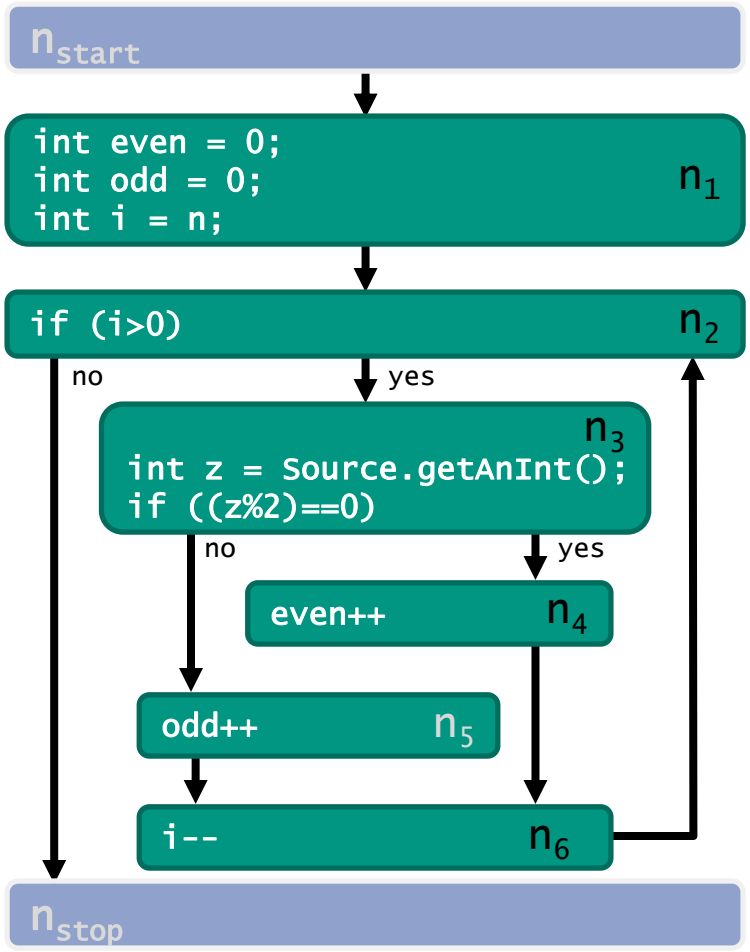
- Path coverage

  P = Z ∪ {(Start, 1, 3, 4, 5, Stop)} ∪ {(Start, 1, 2, 3, 5, Stop)}

Overhead of the path coverage – Example

```
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 Path
0	I
1	II
2	IIII
…	…
k	2^k



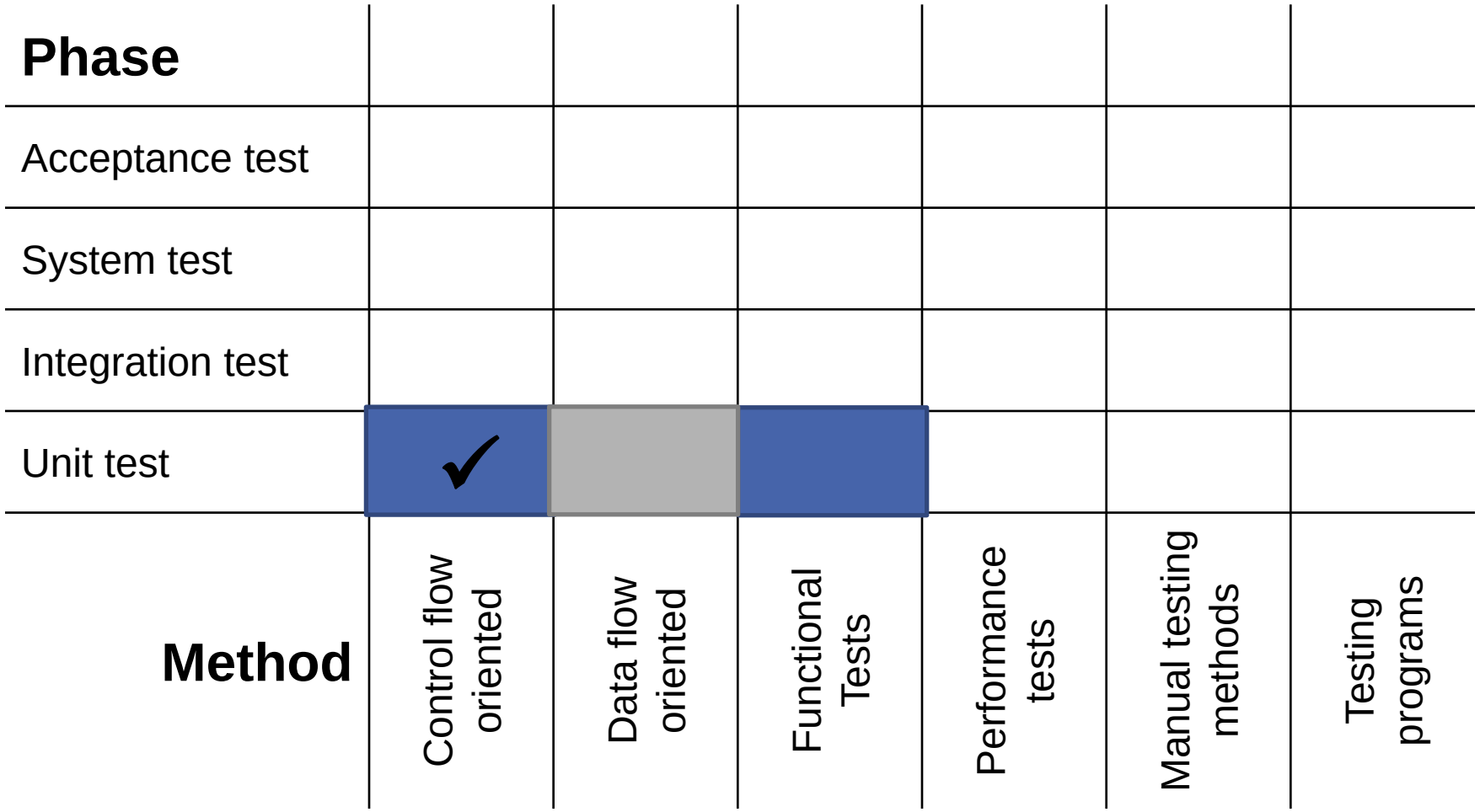
Summary: CFO Test Strategies

- **Instruction coverage test** is the weakest criterion. Each statement must be executed at least once in order to have a chance to find defects in it. (if you do not execute anything at all, you will not discover any defects there)
- **Branch coverage** subsumes statement coverage . Requires that branches be executed at least once to have a chance to detect defects in all branches.
- **Path coverage** is the most elaborate criterion, and even for small programs with loops is not feasible.
- **In general:** the different test strategies are also **test completeness criteria**. (Example: a test of the branch coverage is complete if $C_1 = 1$)

Following parts on Testing are self-study (not relevant for the exam)

TESTING (SELF-STUDY)

Overview Matrix



Functional Tests

- **Goal**: Testing the specified functionality
  - Derive test cases from the specification
  - Internal structure of the test object not considered because unknown to the tester
- **Advantages**:
  - Test cases can be created independently of the implementation (before or at the same time)
  - Avoids " Nearsightedness" in the selection
- **Disadvantage**:
  - Possible **critical paths** not known & not tested

formal – automated
informal – manual

Test cases include
- Input data
- Expected output data / reaction

Methods for creating functional test cases

- **Functional equivalence class formation**
- **Boundary value analysis**
- **Random test**
- Test of state machines

Functional equivalence class formation

- **Assumption**: A program / module responds to the processing of a value from a certain range as well as processing any other value from that range.
- **Example** : We expect that a function that correctly calculates 422 will also calculate 402 correctly.
- **Approach** : Decompose the value range of the input parameters and the definition range of the output parameters in equivalence classes (EC).

Functional equivalence class formation

- **Assumption**: A program / module responds to the processing of a value from a certain range as well as processing any other value from that range.
- **Example** : We expect that a function that correctly calculates 422 will also calculate 402 correctly.
- **Approach** : Decompose the value range of the input parameters and the definition range of the output parameters in equivalence classes (EC).

Does not have to be in the definition range of the input parameters!
→ Testing error handling

Functional equivalence class formation

- Formation of equivalence classes (approach idea)
  - **Partitioning** from a large EC
  - Division based on the definition of area boundaries
  - Distribution based on **assumed** processing method limits
- Selection of representatives
  - Let *m* be the number of EC of the input parameters and *n* the number of EC of the output parameters.
  - Then $m \cdot n$ different EC are generated from which each 1 **arbitrary** representative is used for testing

Equivalence Classification: Example (1)

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

EC	Description	Expected 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 input	Month < 1 or Month > 12
$EC_{Y,LY}$	Leap year	1904, 2000, 2012, ...
$EC_{Y,Not-LY}$	Not-leap year	1901, 1900, ...
$EC_{Y,Error}$	Invalid input	Year < 0

Equivalence Classification: Example (2)

- Derivation of test cases (selection of valid entries):

Equivalence Classes	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 Classification: Example (3)

- Derivation of test cases (selection of **invalid** entries):

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

Boundary value analysis

- Further development of functional equivalence class formation
- **Observations**:
  - Off-by-one: Close by is also over
  - Test cases that cover the limits of equivalence classes and their immediate environment are particularly effective

  ⇒ Do not use any element from the EC, but 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 (infinities), ...

Random testing

- Testing the function with random test cases
- **Observation**: Testers tend to generate test cases that were considered obvious in the implementation (a kind of operational blindness)
- **Advantage**: Nondeterministic method treats all test cases equally
- Useful as a supplement to other methods
- Useful if you want to generate many test cases (e.g. for sorting) and the verification of correctness can be done automatically.
- Useful as a subordinate criterion (e.g. in the process of functional EC formation)

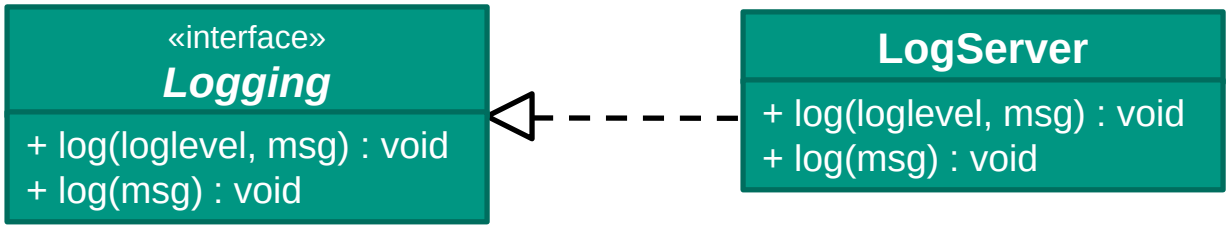
Use of test doubles (test assistants) (1)

- Objects do not live in isolation, objects work together in one application.
- Collaboration results in dependencies on testing.
- How can individual properties be tested for these dependencies?
  - What exactly should be tested?
  - Which problems should be tested?
- How can tests be written that resolve dependencies?

Use of test doubles (test assistants) (2)

- If classes depend on each other, unimplemented classes can be replaced by **test doubles** (stub, dummy, or mock).
- Stubs or dummies **represent** the missing implementations.
- Stubs and dummies are successively replaced by real implementations; mocks continue to be useful for future testing of the program.

Example: Dummy (1)

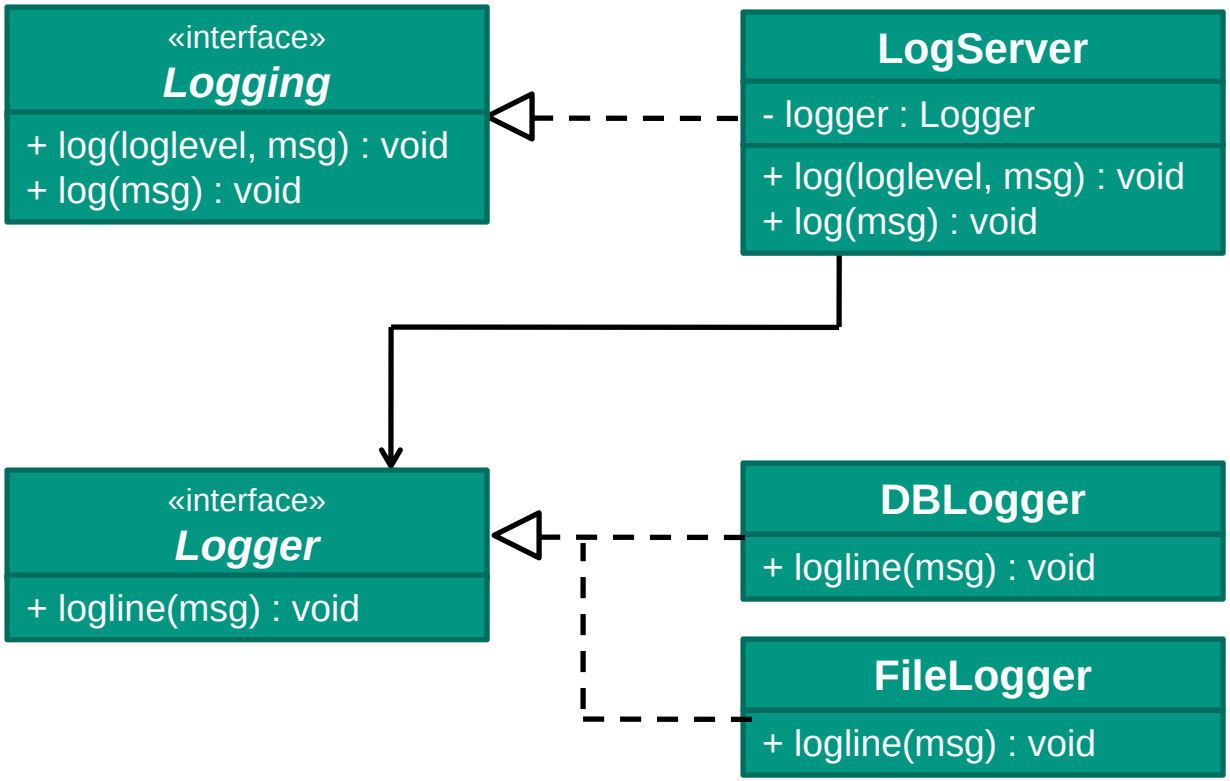


- Provide system-wide logging management
- `log (loglevel, msg)` logs messages of different priority
- `log (msg)` logs messages with default priority "2"

Example: Dummy (2)

- **Problem**: Where does LogServer log?
  - For initial purposes, the file system is provided
  - Relational database could be another option
  - Both options are not optimal for testing the LogServer.
- **Solution**:
  - Hide log medium behind interface **logger**

Example: Dummy (3)

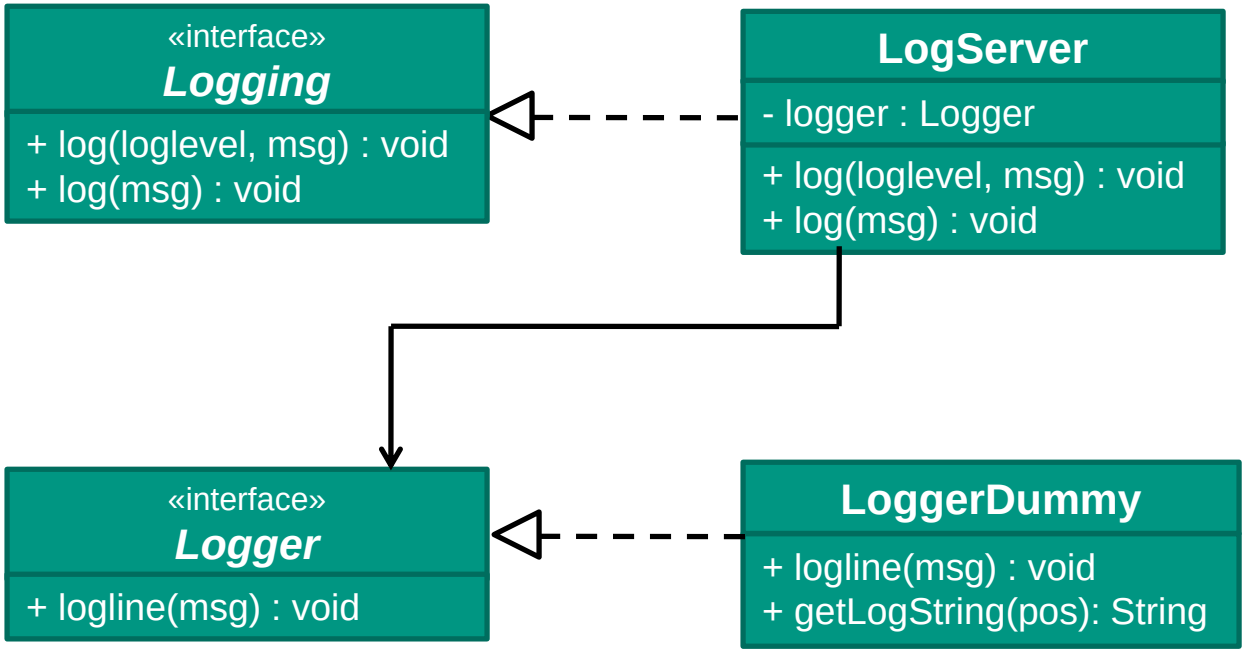


Example: Dummy (4)

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

- **How good is that for testing?**

Example: Dummy (5)



Example: Dummy (6)

- `LoggerDummy` implements the logger interface for `LogServer`, and
- `getLogString` for the test
  - `getLogString (pos)` provides the pos-th (e.g. 4th) message
- Now we can write a test for the LogServer:

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

   logServer.log(0, “first row");
   logServer.log(1, “second row");
   logServer.log(“third row");

   assertEquals("(0): first row ",
       logger.getLogString(0));
   assertEquals("(1): second row ",
       logger.getLogString(1));
   assertEquals("(2): third row ",
       logger.getLogString(2));
}
```

Example: Dummy (7)

- `LogServer`, which is independent of log medium.
- Interface for testing our `LogServer`
- Check the `LogServer` without explicitly visualizing the implementation.
- **Question**: What can be improved here?
- **Answer**: Test code can migrate to `LoggerDummy` ➔ It becomes a `loggerMock`

Example: Mock object (1)

```
public class LoggerMock 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 contents.
- Called at the end of a test.

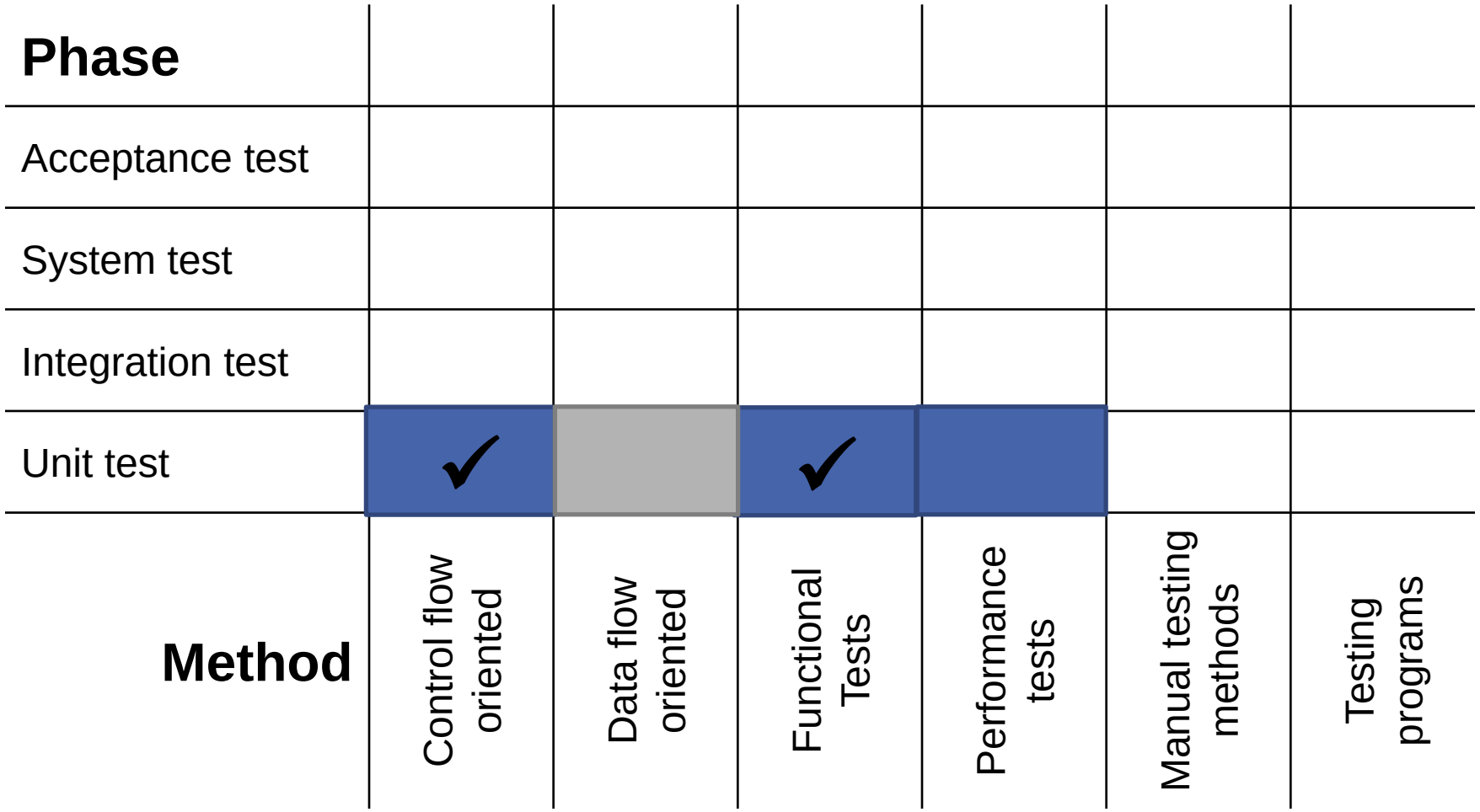
Example: Mock object (2)

- Test code:

```
LoggerMock logger;
@Test
public void testSimpleLogging() {
  logger.addExpectedLine("(0): first row");
  logger.addExpectedLine("(1): second row ");
  logger.addExpectedLine("(2): third row ");
  logServer.log(0, " first row ");
  logServer.log(1, " second row ");
  logServer.log(" third row ");
  logger.verify();
}
```

- Errors are only reported when calling **verify ()**
- For more complicated behavior, finding the error cause is difficult

Overview Matrix



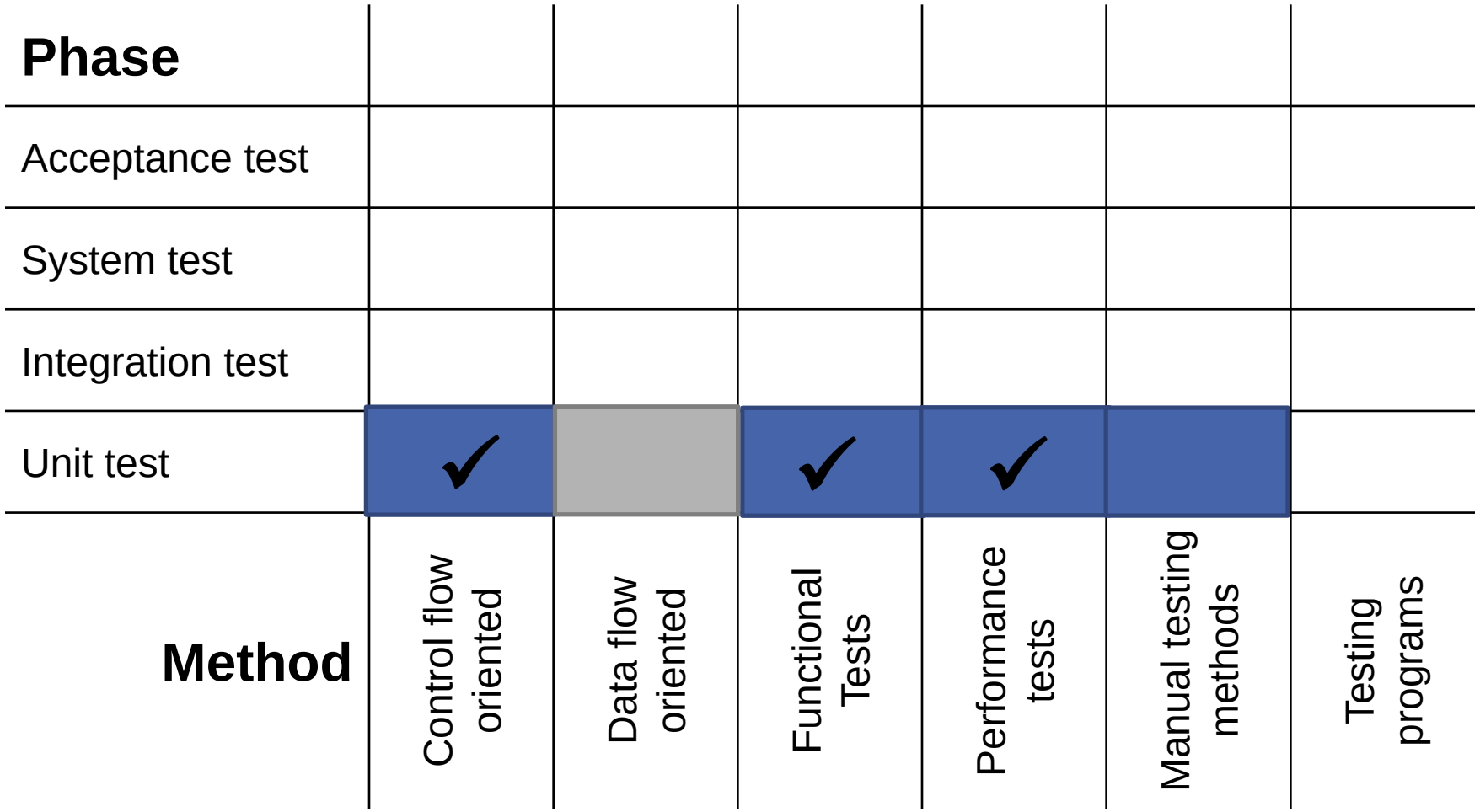
Performance tests: Load tests

- Tests the system / component for reliability and compliance with the specification **within the allowed limits**
  - Can the system serve the required number of users?
  - Can guaranteed response times be kept?
  - Can this load be managed 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 under overloading?
    - Thrashing
    - Exponential increase of the response time
    - Downtime
- Does the system return to defined behavior after the overload has dropped?
  - How long does it take?

Overview Matrix



Manual testing (review) methods

- Only way to check the semantics
- Expensive (up to 20% of the development costs)
- Time for the reviews (checks) must be scheduled
- Psychological pressure on the assessment of an individual's work by a group (dock, defense)

→ Consideration of such **social** conflict potentials in the review process:

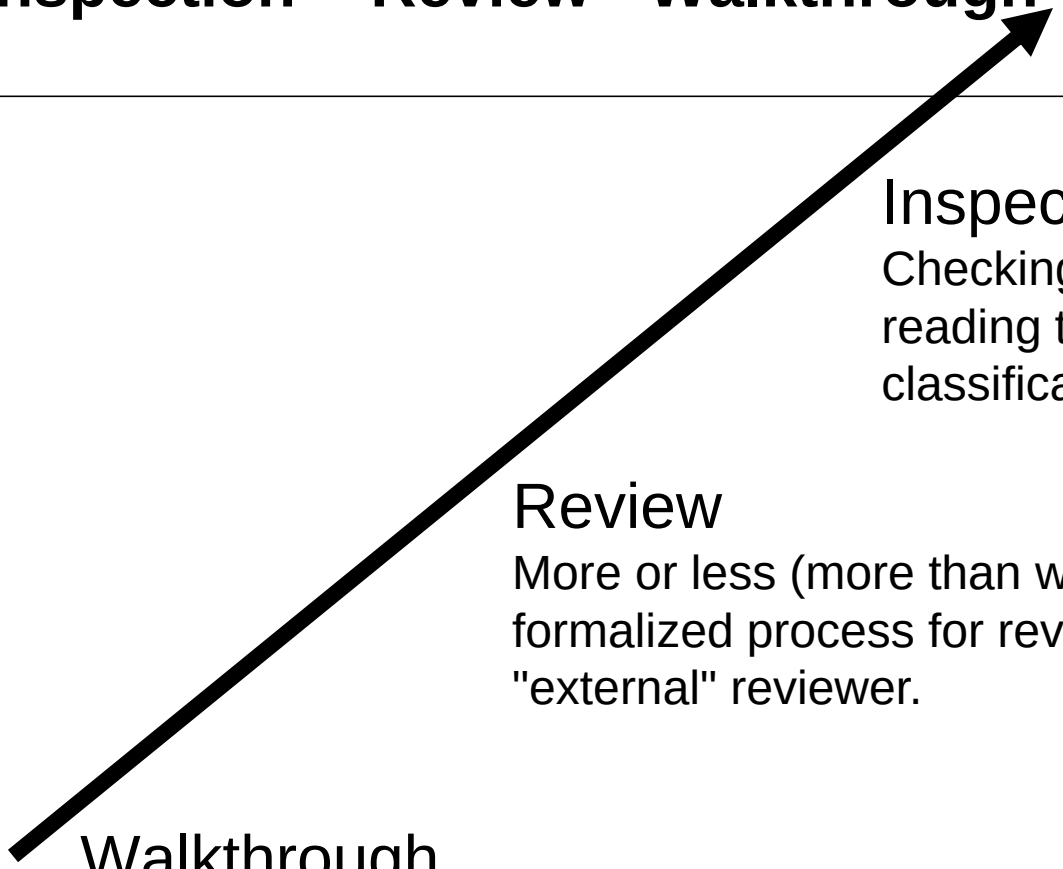
- The work, not the person is examined
- Regularly assess the work of all employees
- If possible, no customers / senior managers present during review

Software inspections – Overview

- Several inspectors (2 to 8) **independently** investigate the same artifact (software document)
- Found defects are written down and commonalities are **discussed**
- Main focus is to identify problems; solve later
- Structured process
  - Roles and forms
  - Checklists and perspectives

} Adaptation to document type

Inspection - Review - Walkthrough



Inspection

Checking based on checklists or reading techniques (e.g. defect classification)

Review

More or less (more than walkthrough, less than inspection) formalized process for reviewing written documents by an "external" reviewer.

Walkthrough

The developer guides one or more colleagues through part of the code or design he has written. Colleagues ask questions and comment on style, defects, compliance with development standards, and other issues. (according to ANSI / IEEE 729-1983)

Definition: Inspection

- Inspection is a formal **quality assurance** technique in which requirements, design or code are thoroughly reviewed by a person or group of persons other than the author.
  - The purpose is to find bugs, development standards violations and other issues (according to ANSI / IEEE Standard 729-1983).

Advantages & disadvantages of inspections

- **Advantages**
  - Applicable to all software documents: requirements, specifications, designs, code, test cases, ...
  - At any time and early feasible
  - Very effective in industrial practice
- **Disadvantage**
  - Manual process
  - Consume time of several employees
  - Expensive
  - "Static" (as opposed to testing)

IBM
Hewlett Packard
Motorola
Siemens
NASA

Paying for benefits and costs

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

Reading Techniques - Checklists: Example

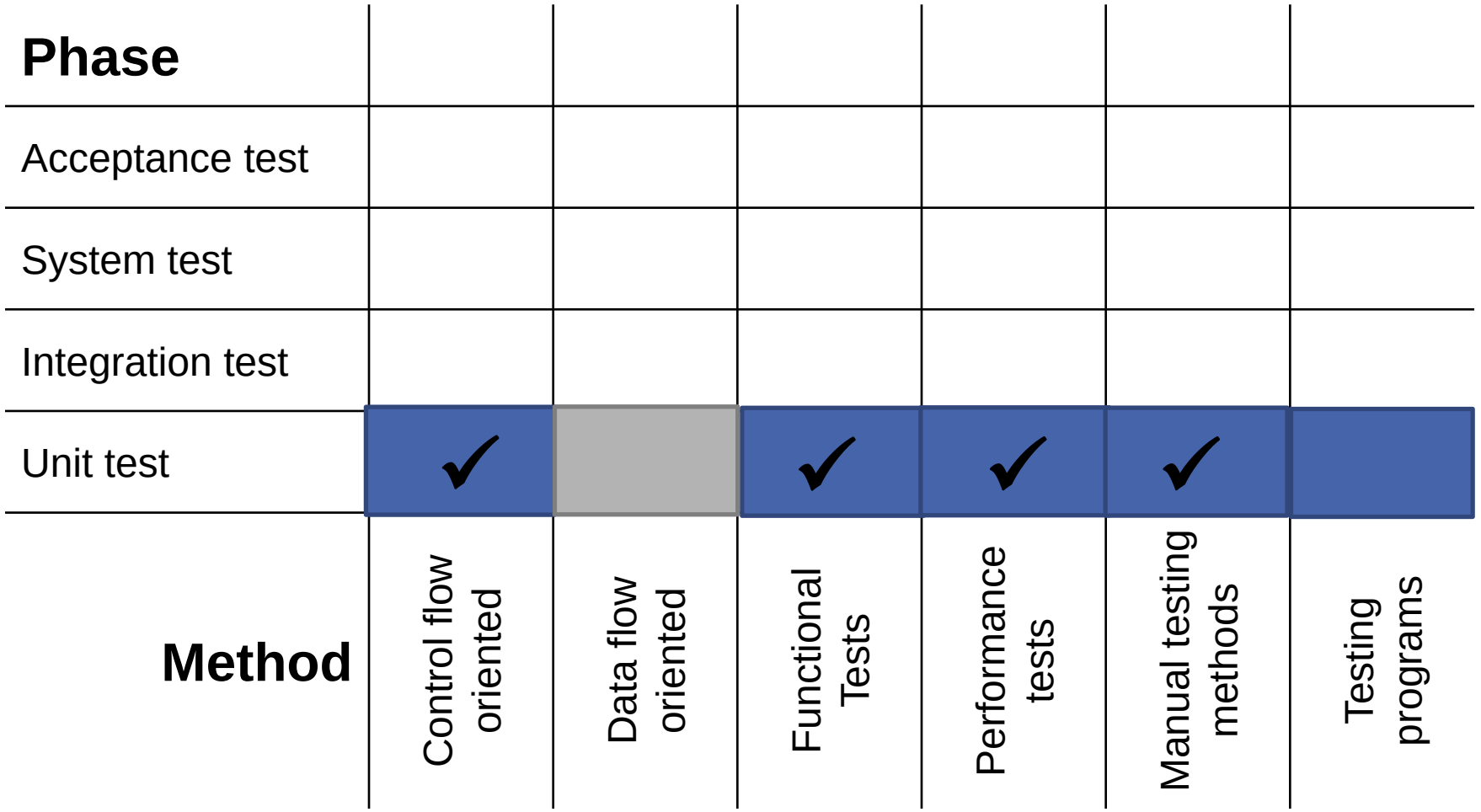
- A **possible** checklist for Java code could be the following defect classes:
- Variable, attribute and constant declarations
- Method definitions
- Class definitions
- Data references
- Calculations
- Comparisons
- Control flow
- Input/output
- Module interface
- Comments, documentation
- Code formatting
- Modularity
- Memory usage
- Execution speed

Checklist by Christopher Fox, 1999.

Inspection support tools

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

Overview Matrix



Testing (analyzing) programs

- The static analysis of a software takes place during the compilation of the source code.
- For the static analysis of the source code there are special applications and plug-ins for many development environments.

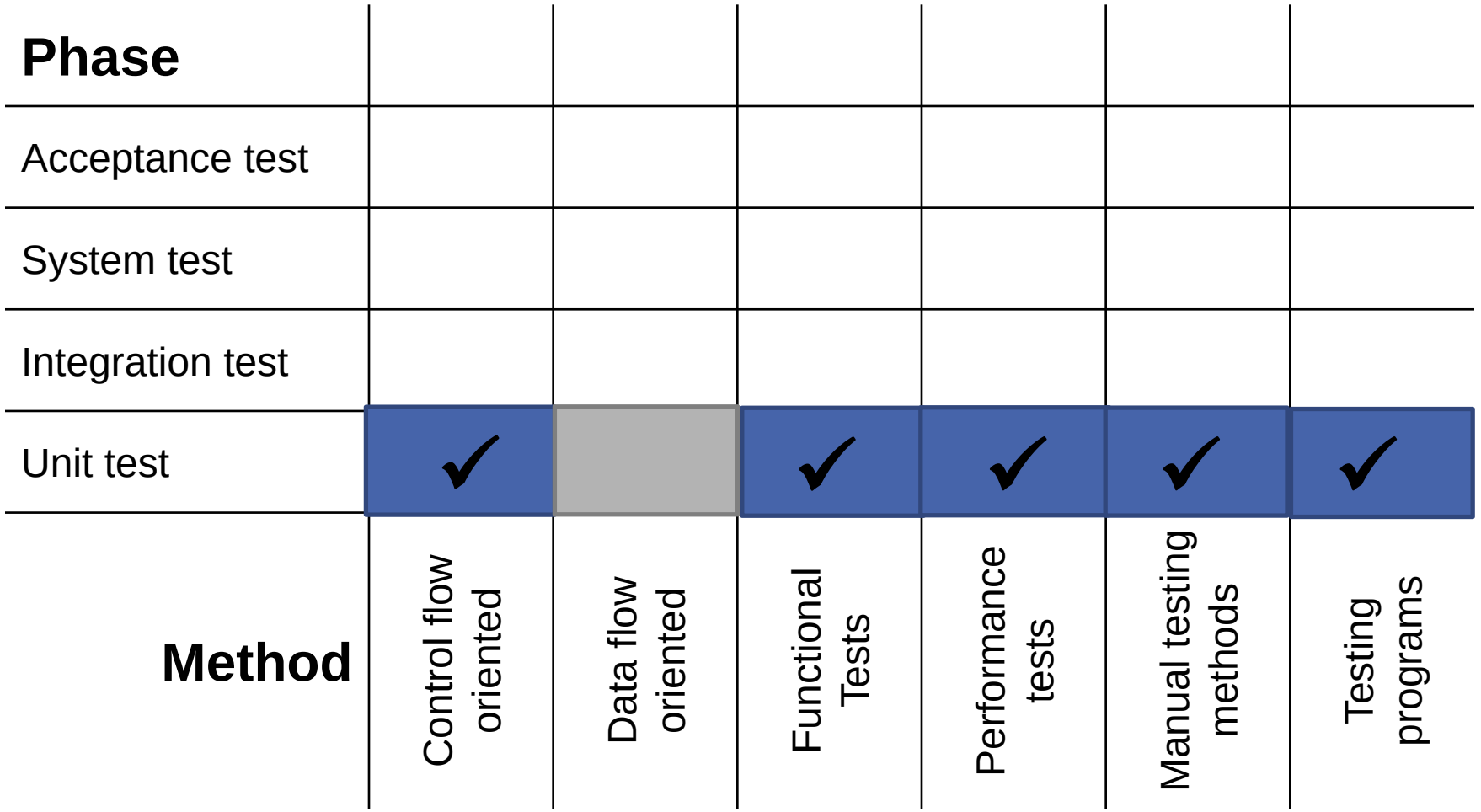
Testing (analyzing) programs

- Warnings and errors
  - They are displayed by the development environment. These include, for example:
    - Possibly uninitialized variables
    - Unreachable instructions
    - Unnecessary instructions
- Check programming style
  - Indentation style (inconsistent or forgotten)
  - Forget JavaDoc comments
  - Parameter not declared final

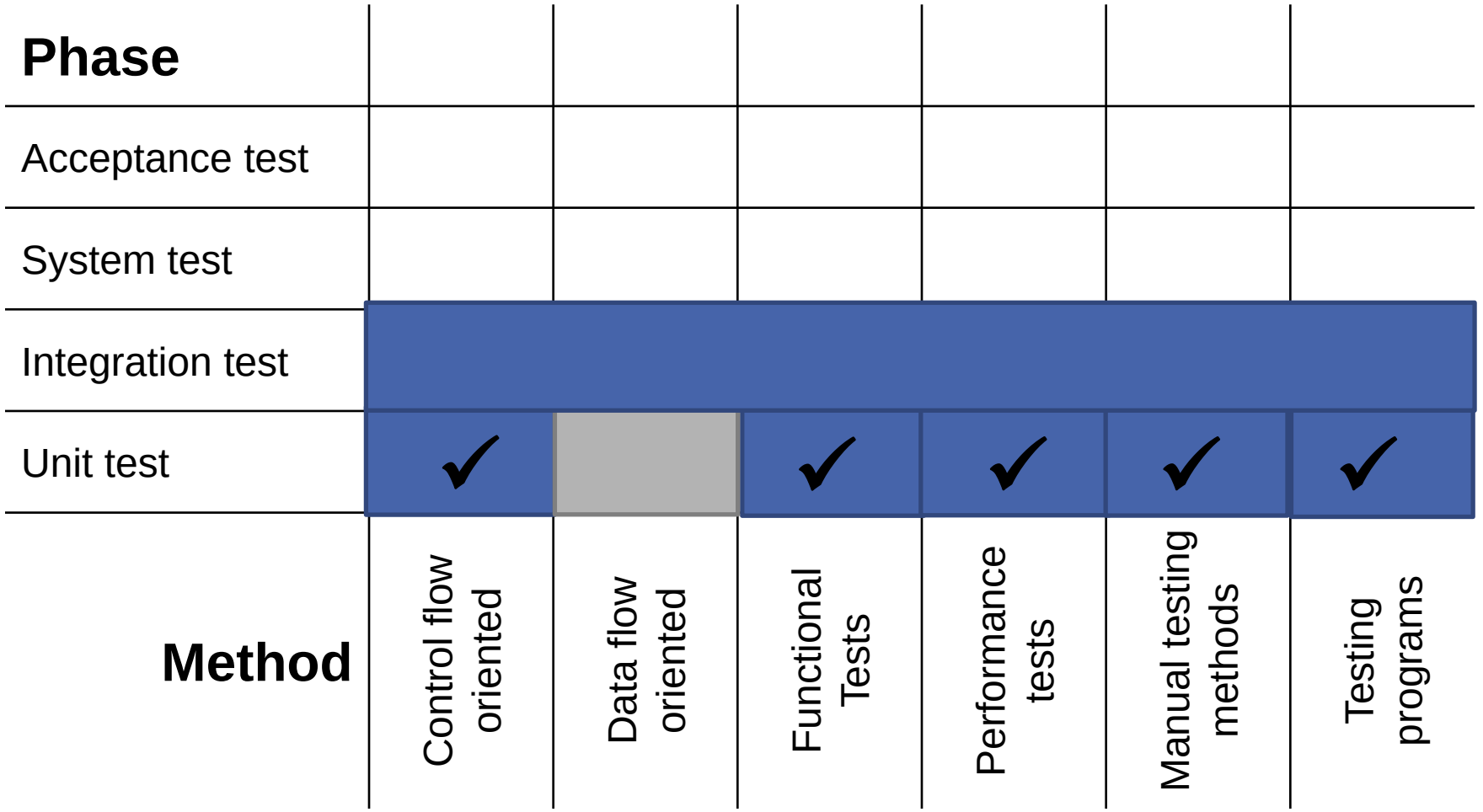
Testing (analyzing) programs

- Find errors based on error patterns
  - With the help of a static analysis, errors can be found based on specific patterns
- Various plug-ins for development environments and tools that provide the above functionalities

Overview Matrix



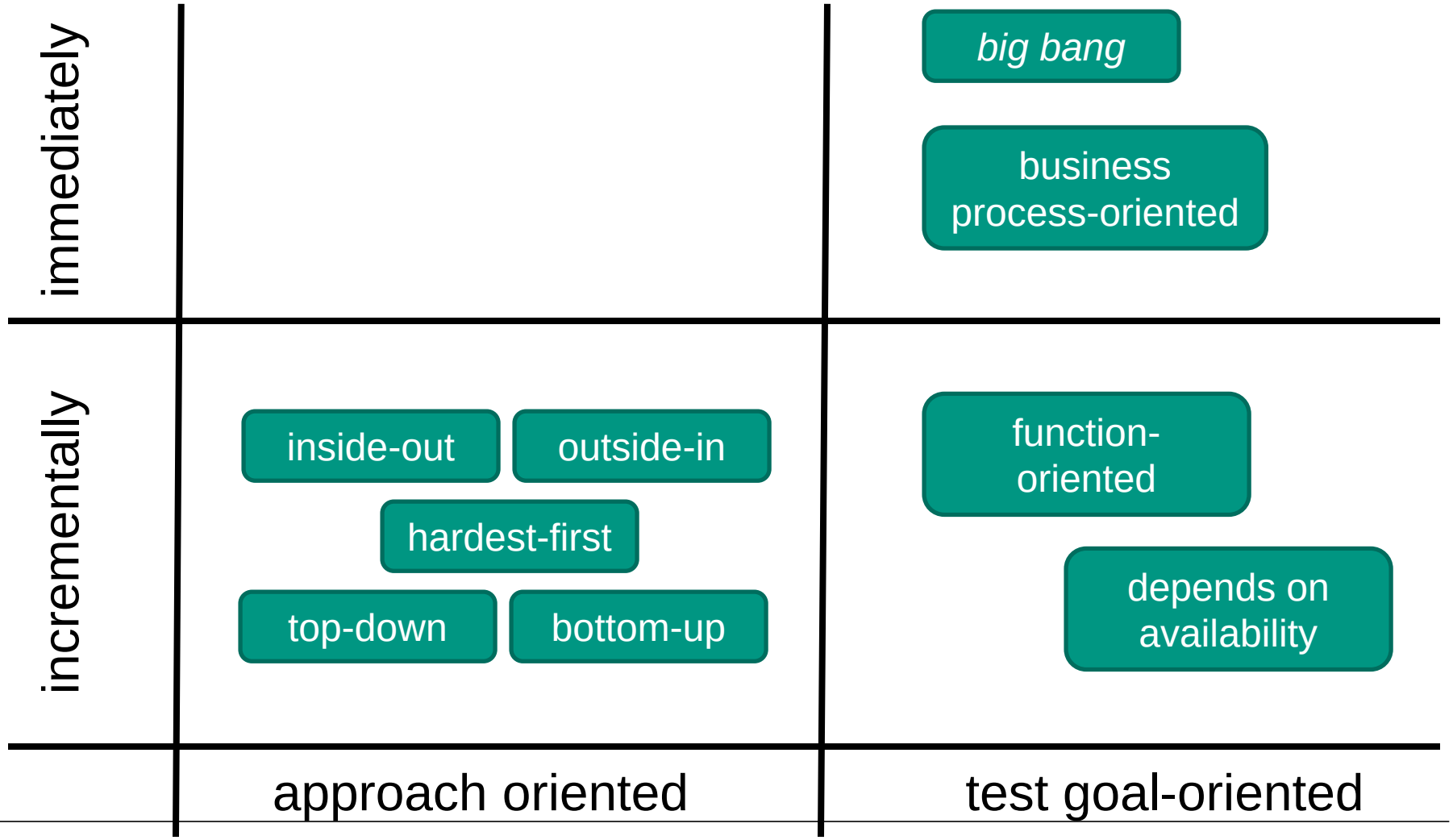
Overview Matrix



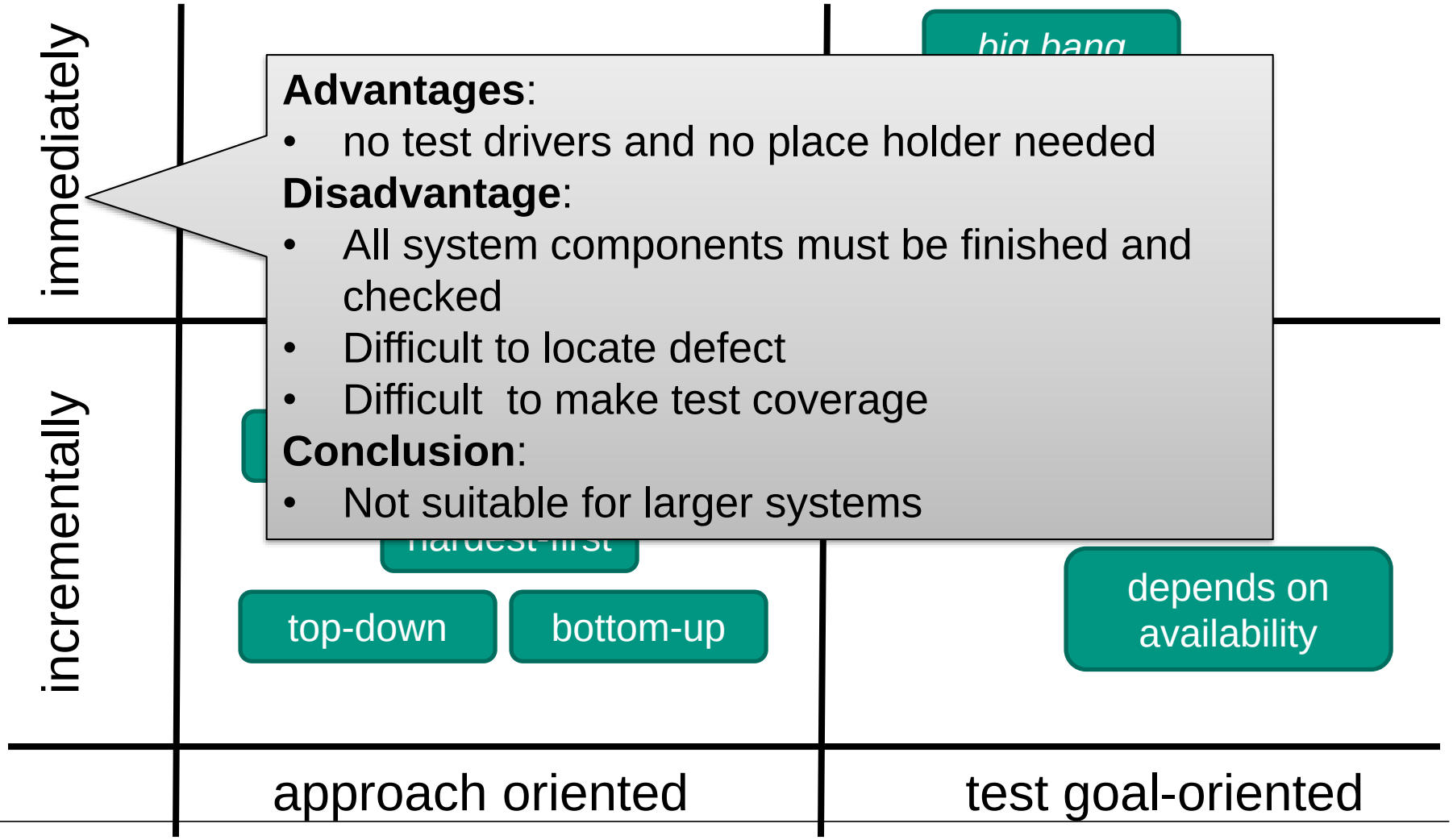
Integration test

- **Prerequisite**: Each component involved has already been checked for itself
- Gradual integration
  - Integrate another component into the already tested component set
  - ... and check again
  - ... until the system is completely integrated.
  - The interaction of the components is tested
  - Stubs and dummies are replaced with the implementations

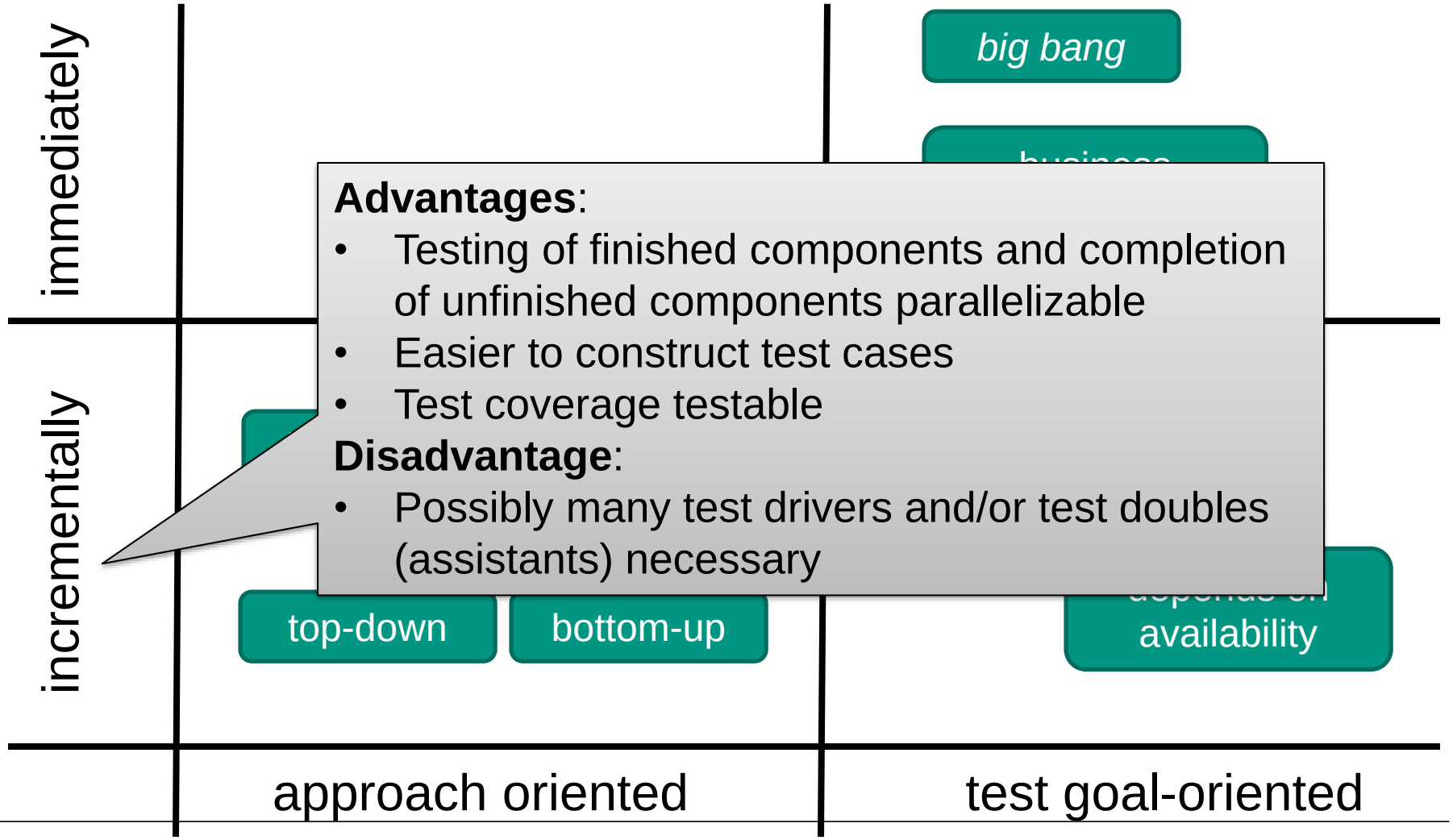
Integration strategies



Integration strategies



Integration strategies



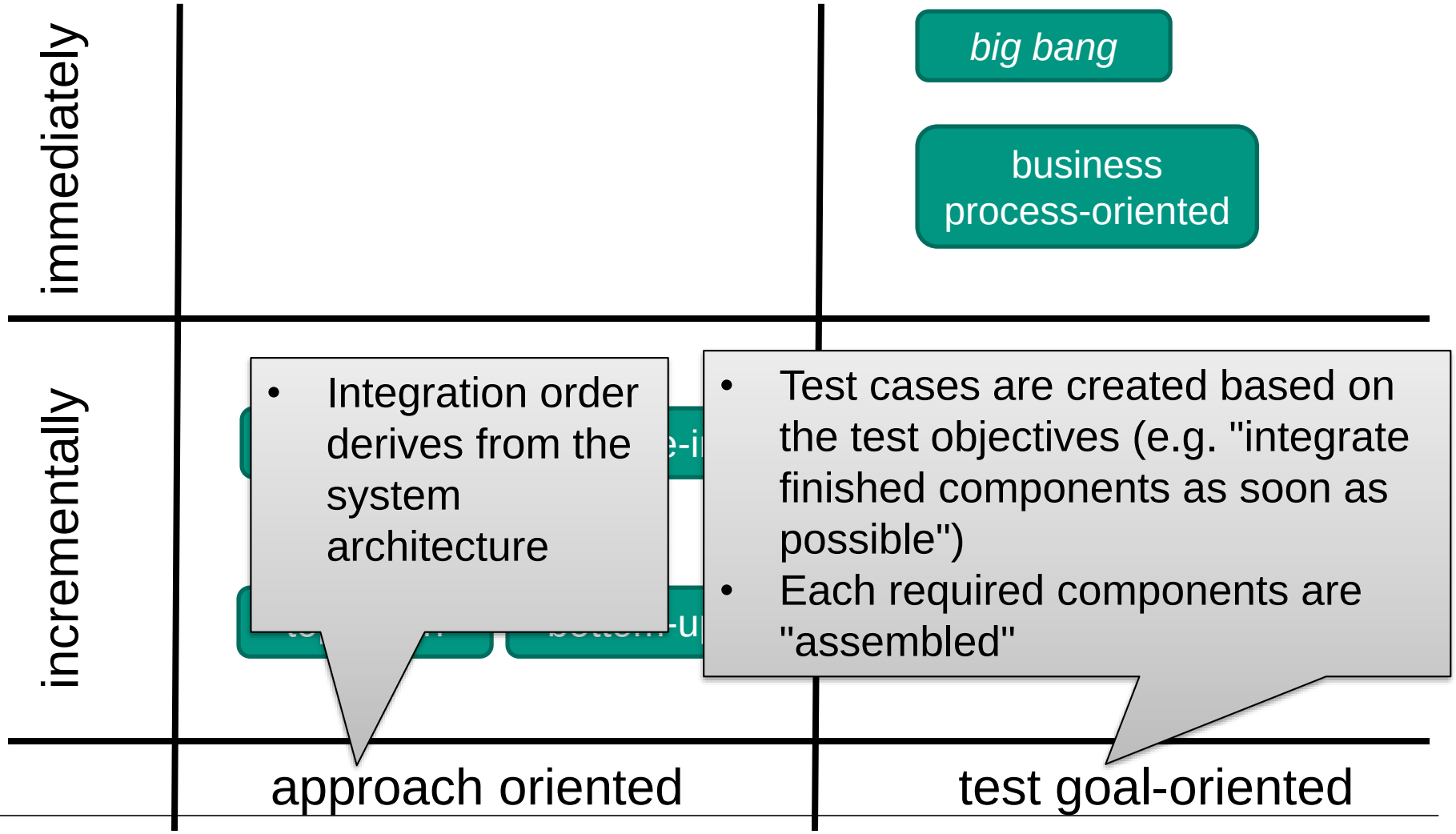
Advantages:

- Testing of finished components and completion of unfinished components parallelizable
- Easier to construct test cases
- Test coverage testable

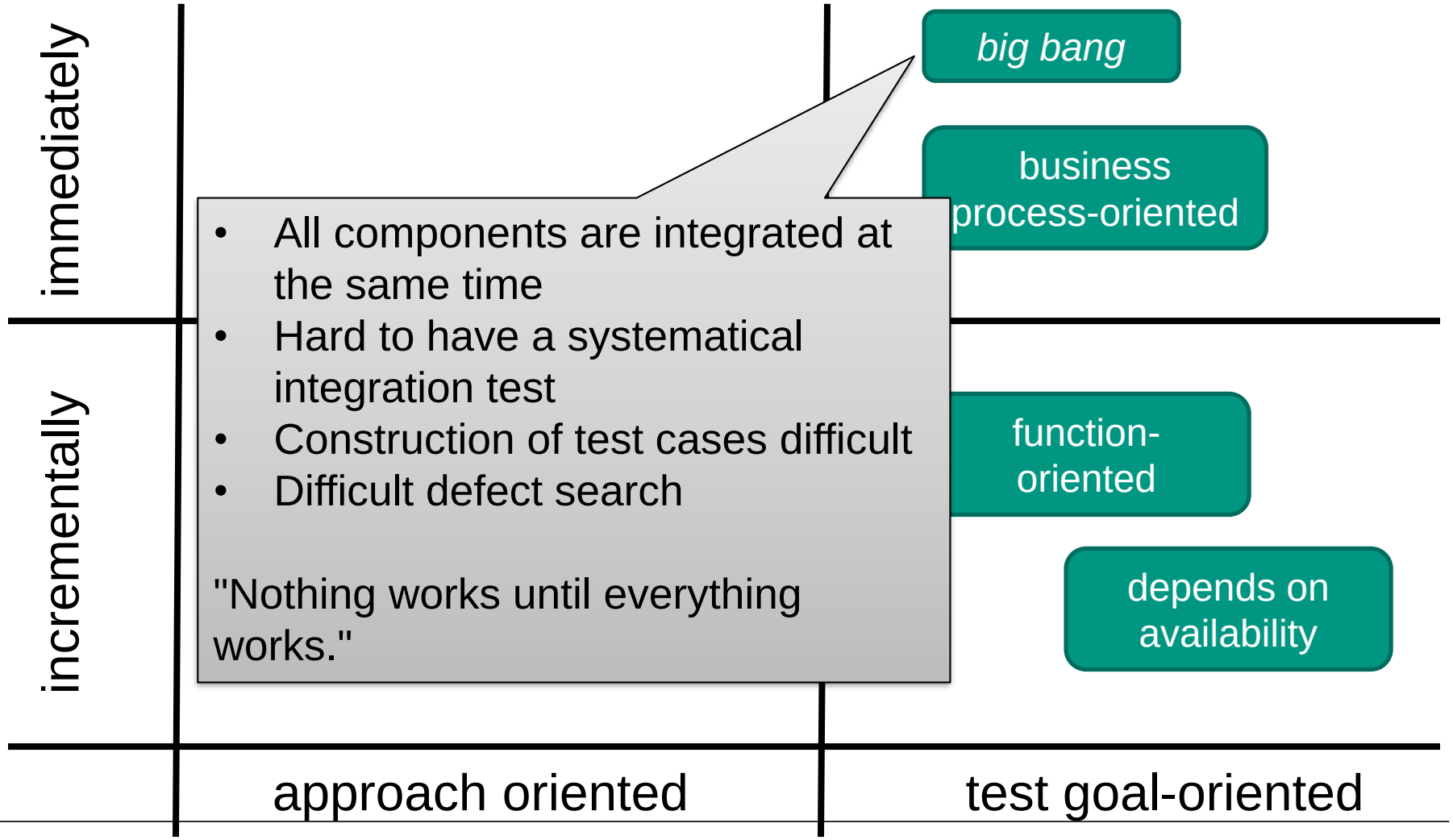
Disadvantage:

- Possibly many test drivers and/or test doubles (assistants) necessary

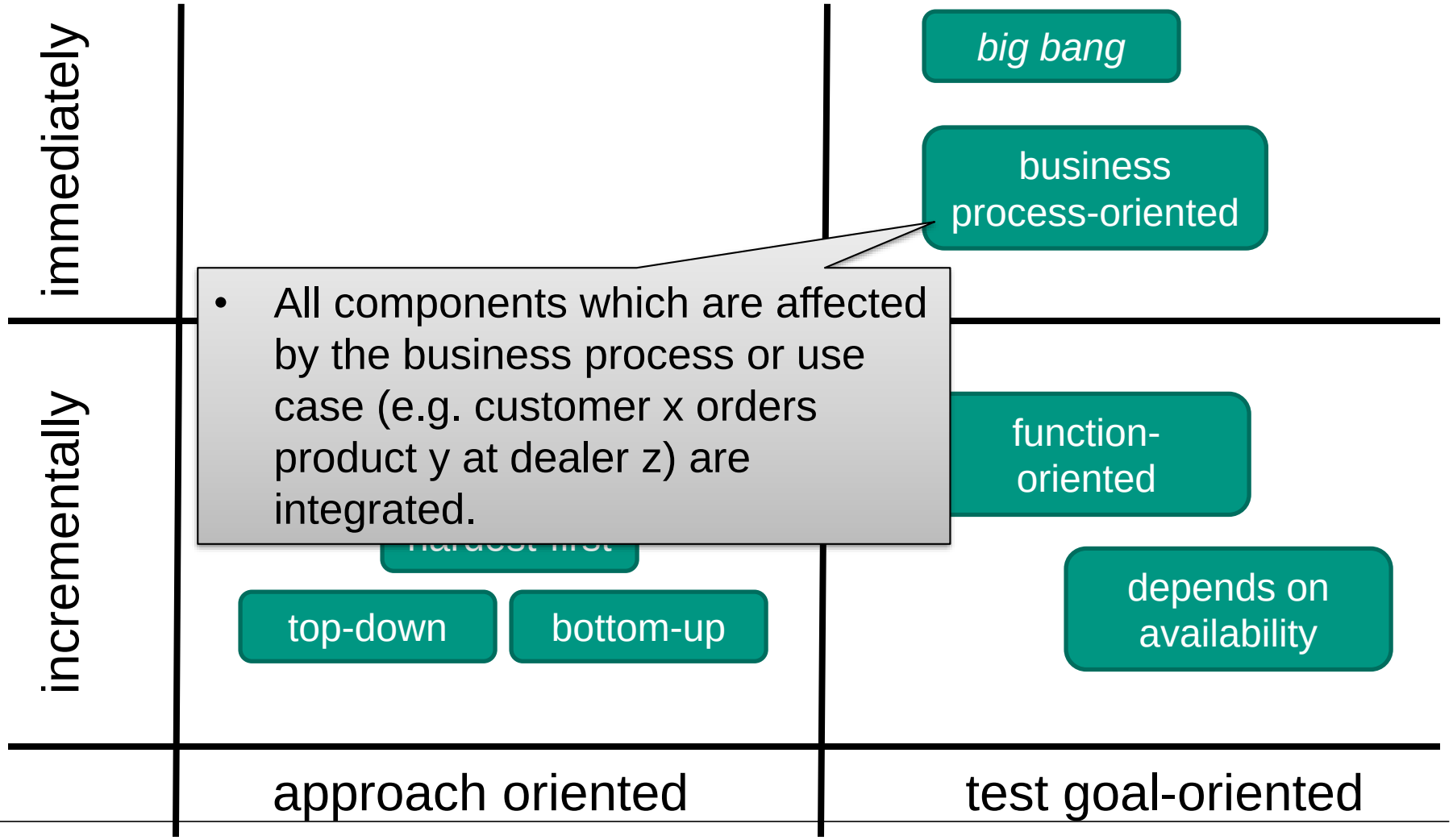
Integration strategies



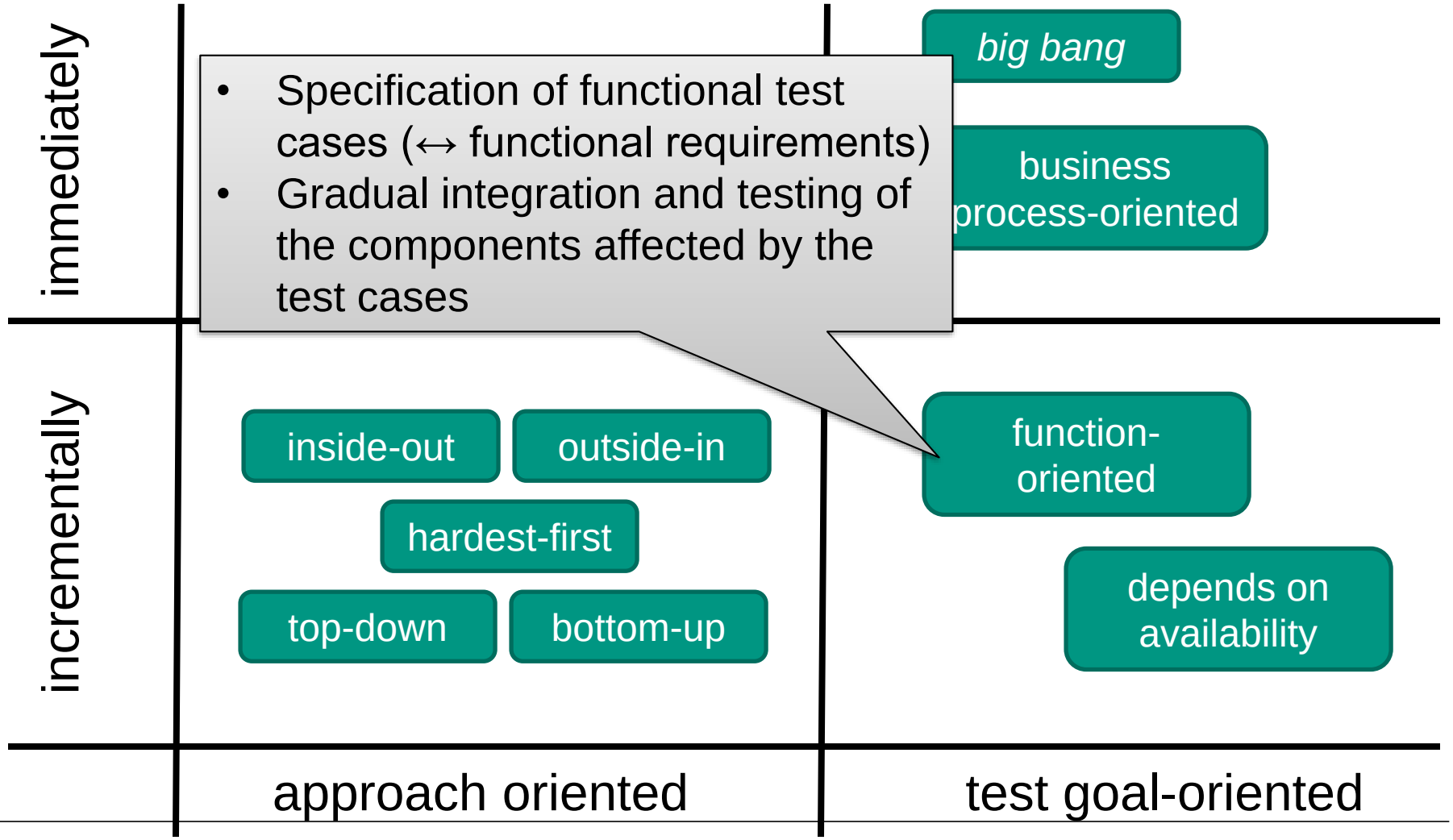
Integration strategies



Integration strategies

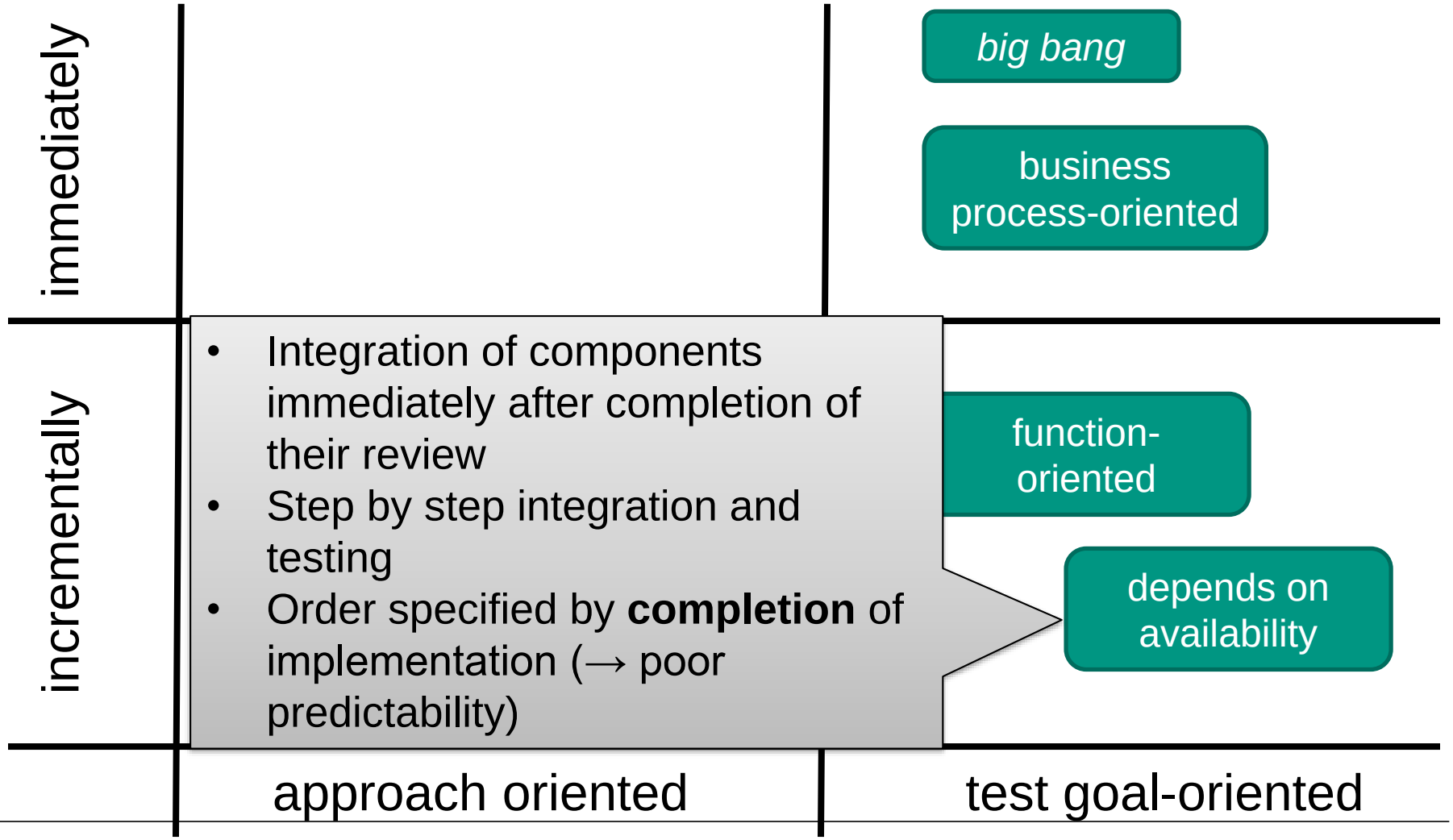


Integration strategies



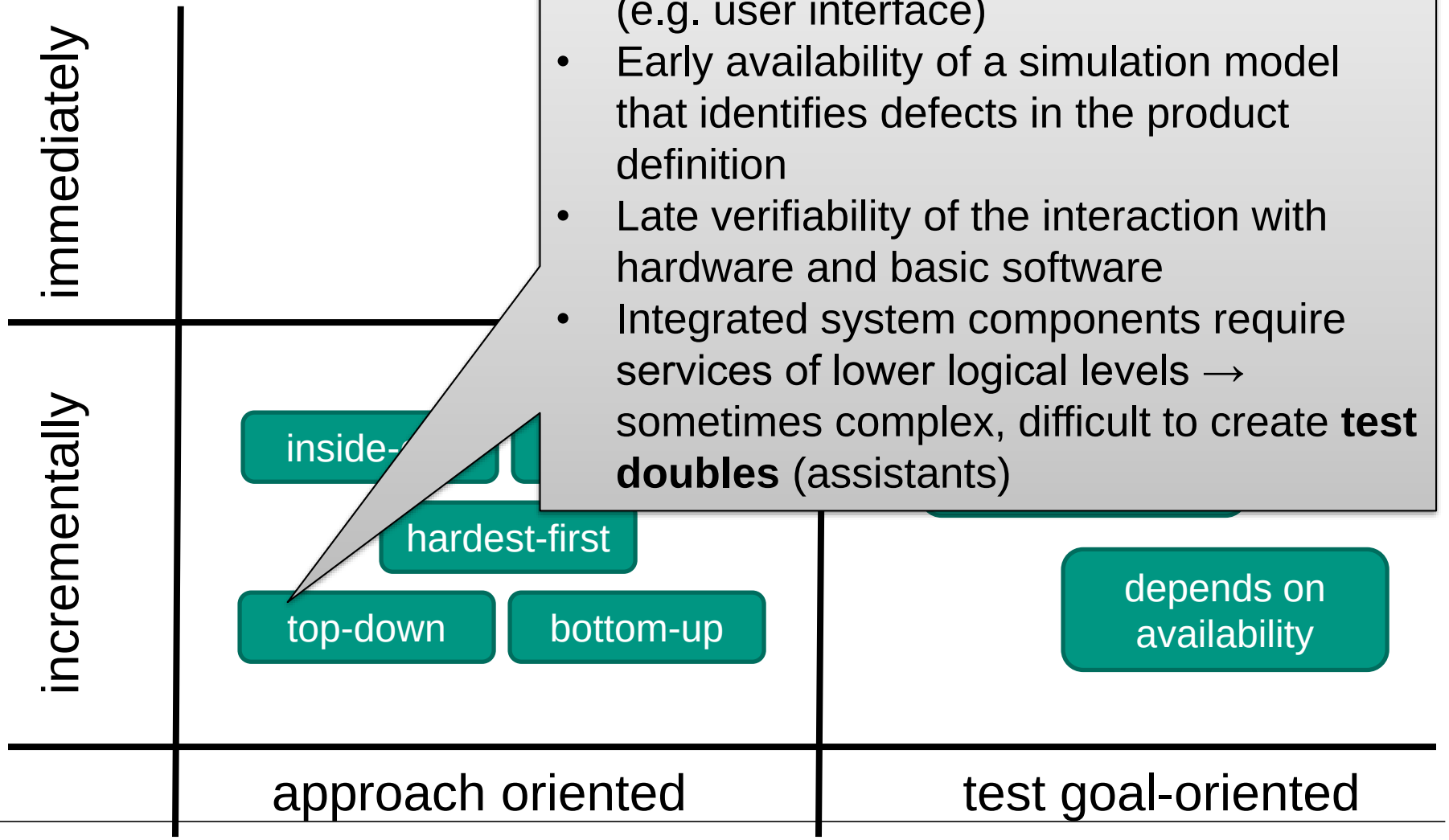
- Specification of functional test cases (↔ functional requirements)
- Gradual integration and testing of the components affected by the test cases

Integration strategies



Integration strategies

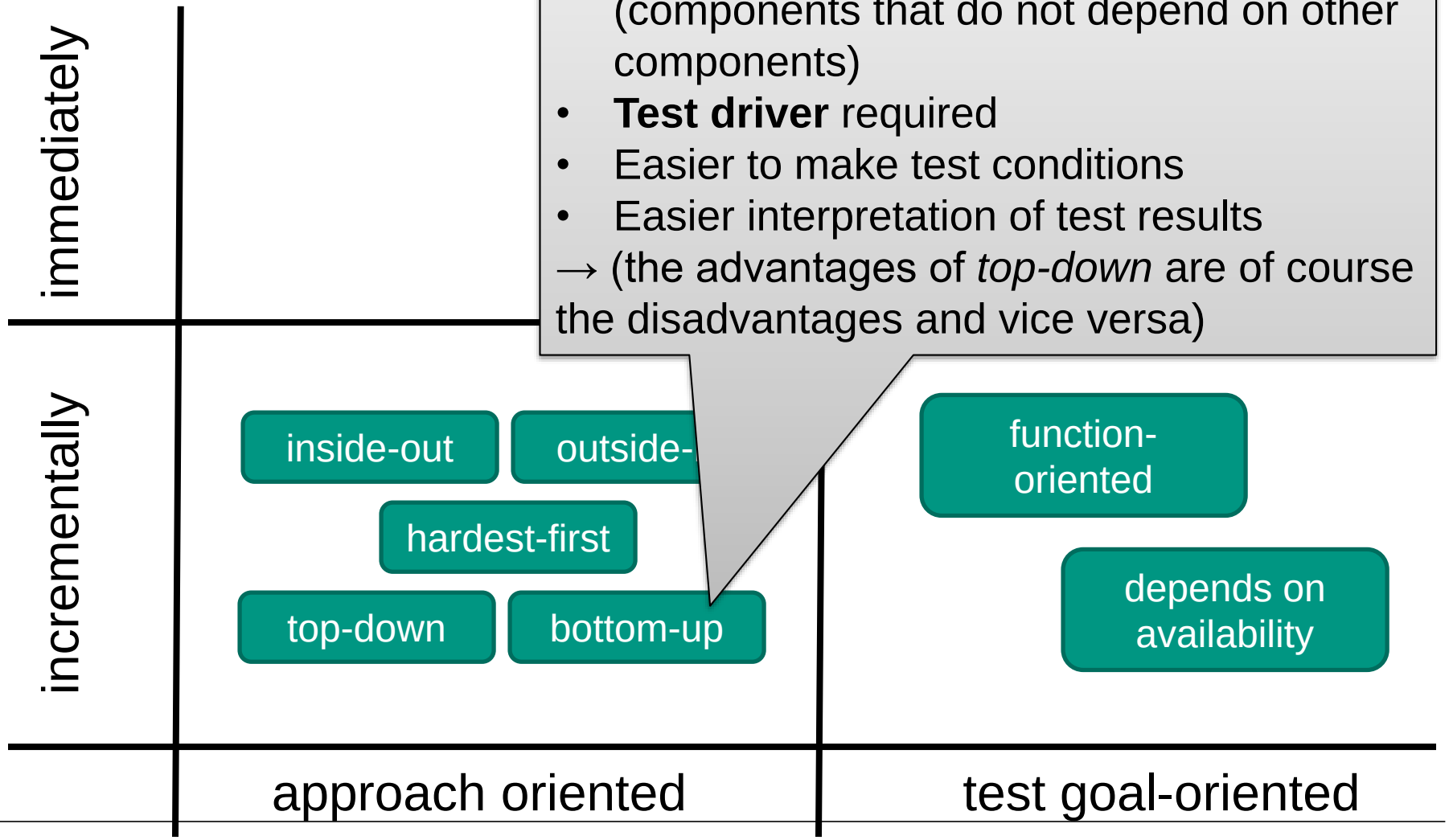
- Integration from the **highest logical level** (e.g. user interface)
- Early availability of a simulation model that identifies defects in the product definition
- Late verifiability of the interaction with hardware and basic software
- Integrated system components require services of lower logical levels → sometimes complex, difficult to create **test doubles** (assistants)



Integration strategies

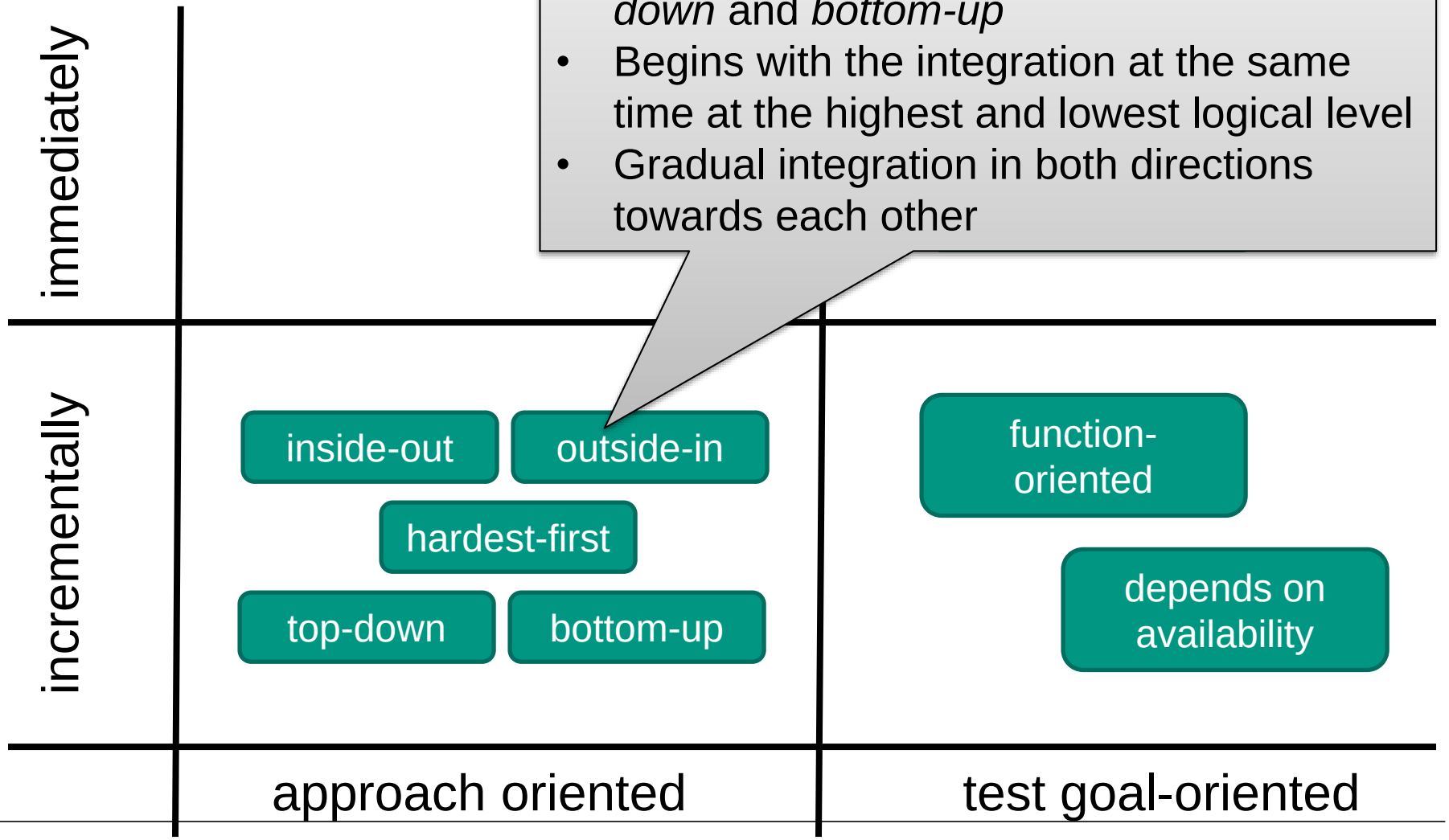
- Integration from **the lowest logical level** (components that do not depend on other components)
- **Test driver** required
- Easier to make test conditions
- Easier interpretation of test results

→ (the advantages of *top-down* are of course the disadvantages and vice versa)



Integration strategies

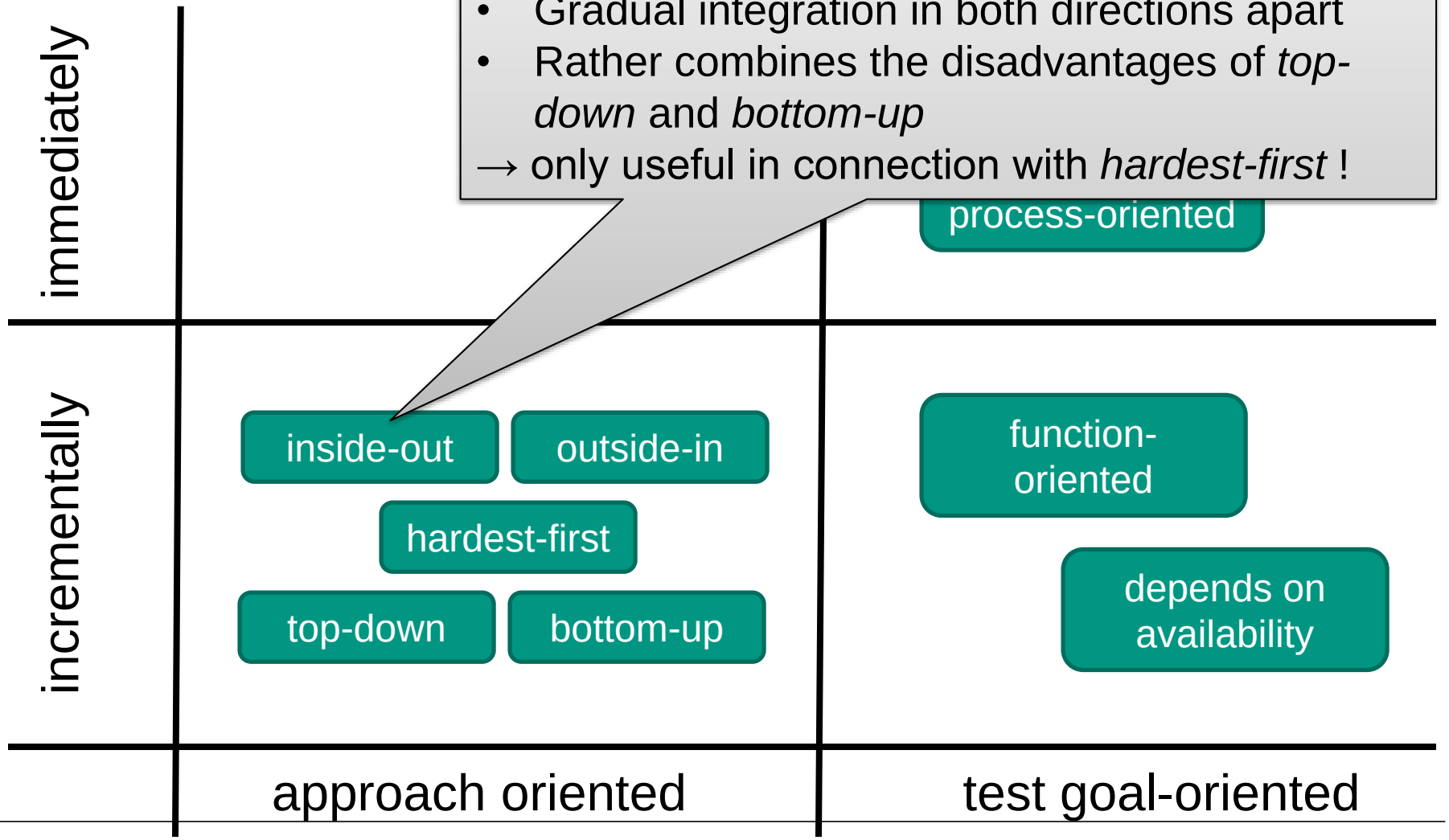
- Try to reduce the disadvantages of *top-down* and *bottom-up*
- Begins with the integration at the same time at the highest and lowest logical level
- Gradual integration in both directions towards each other



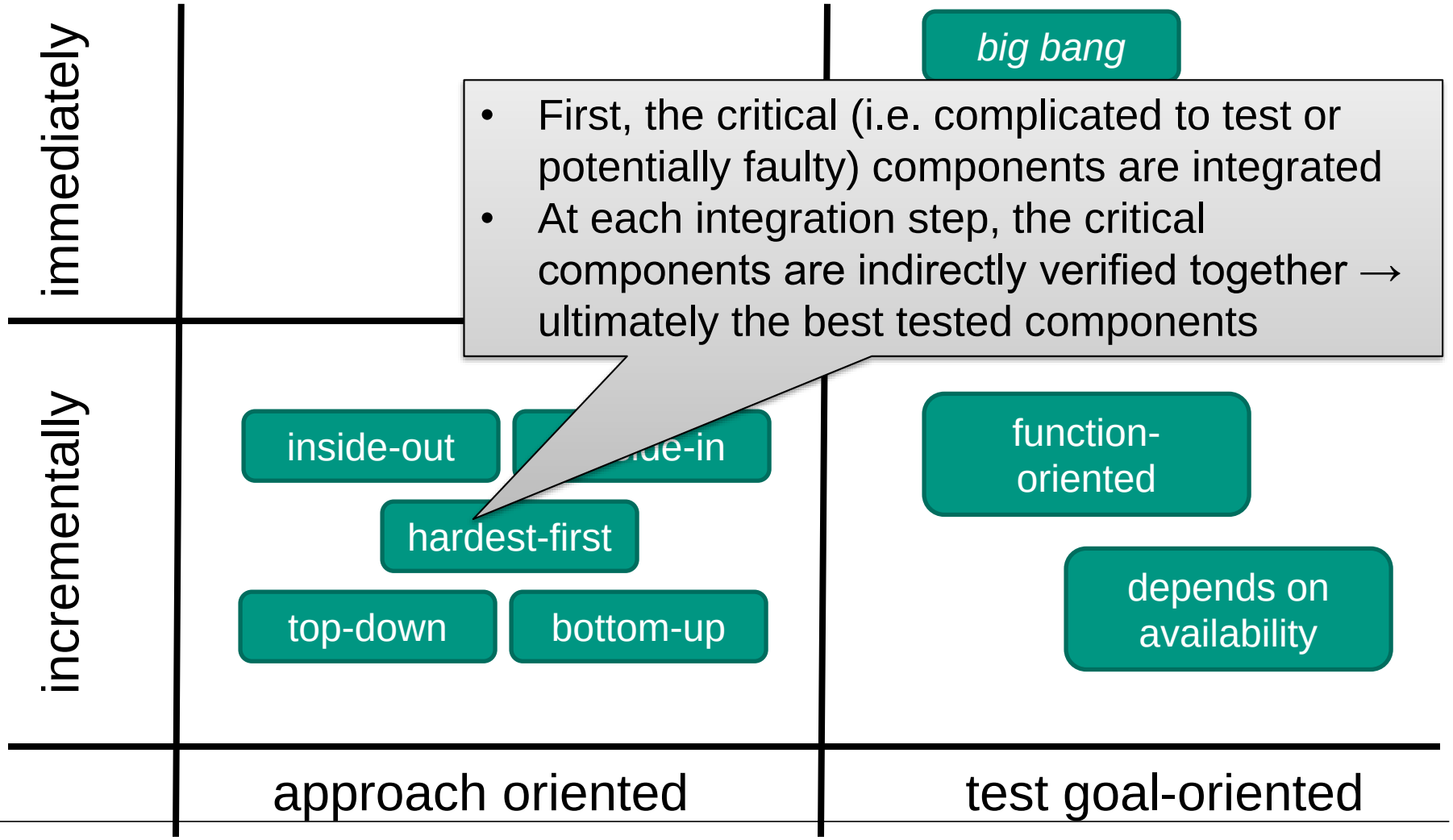
Integration strategies

- Begins at the middle level with the integration
- Gradual integration in both directions apart
- Rather combines the disadvantages of *top-down* and *bottom-up*

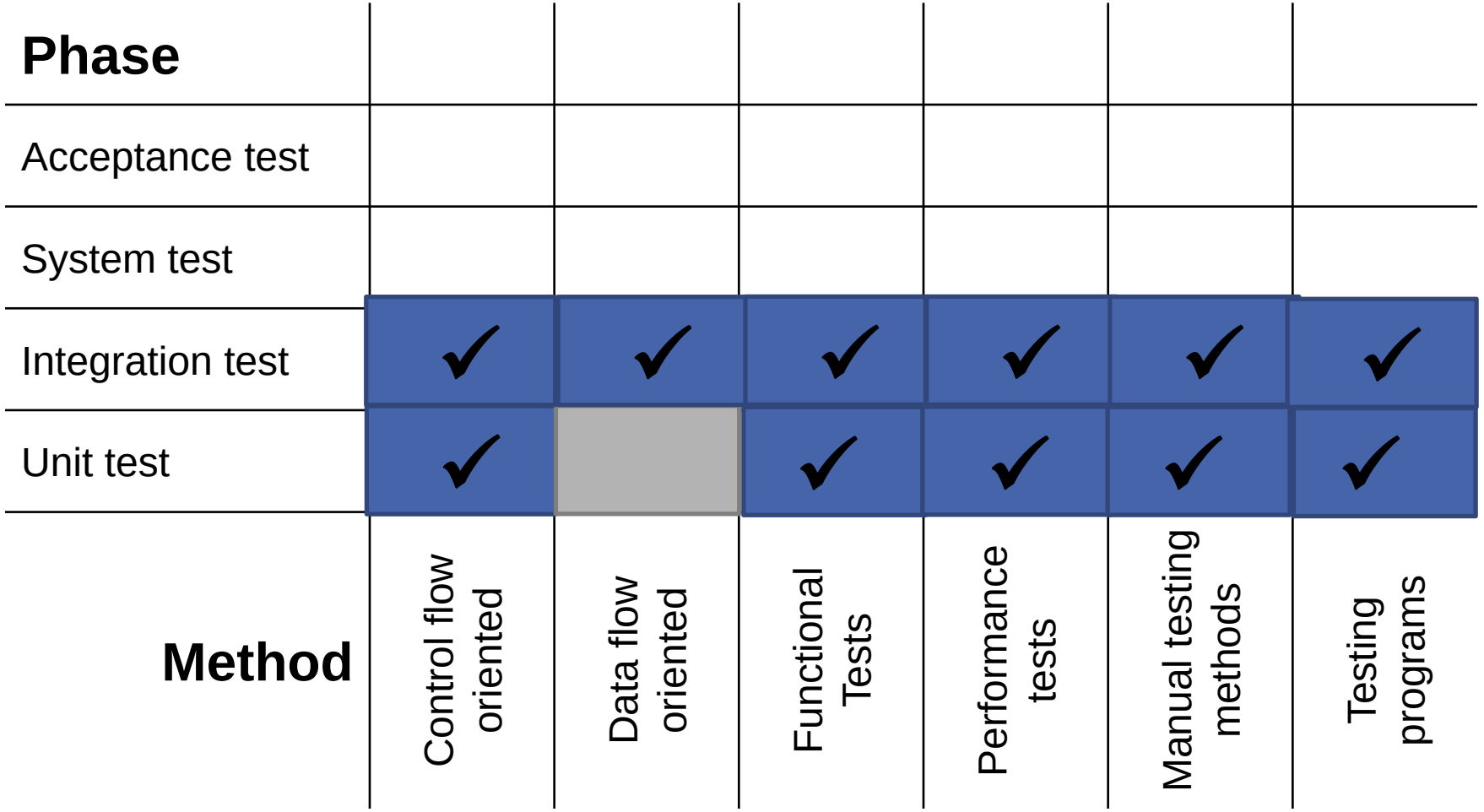
→ only useful in connection with *hardest-first* !



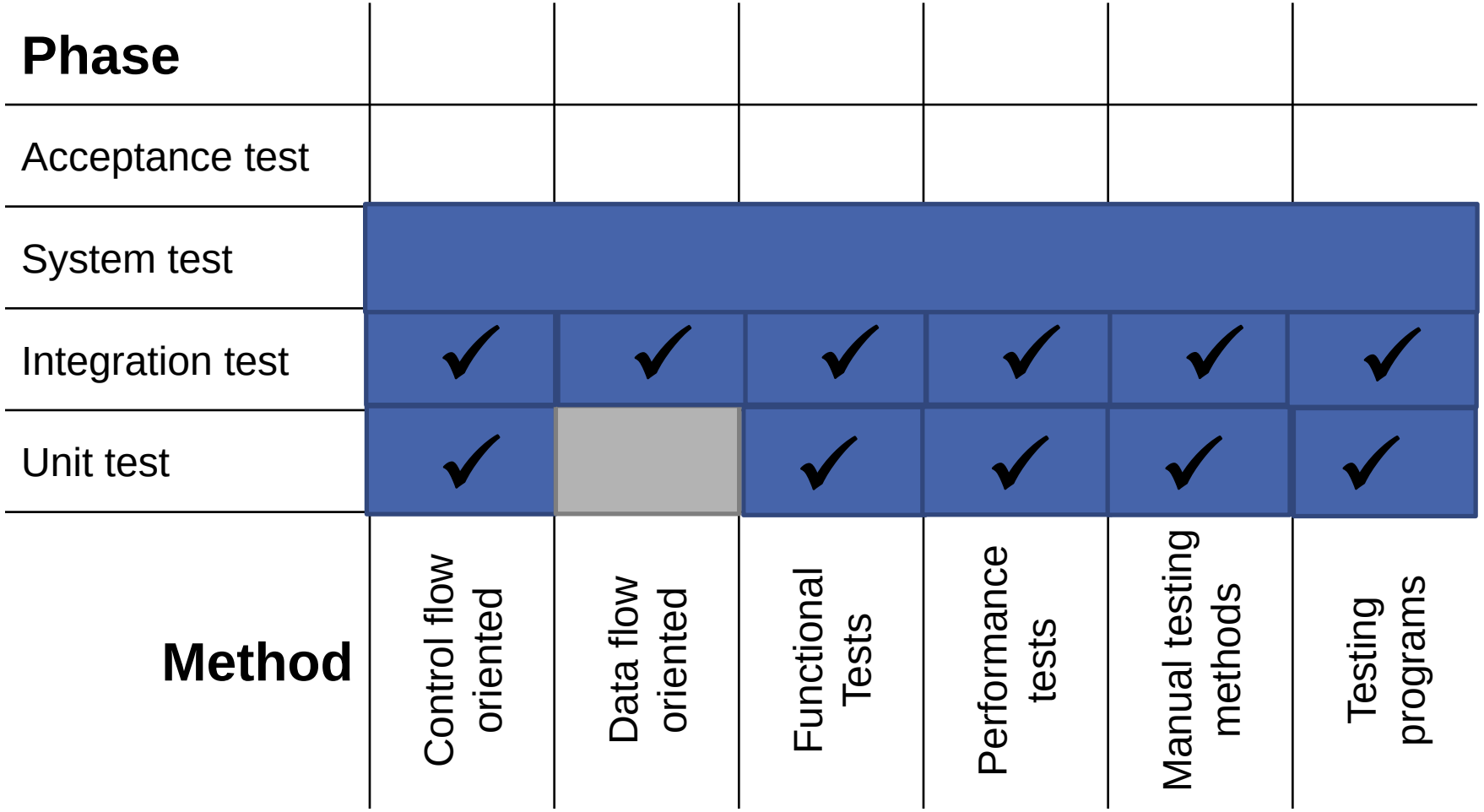
Integration strategies



Overview Matrix



Overview Matrix



System test

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

Classification of system tests

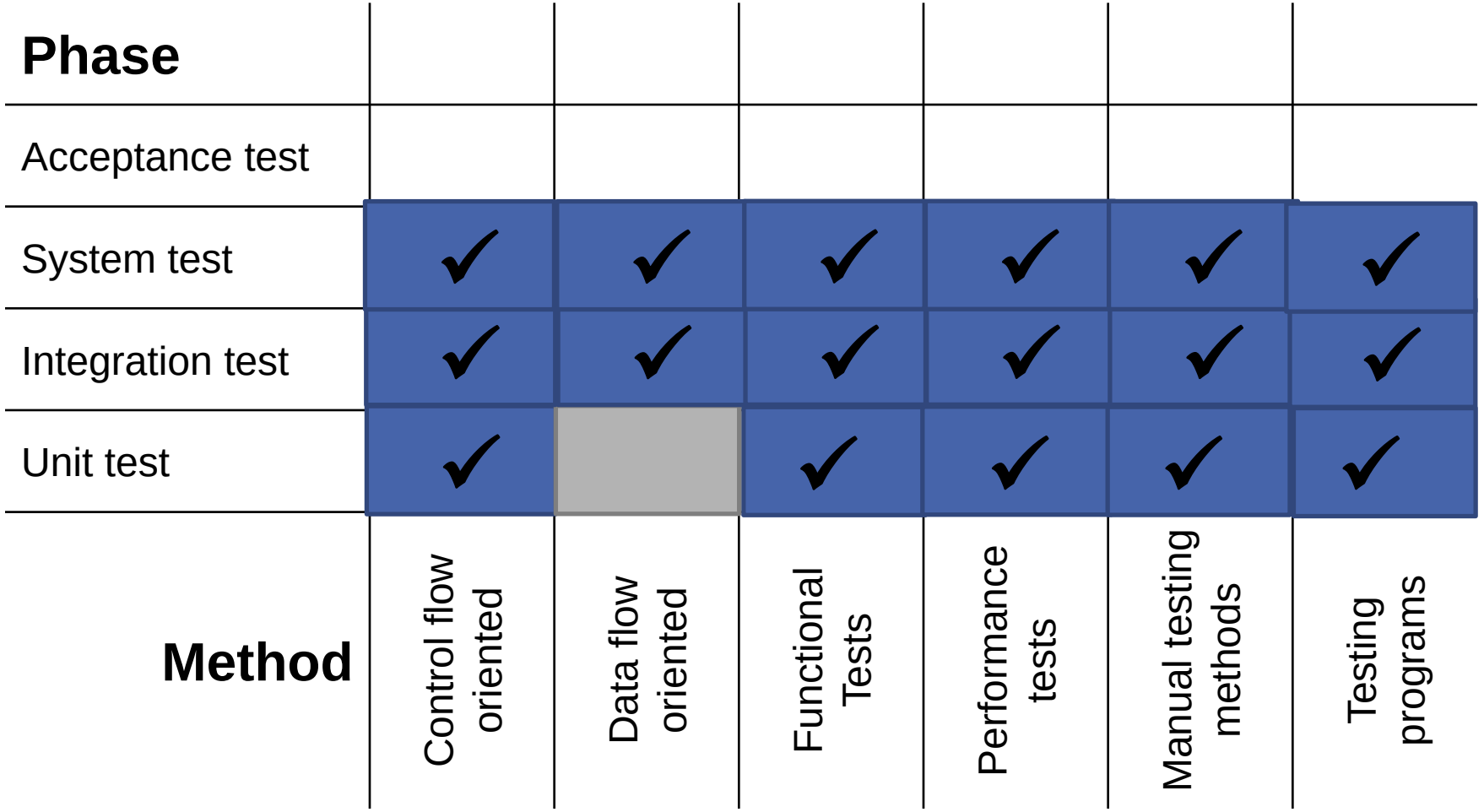
- The system test is divided into the functional and non-functional system tests.
  - **Functional system test:** Checking the functional quality **features** correctness and completeness.
  - **Non-functional system test:** Check non-functional quality characteristics, such as
    - Safety
    - Usability
    - Interoperability
    - Documentation
    - Reliability

… …and of course Performance tests!

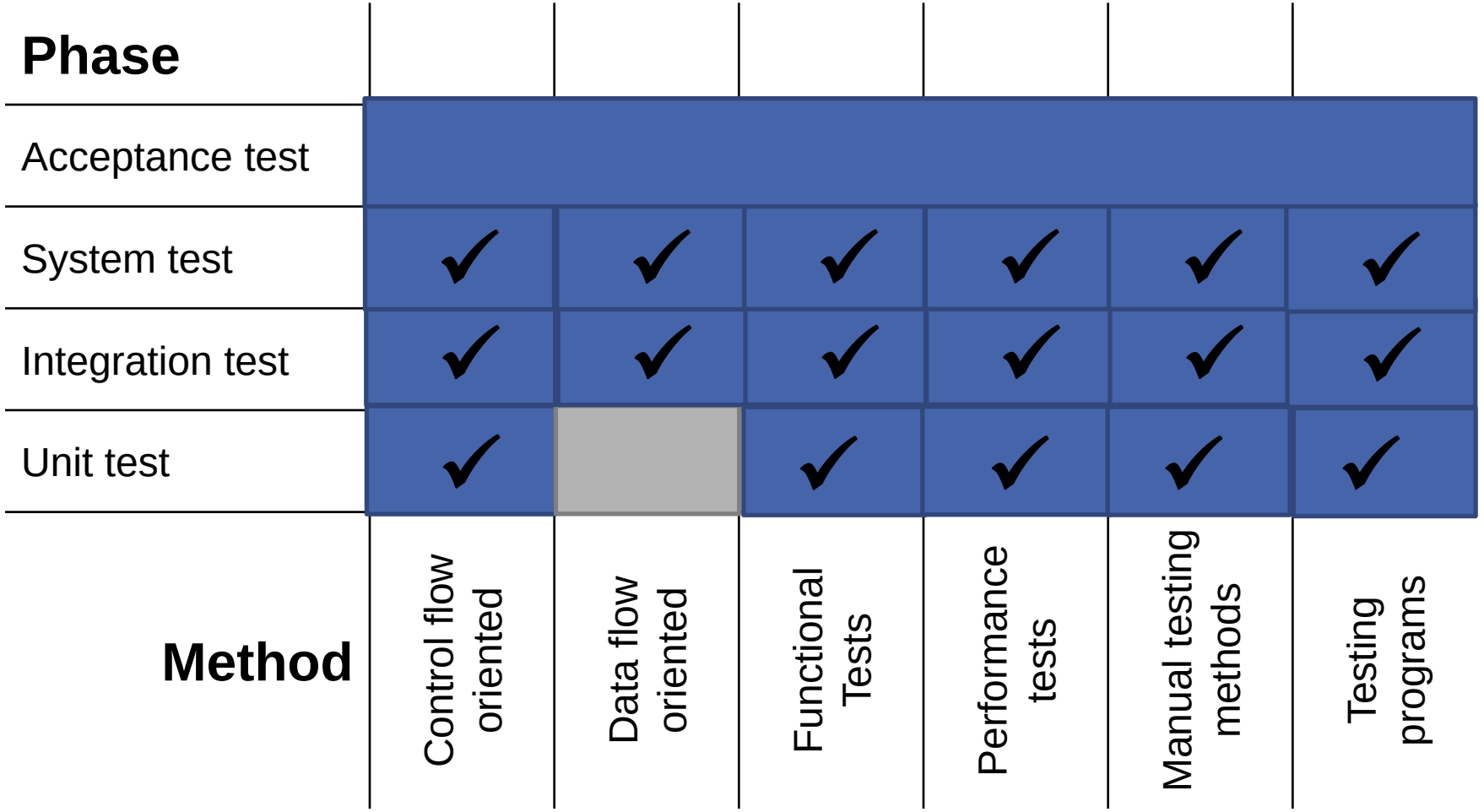
Definition: Regression test

- A **regression test** is the repetition of an already fully executed system tests, e.g. due to maintenance, modification and correction of the considered system.
- **Purpose**: to ensure that the system has not returned to a worse state than before.
- To simplify the test evaluation, the results of the regression test are compared with the results of the previous test.

Overview Matrix



Overview Matrix



Acceptance tests

- Special system test in which
  - The customer **observes**, **participates** and/or conducts
  - The real application environment used by the customer is used
  - And if possible real data of the customer are used
- The customer can **take over** and/or modify test cases of the system test and/or carry out own test scenarios, but ...
- ... the formal acceptance (for which the acceptance test is the basis) is the binding explanation of the acceptance (in the legal sense) of a product by the customer

Can testing prove bug-free programs?

- No, it cannot!!
- What are the implications of 100% path coverage testing?
- How to save regression testing effort?
- How to automatically generate test cases?
- … and many other questions and research topics on testing…

Summary

- Testing
- CF-oriented testing (CFO)
  - Statement coverage, branch coverage, path coverage
- Functional testing
- Performance testing
- Manual testing methods, etc.
- Integration test
- System test
- Acceptance test

References

- [BrDu04] B. Bruegge, A.H. Dutoit, Object-Oriented Software Engineering: Using UML, Patterns and Java, Pearson Prentice Hall, 2004, S. 435ff.
- https://www.tutorialspoint.com/software_engineering/software_testing_overview.htm