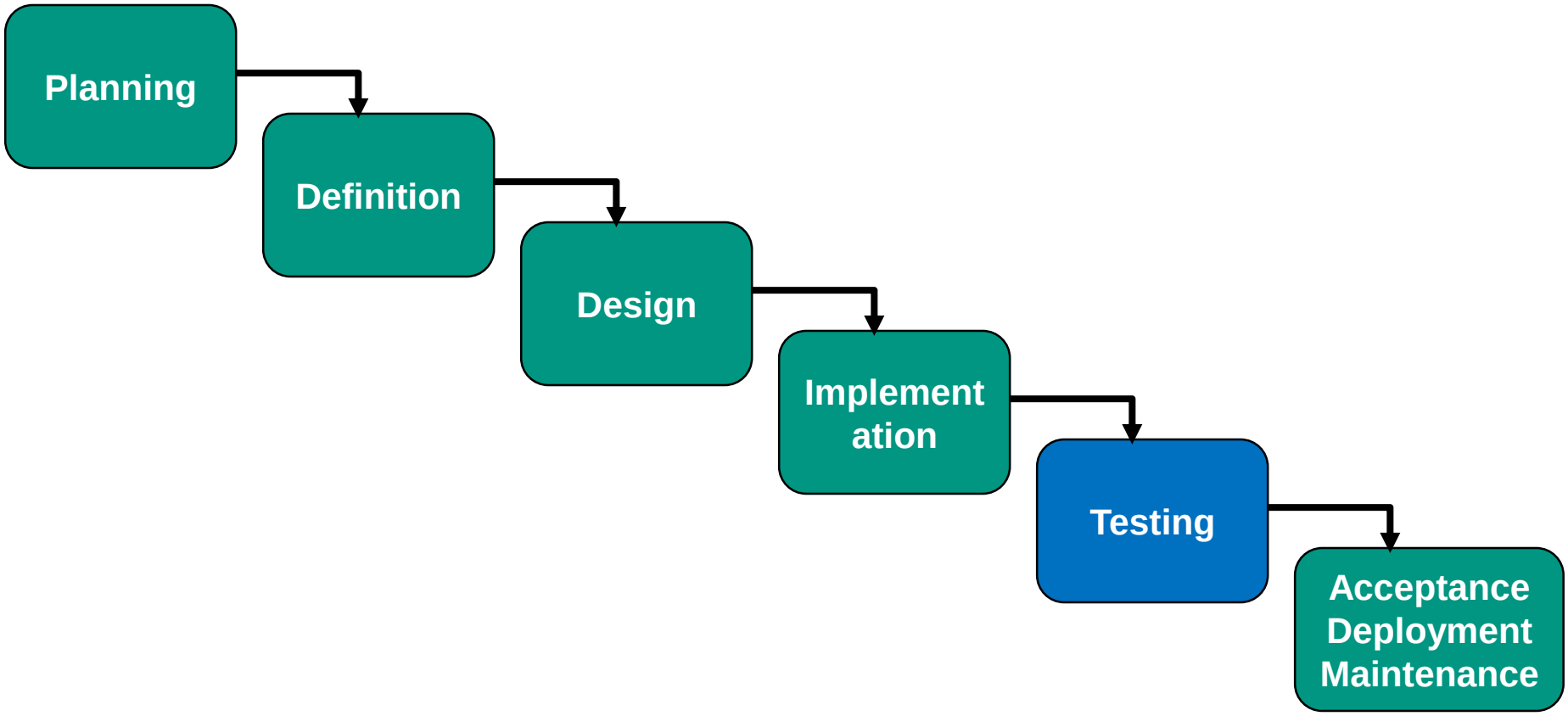
Quality Assurance Concepts

CMPS115 – Summer 2017

Waterfall of Activities



Reading

- This lecture corresponds to Chapter 11 of

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

Why Testing?

- Software artefacts always contain defects
- The later a defect is found the more it costs to eliminate it.
- Goal: Find defects as soon as possible..

Goal of testing is defect detection

- A test is successful if it finds a problem!

- How much testing?
  - Complete testing not possible:
    except for trivial programs,
    number of possible input combinations is astronomical (or infinite)
- Formal correctness proofs
  (consistency of program with specification)
  - not feasible for large systems
  - how do we know the specification is “correct”?
- Key Question: How do we know that we can stop testing?
  (Test completeness criteria (in a practical sense))

Important distinction

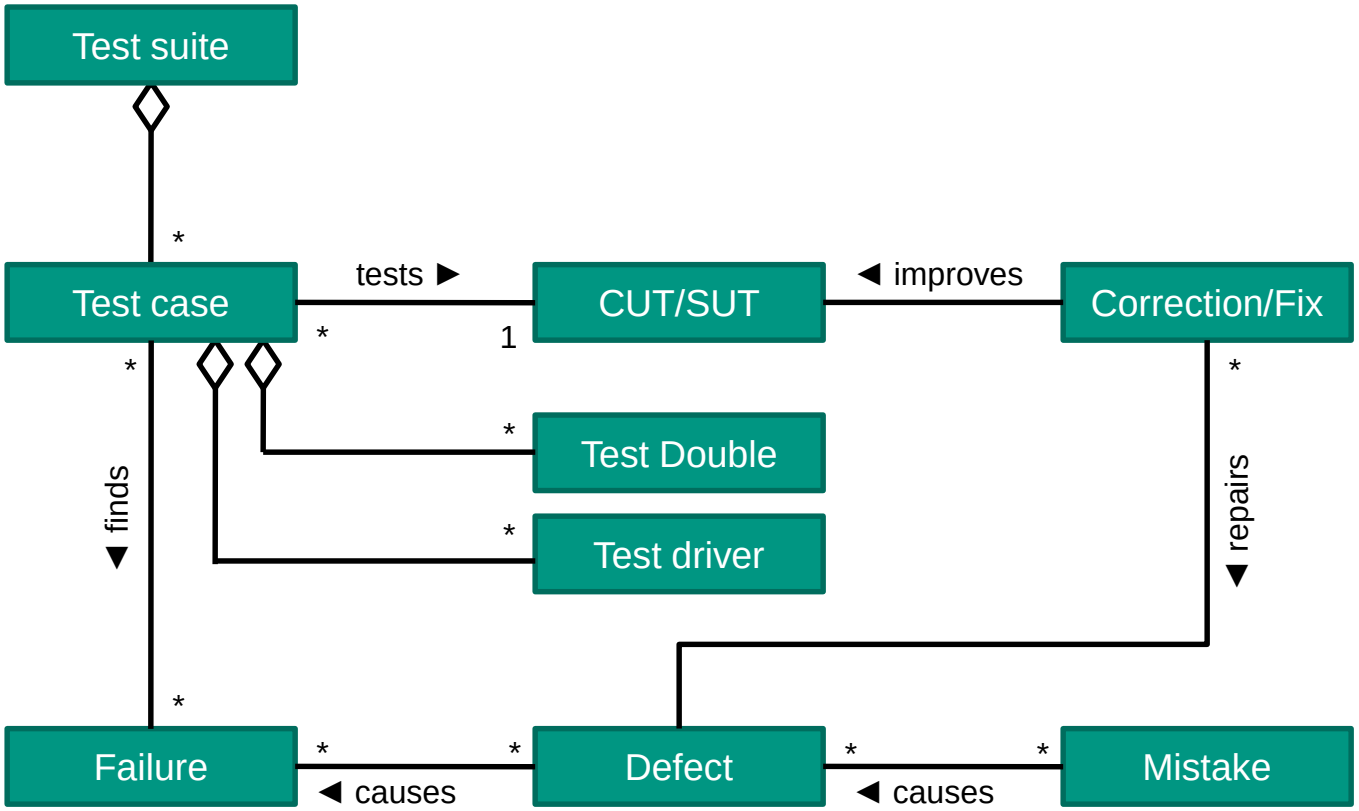
- Testing
  - discovers defects/problems
  - proves correctness for sample of inputs
- (Formal) Verification
  - proves correctness for all inputs
    (of one artefact with respect to another artefact)
    - specification with respect to requirements (if formalized)
    - implementation with respect to specification
- Analysis
  - determine properties of artefact/system component
  - e.g.
    - code: every part of the program will be executed for some input
    - specification: withdrawing money from an account will not result in a negative balance

Types of problems

- Failure:
  software fails to behave as specified
  - failure is an event during software execution
- Fault (defect, “bug”):
  a problem in the structure, form, or representation of the software product that can cause a failure
  - fault is a state
- Error (Mistake)
  - human action or misjudgment that causes a defect

- Errors ➔ Faults ➔ Failures

Relationship between problem types and testing



Testing concepts

- (Software) Test: executes single component or subsystem under known conditions (environment, inputs) and checks the behavior and results (reactions and outputs)
- Test object,
  Component under test (CUT),
  System under test (SUT):
  component or subsystem being tested
- Test case:
  input and configuration data for the execution of the test;
  self-validation: compare expected to actual results
- Test driver, Testing framework, Testing harness:
  supplies test data to test objects and initiates execution

Test Doubles

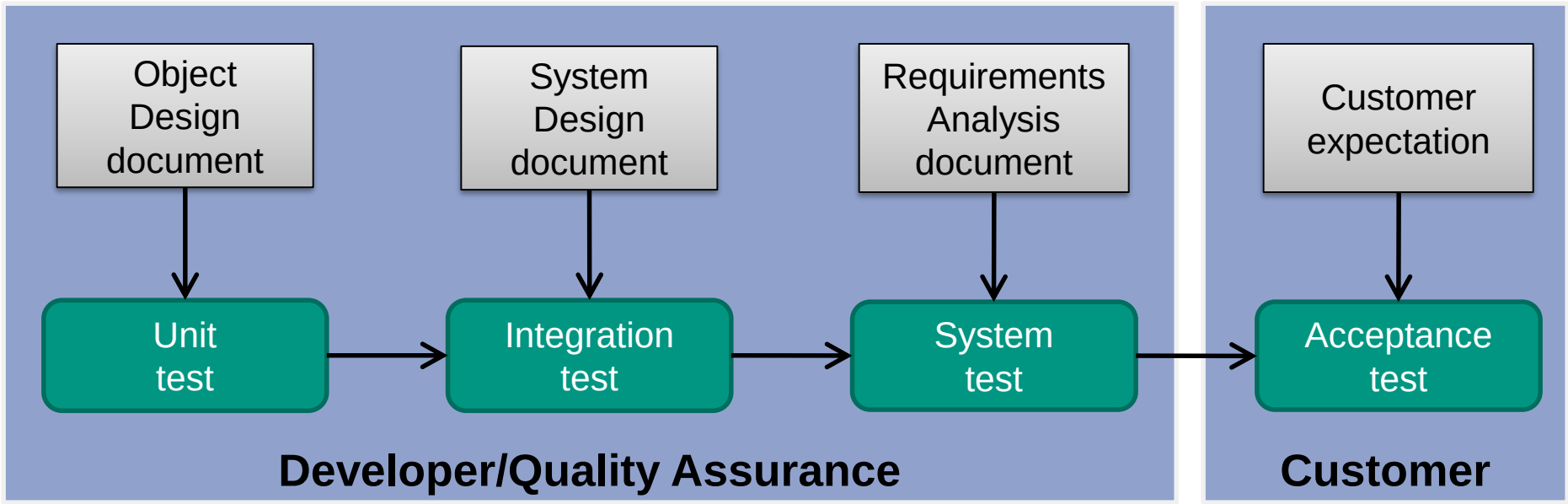
- Test Doubles (Meszaros)
  test supporting components that stand in for components that component under test (CUT) depends on (DoC: depended-on component)
  - Help to isolate CUT from dependence on other components
    - Perhaps DoCs are not yet existent
  - Speed up tests
    - E.g. replace cloud database by in memory database

- Terminology for flavors of test doubles not uniform
  - Test stub, dummy object, mock object, fake object
  - See lecture slides on Unit Test Patterns

Error types by phase-specific artefacts

- Requirements error (in Requirements definition)
  - incomplete capture of functional and non-functional requirements
  - inconsistent requirements
  - infeasible requirements
  - Incorrect requirements
- Design errors (in System specification)
  - incomplete or incorrect rendering of requirements
    - E.g. error in test case definition
  - inconsistent specification or design (internal inconsistency)
  - inconsistencies between requirements and specification
- Implementation error (Defect in program/source code)
  - incorrect realization of specification

Test Phases



Unit tests check function of single module (class, object) by observing processing of test data. Done by developer

The **integration test** checks the interaction between already unit tested components

The **system test** checks is the final test in the development environment or in separate test environment by the developing organization.

The **acceptance test** is the final test in the customer‘s production environment directed by customer; basis for accepting or rejecting product delivery.

Test-driven development; Continuous integration

Classification of Quality Assurance Approaches

- Dynamic Approaches: Testing
  requires software execution
  - structure-based tests (white/glass box testing)
    - control flow oriented
    - data flow oriented
  - functional tests (black box testing)
  - performance testing (black box testing)

- Static Approaches: Checking, Reviewing
  requires software artefacts
  - manual methods:
    pair programming, inspections, reviews, walkthroughs
  - analysis tools: static program analysis

Dynamic vs Static QA Methods

- Dynamic methods ("Testing")
  - Execute compiled/interpreted program code with selected test data
  - Test data and execution environment approximate target environment (in which system will be deployed) to varying degree
  - Testing takes samples of system behavior
    - Exhaustive testing typically not possible
    - Testing can only prove the presence of faults, not their absence
- Static methods
  - Analyze system models or code without executing it
  - Human methods (reviews, inspections, walk-throughs) analyze system models or source code
  - (Computer-based) Analysis tools in addition can also analyze object code
    - Increasingly used for critical embedded code

Black Box vs. White/Glass Box Testing

- Focus of **Black Box** Testing:
  Did we build the right system? (Validation)
  - Tests designed to check user-observable behavior against requirements
  - Test coverage: relative to possible inputs and outputs
  - Ignores internal system structure (architecture etc.)

- Focus of **White/Glass/Clear Box** Testing:
  Did we build the system right? (Verification)
  - Tests designed to check system-internal behavior
  - Test coverage: relative to system components (at various levels)
    - e.g. statements, execution paths, conditions, classes, packages, etc.

Overview Matrix

Phase						
Acceptance test						
System test						
Integration test						
Component test						
Method	Control flow oriented	Data flow oriented	Functional tests	Performance tests	Manual V&V methods	Automated V&V tools

Component testing: Control Flow Oriented (CFO)

- Statement coverage
- Branch coverage
- Path coverage
- Condition coverage
  - various flavors

- Defined via Control Flow Graph (CFG)
  - see compiler construction:
    - Static analysis
    - code generation/optimization

Excursion: Control Flow Graphs

- Coverage criteria for White box testing defined via control flow graph
  - Definition of intermediate language
  - Definition of transformation or source code into intermediate language
  - Definition of control flow graph (CFG)
- Finally: definition of testing methods and coverage criteria

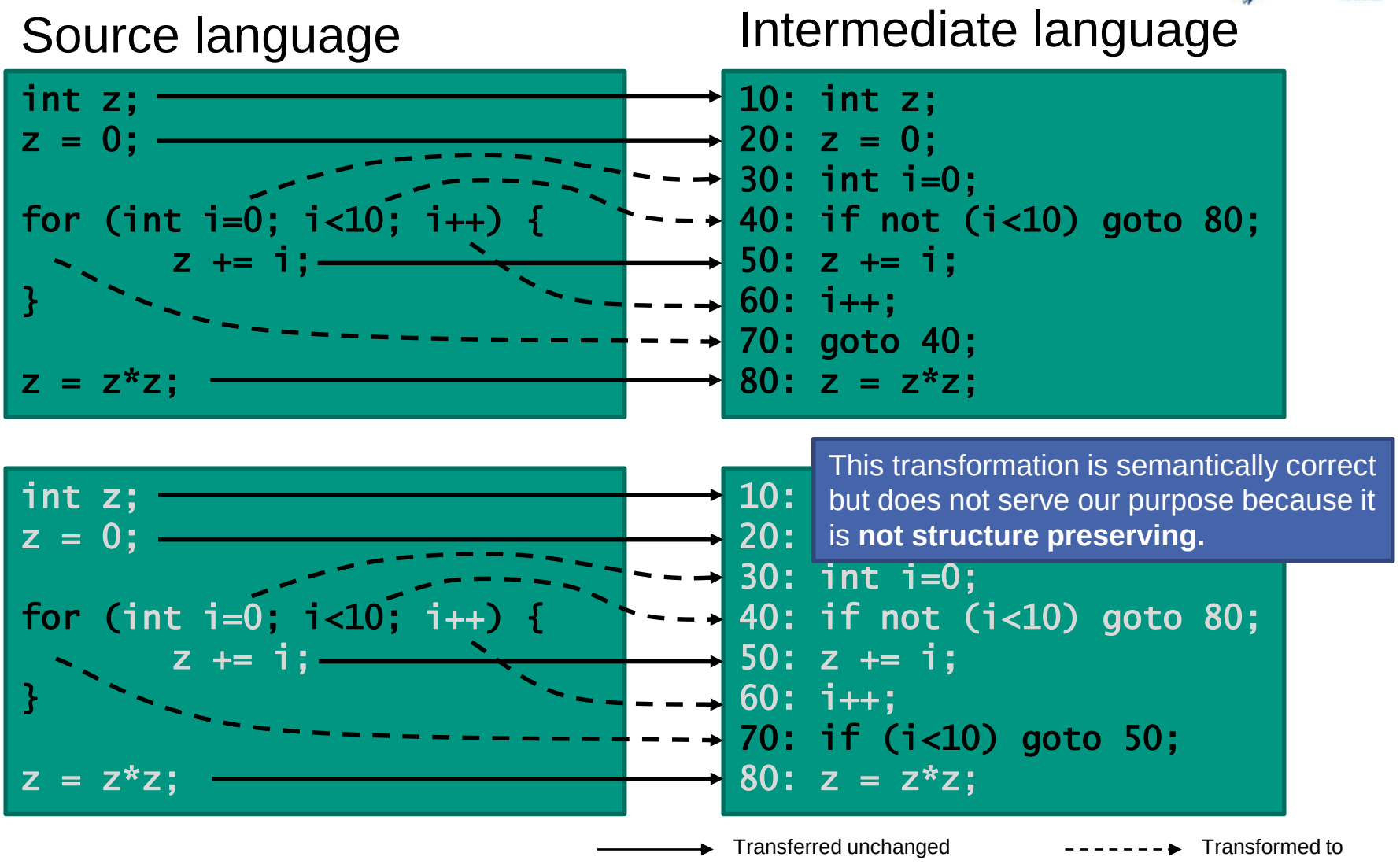
Definition: Intermediate Language

- A kind of high-level assembler code
- Divide source language constructs into
  - Non-control constructs
    - Assignment, method call, input/output
  - Control constructs
    - Conditionals, looping constructs
- Intermediate language
  - Non-control constructs as in source language
  - Control constructs
    - Conditional jumps
    - Unconditional jumps

Definition: Structure preserving transformation

- Structure preserving transformation
  - From source language (e.g. Java) into intermediate language
  - Control constructs (of the source language) translated into control constructs of the intermediate language
    - Without replicating or deleting code (e.g. no loop unrolling or optimizations)
  - Non-control constructs transferred unchanged
    - Leaving their execution order unchanged

Example transformation



Definition: Basic Block

- A Basic Block is a maximal sequence of straight-line code in the intermediate language such that
  - Control enters at the beginning of the block
  - Block contains no jumps except at the end

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 (CFG)

- A Control Flow Graph of a program P is a directed graph

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

where

- N: set of basic blocks in P
- $E \subseteq N \times N$: the set of directed edges (corresponding to jumps)
- n_{start}: the start basic block
- n_{stop}: the final basic block

Constructing the Control Flow Graph

Sample Program:

```
…
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();
}
…
```

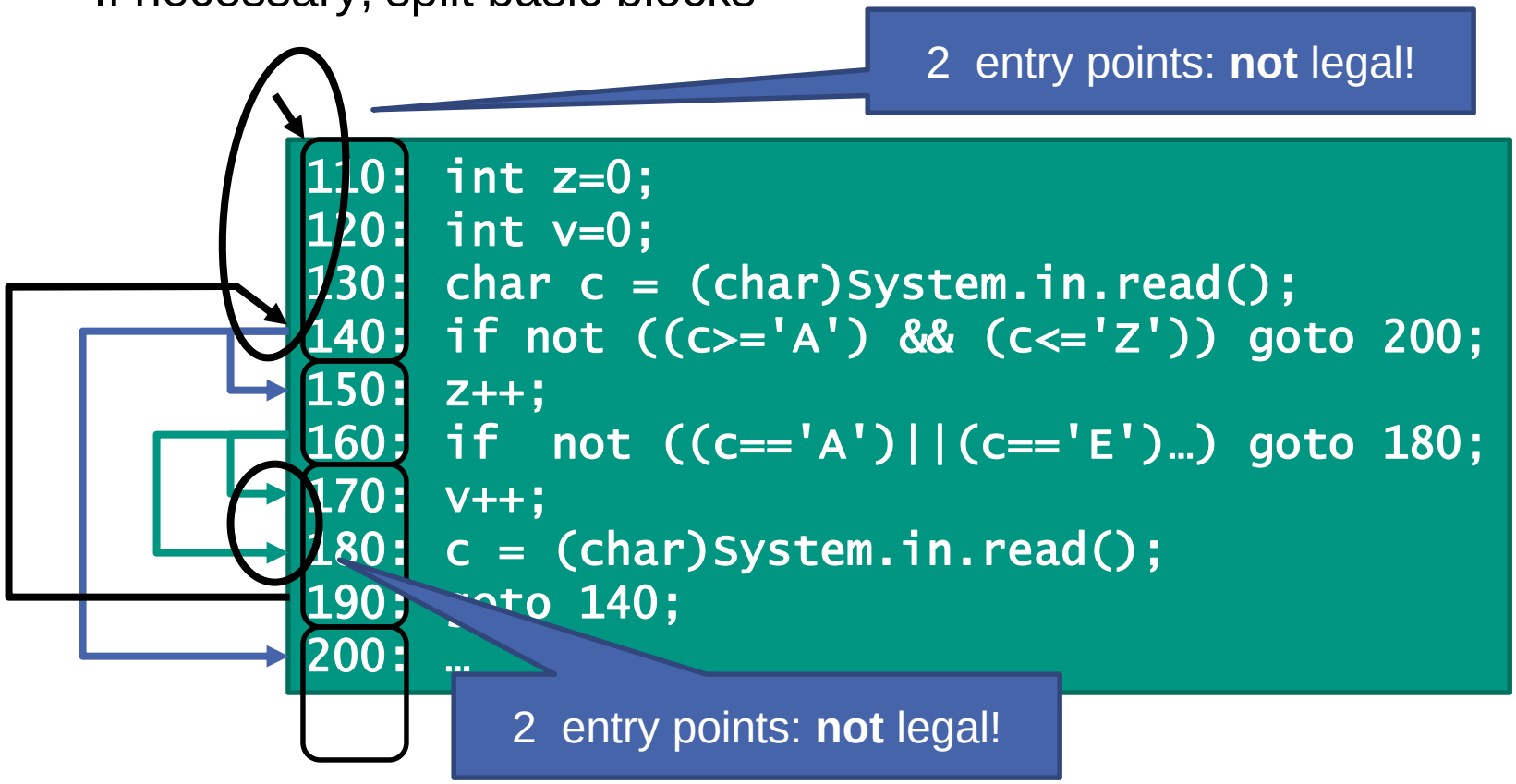
Constructing the Control Flow Graph (2)

- Step 1:
  Transform source into intermediate language
- Step 2:
  Combine statement sequences that end 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:  …
```

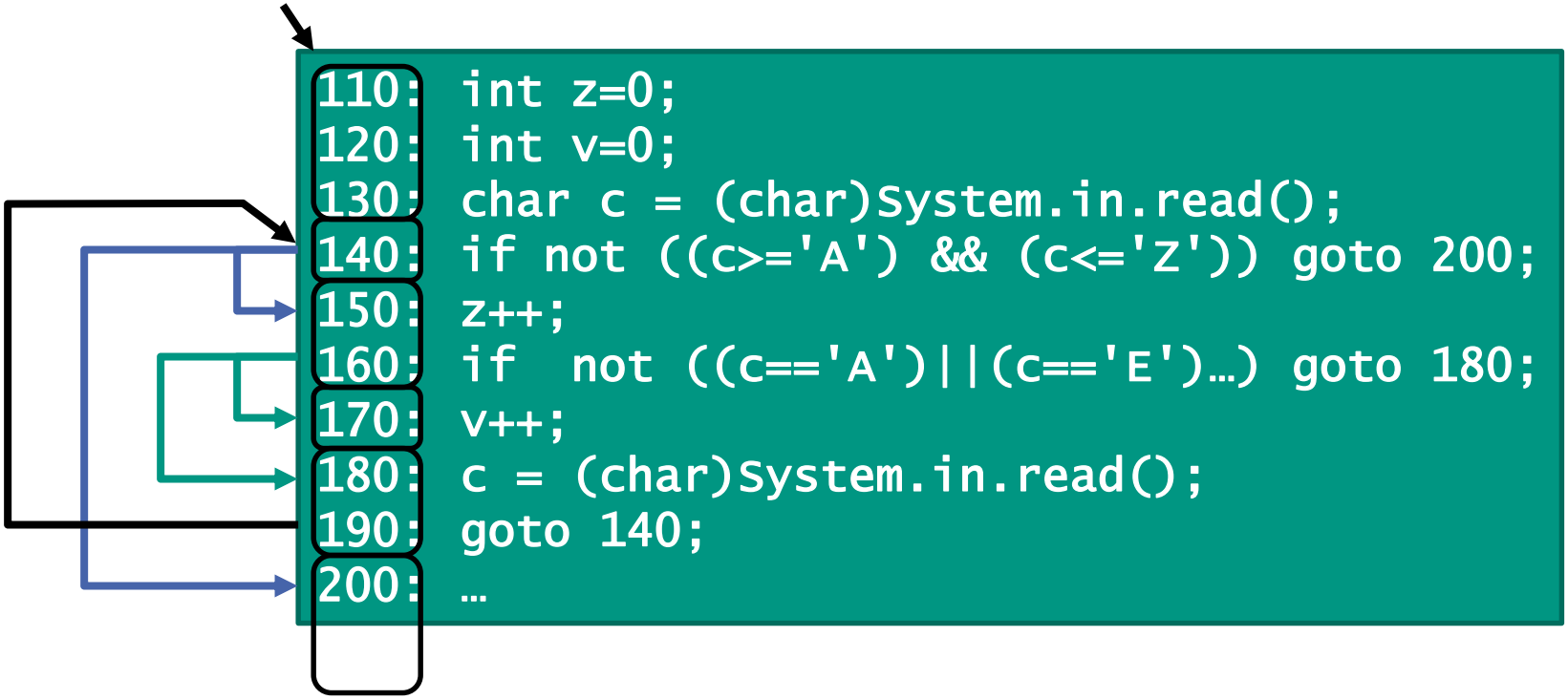
Constructing the Control Flow Graph (3)

- Step 3:
  Check whether control enters only at the beginning
- Step 4:
  If necessary, split basic blocks

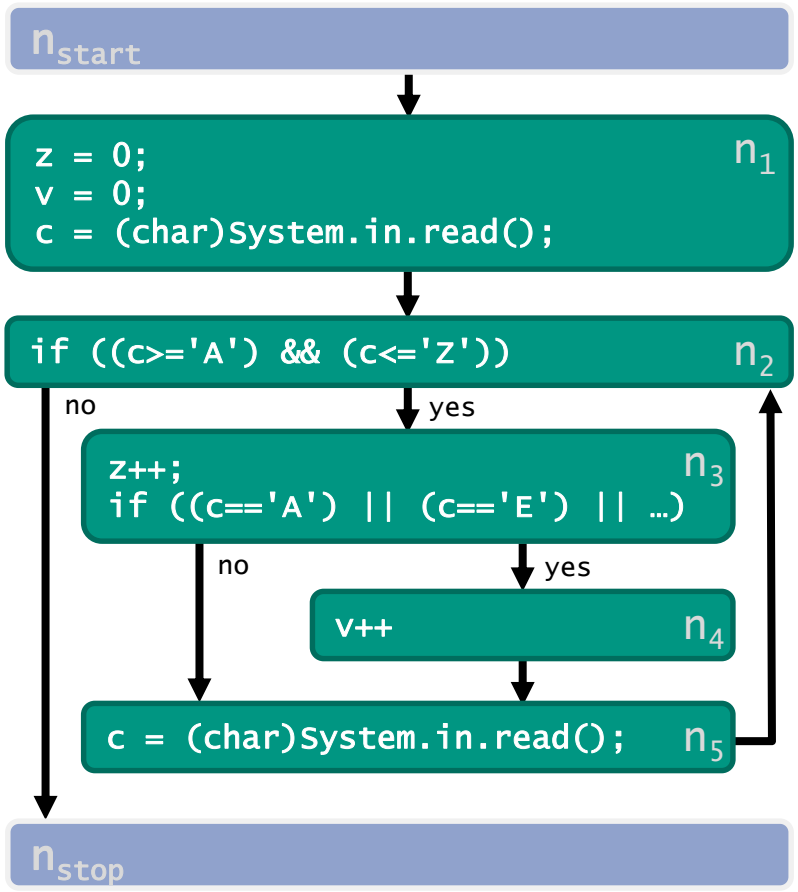


Constructing the Control Flow Graph (4)

- Step 3:
  Check whether control enters only at the beginning
- Step 4:
  If necessary, split basic blocks

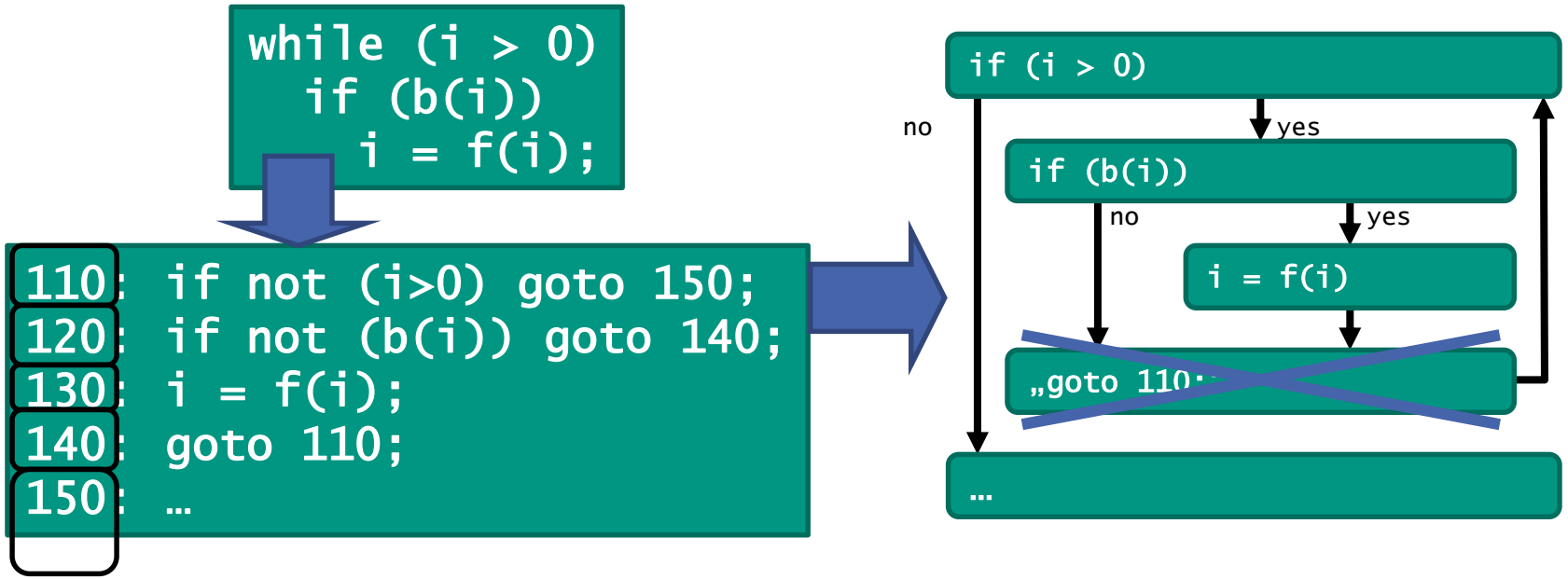


Sample Control Flow Graph



Simplification of CFG

- Basic blocks that consist only of an unconditional jump can be removed
  - Sometimes such blocks are the result of block splitting



Definitions: Branch, Complete Path

- An edge $e \in E$ in a CFG G is called a branch.
  - Branches are directed.

- A complete path starts at the starting node n_{start} and ends at the final node n_{stop}.

Definition: Statement Coverage

- Statement Coverage
  - A metric for test suites
  - A test strategy
- Metric (sometimes called C_0) for statement coverage

$$C_{\text{statement}} = \frac{number\ of\ statements\ executed\ by\ tests}{number\ of\ all\ statements\ in\ program\ P}$$

- Test Strategy *Statement Coverage*
  - Requires execution of all basic blocks in program P
  - Not a sufficient test criterion
    - Cannot discover missing program parts
    - May hint at unreachable program parts (dead code)
      - No test suite can be constructed that executes unreachable code
      - Dead code: sometimes faulty logic; sometimes unremoved junk

Definition: Branch Coverage

- Metric for branch coverage (sometimes called C_1):

$$C_{branch} = \frac{number\ of\ branches\ traversed}{total\ number\ of\ branches}$$

- Test strategy *Branch Coverage*
  - Requires traversal of all branches (edges) in a CFG for program P.
  - Helps discover unreachable branches
  - Does not consider combinations of branches (paths) or complex conditions
  - Loops not sufficiently tested
  - Missing branches are not testable and not discovered

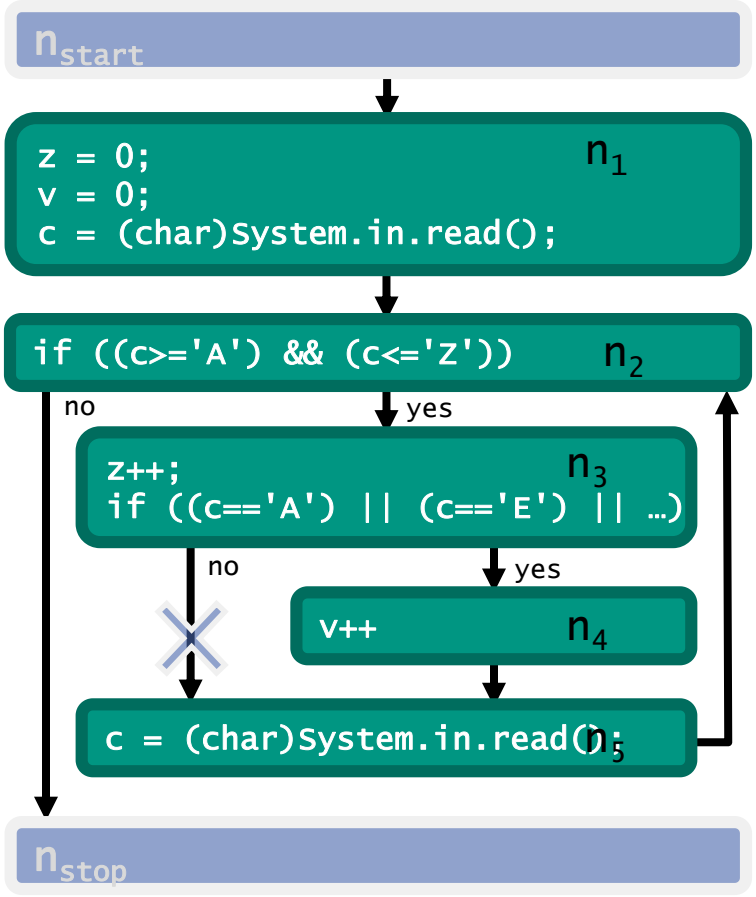
Statement vs. Branch Coverage

Statement Coverage: all nodes

Example sequence:

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

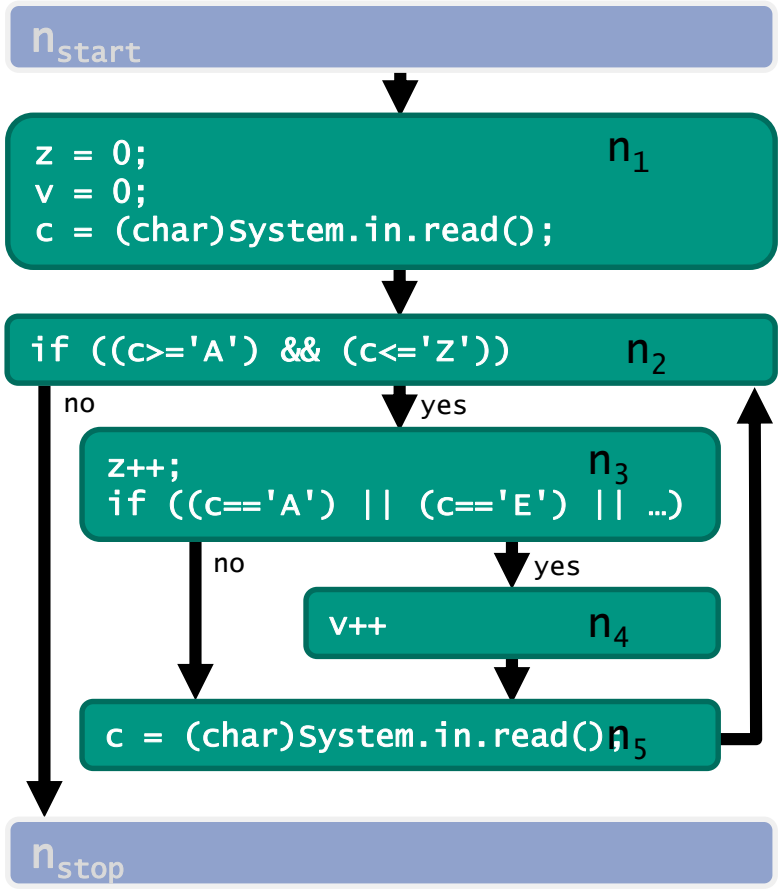
→ branch (n_3, n_5) not executed



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_{stop})



Definition: Path Coverage

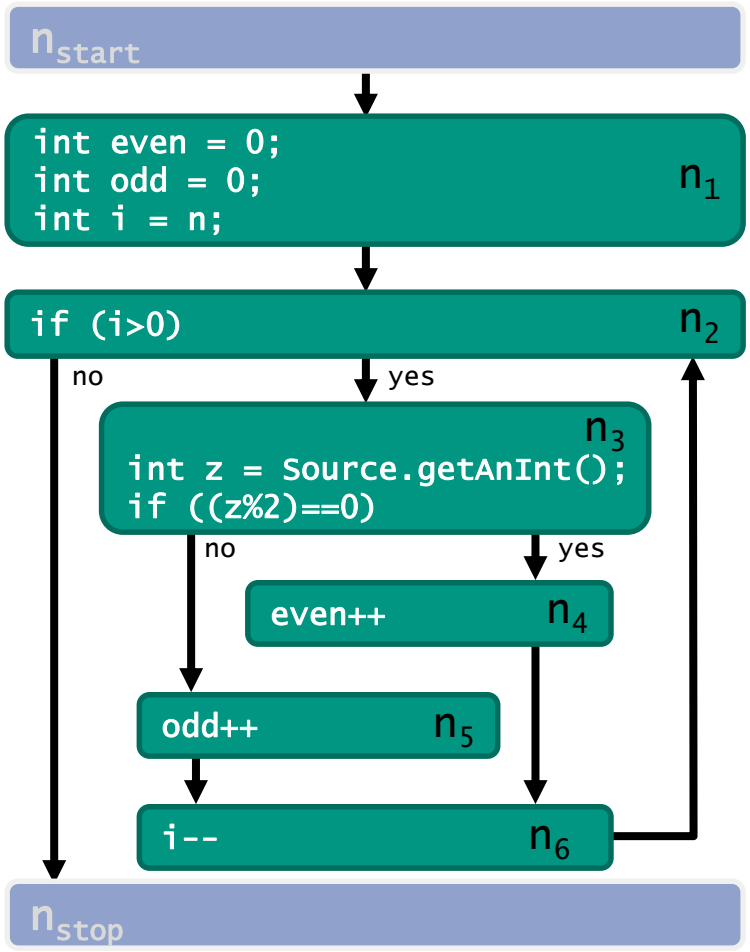
- Test strategy *Path Coverage*
  - Requires execution of all *complete paths* in program P.
  - Loops lead to dramatic increase in number of paths.
  - Some paths may not be executable
    - due to mutually exclusive (i.e. contradictory) conditions
  - Most powerful test strategy
  - “too powerful”: not practical

Growth of number of paths

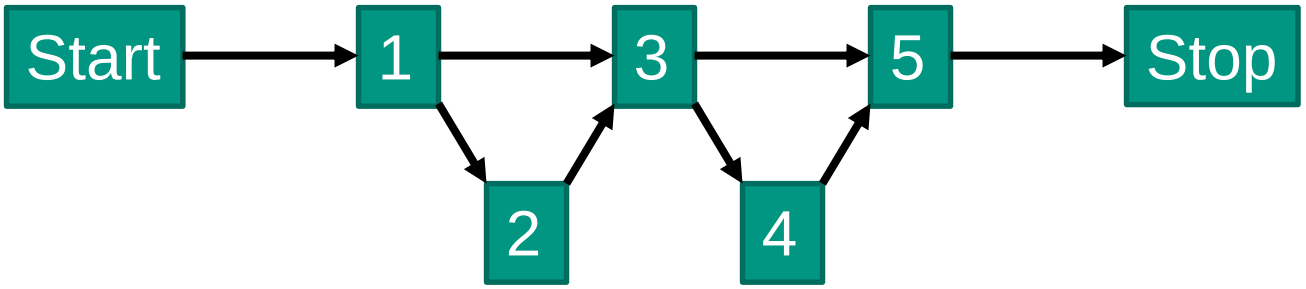
Sample Program

```
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	# Paths
0	I
1	II
2	IIII
…	…
k	2^k



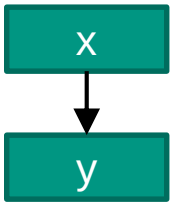
Example:
Contrast Statement, Branch, Path coverage



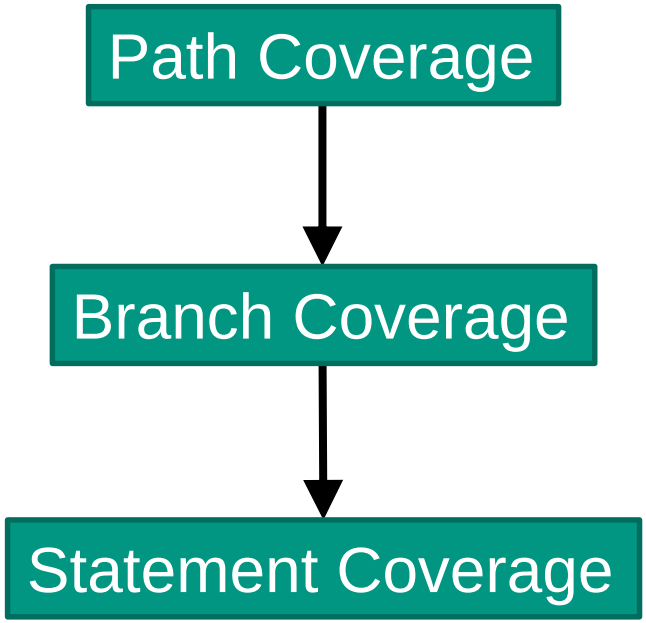
- Statement coverage
  A = { (Start, 1, 2, 3, 4, 5, Stop) }
- Branch coverage
  Z = A ∪ { (Start, 1, 3, 5, Stop) }
- Path coverage
  P = Z ∪ { (Start, 1, 3, 4, 5, Stop) }
  ∪ { (Start, 1, 2, 3, 5, Stop) }

Subsumption of test criteria

- Test criterion X subsumes test criterion Y if satisfaction of X implies satisfaction of Y



Test criterion X subsumes test criterion Y



Summary: CFO based Strategies

- Statement Coverage: weakest criterion
  - Each statement must be executed at least once if we are to discover any defects in it
- Branch Coverage:
  - Each branch (edge between basic blocks) must be exercised at least once to discover any defects
- Path Coverage: most resource intensive criterion
  - Even for small programs not practical
  - More practical version: Boundary Interior Test
    - Design test set that executes each loop body once
    - Design second set that executes each loop body twice
    - Ensure you cover all paths inside loop body;
      apply boundary interior test for nested loops

Overview Matrix

Phase						
Acceptance test						
System test						
Integration test						
Component test	✓					
	Control flow oriented	Data flow oriented	Functional tests	Performance tests	Manual V&V methods	Automated V&V tools
Method						

Component tests: Functional tests

- Goal: Test of specified functionality
  - derive test cases from specifications
    - test driven development:
      test cases are the specification
      - Cf. Meyer's critique of this idea in Agile! (Piazza)
  - Internal structure not considered – **Black Box testing**

  - Advantages:
    - test cases independent from implementations
      - can be generated before implementation
    - avoids test case selection bias
      - test case developer may be agnostic re implementation
  - Limitation:
    - cannot assess quality of code (structure)

Functional Tests -- Illustration

- Test cases contain:
  - Input data
  - Output data/response (expected)

- Problem:
  test the function $f(x) = \sqrt{x^3}$

  - Cannot test exhaustively for all input values

  - How to select test data to maximize chance of discovering defects?

Dealing with large test data space

- How can we limit functional testing to a reasonable number of test cases?
  - functional equivalence classes
    - legal/illegal inputs
  - boundary case analysis
    - e.g. less than zero, zero, greater than zero
  - random testing
  - testing of state automata
    - E.g. “Model checking”:
      - Verification (logic) based technology
      - Reduces infinite value spaces to (potentially still large) finite spaces by forming equivalence classes of values

Functional Equivalence Classes

- Assume: execution of method/program/module is virtually the same for values within a certain range
  - E.g. a function that correctly computes 42^2 also computes 40^2 correctly
  - Note:
    Test values need not be legal input values;
    we also need to test error handling!
- Approach:
  partition input domain into equivalence classes

Functional Equivalence Classes (2)

- Construction of Equivalence Classes (EC) (Idea)
  - Partition possible inputs into legal and illegal inputs
  - Further partition inputs based on other properties
    - And may therefore exercise different parts of the program
- Select representatives from equivalence classes
  - Select for each input equivalence class a representative
  - E.g. test multiplication of integers
    - First approach: Integers = Even U Odd
      - Possible test set: {(4, 6), (5, 6), (6, 5), (3, 5)}
    - Second approach: Integers = Negative U {0} U Positive
    - Third approach: combination of first and second approach

Example: Forming Equivalence Classes (1)

- Sample function: determine number of days in a month
  - Inputs: year, month: Integer

Equ. 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
$EC_{M,Feb}$	Months with 28 oder 29 days	2
$EC_{M,error}$	Invalid (month) inputs	Month < 1 oder Month > 12
$EC_{Y,\ LY}$	Leap years	1904, 2000, 2012, ...
$EC_{Y,\ nonLY}$	Non-leap years	1901, 1900, ...
$EC_{Y,error}$	Invalid (year) inputs	Year < 0

Example: Forming Equivalence Classes (2)

- Derive test cases (for valid inputs)

Equivalence Class	Input		Expected Output
	Month	Year	
$EC_{M,31}$ und $EC_{Y,non\text{-}LY}$	7	1900	31
$EC_{M,31}$ und $EC_{Y,LY}$	7	2000	31
$EC_{M,30}$ und $EC_{Y,\ non\text{-}LY}$	6	1900	30
$EC_{M,30}$ und $EC_{Y,LY}$	6	2000	30
$EC_{M,Feb}$ und $EC_{Y,\ non\text{-}LY}$	2	1900	28
$EC_{M,Feb}$ und $EC_{Y,LY}$	2	2000	29

Example: Forming Equivalence Classes (3)

- Derive test cases (for invalid inputs)

Equivalence Class Combinations	Input		Expected Output
	Month	Year	
$EC_{M,\ error}$ and $EC_{Y,non\text{-}LY}$	-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 $EC_{Y,\ error}$	6	-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: off by one is still off (!)
  - Test cases that cover the boundaries of equivalence classes and their immediate neighborhood are particularly effective

Therefore:

- Pick test values that are boundary values of equivalence classes or immediate neighbors
  - Example: month
    use 0 and 1, 12 and 13
- Special candidates:
  - Boundaries and special values: 0, 1, Max_Int, NaN, …

Random Testing

- Random Testing: pick test values at random
  - Observation:
    Testers tend to pick values that are representative of “normal” system behavior (and were therefore the basis for the system design)
- Advantage: all values equally likely as test values
  - No tester bias in test case selection
- Complements other methods
  - E.g. equivalence classes
- Suitable for automated testing

Testbed construction

- Objects collaborate with other objects in an application
- Collaboration results in dependencies
- How can we test individual objects/properties in isolation
  - Exactly what is to be tested?
  - What potential problems are the tests to address?
- How can we construct test that get around the dependencies?

Test Helpers

- Use Test Doubles as approximations when there are dependencies on classes that
  - are not implemented yet
  - Inefficient/expensive to use
  - Difficult to observe
- Doubles as stand-ins for still missing implementations
  - Top-down or Outside-In development
  - Replace stubs and dummies by actual implementations

- See Lecture Notes on Unit Test Patterns

Overview Matrix

Phase	Control flow oriented	Data flow oriented	Functional tests	Performance tests	Manual V&V methods	Automated V&V tools
Acceptance test						
System test						
Integration test						
Component test	✓		✓			
Method						

Performance Tests: Load tests

- Tested system/component for reliability and conformance with specification in at the boundary of valid range
  - Cope with required number of users?
  - System response within guaranteed response times?
  - Can load be managed for indefinite amount of time?
  - What is the load on expected bottlenecks?
  - Are there unexpected bottle necks?

Performance Tests: Stress tests

- Test system behavior when defined limits are exceeded
  - What is the performance when system is overloaded?
    - Thrashing
    - Superlinear increase in response time
    - System grinds to a halt
  - Does the system return to normal behavior when no longer overloaded?
    - How long does that take?

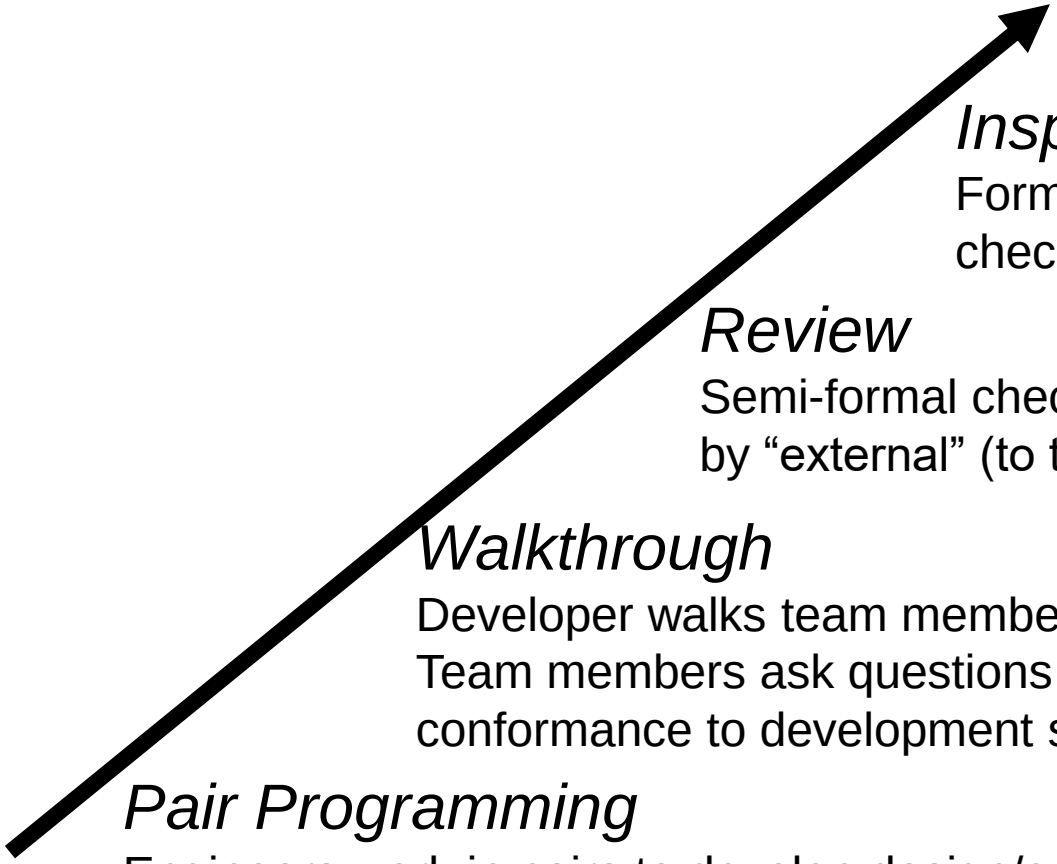
Overview Matrix

Phase						
Acceptance test						
System test						
Integration test						
Component test	✓		✓	✓		
Method	Control flow oriented	Data flow oriented	Functional tests	Performance tests	Manual V&V methods	Automated V&V tools

Manual V&V Methods

- Only way to check semantics
- Expensive (up to 20% of development cost)
- Time for reviews etc. must be planned for
- Psychological pressure when individual work is evaluated by group
- Social conflict potential
  - Work product, not person is under review
  - Need to review work of all team members on regular basis
  - Do it without managers/customers present

Distinctions between Review forms



Inspection

Formal (highly structured) audit using checklisits and reading techniques

Review

Semi-formal check of written document by “external” (to team) expert

Walkthrough

Developer walks team members through a design/section of code. Team members ask questions and comment on style, defects, conformance to development standards, etc.(ANSI/IEEE 729-1983)

Pair Programming

Engineers work in pairs to develop design/code
Driver/Navigator roles (traded off during session)
Driver needs to think out loud

Software Inspections (Overview)

- Several inspectors (2 to 8) inspect independently the same artifact (software document)
- Defects found are recorded;
- Focus is on identifying problems; problem solving comes later
- Structured process
  - Roles and forms
  - Check lists and perspectives

} Adapt to type of artifact

Definition: Inspection

- Inspection is a formal quality assurance technique
  (formal in the bureaucratic sense not the logico-mathematical sense)
  - inspected artifacts include requirements definition, design documents (models), and source code
  - Inspector(s) are different from author(s) of inspected documents
  - Purpose:
    Find errors, violations of standards, and other problems.

(based on ANSI/IEEE Standard 729-1983)

Pros and cons of inspections

- Pros
  - Applicable to all software artifacts
    - Requirements, specifications, designs, code, test cases, user documentation, ….
    - Can be done early in process and at any time during software process
    - Very effective in industrial practice

- Cons
  - Expensive
  - Elaborate organization
  - Uses time of several (usually senior) developers
  - “static” (as opposed to “dynamic” testing)

IBM
Hewlett Packard
Motorola
Siemens
NASA

Cost/Benefit – Some numbers

- More than 50% of defects discovered are found using inspections
  - Up to 90%
  - Typically 50%-75%
  - Can use Inspections to sample defect rate
- Relation of error fixing costs based on inspection vs. testing
  - Between 1:10 and 1:30
- Return on investment (ROI)
  - Often put at substantially over 500%
- Key paper:
  Fagan, M., *Design and Code Inspections to Reduce Errors in Program Development*, IBM Systems Journal 15, 3 (1976): 182-211
  - Note time of publication
  - Human/computer balance has substantially shifted since then

Why are inspections so effective?

- Basic principles:
  - Two pairs of eyes see more than one.
  - “Outsiders” see things the way they are, not the way they were intended
- Inspectors
  - Lack familiarity; take a fresh look
  - Not “vested” software artifacts
  - Often are very experienced; have seen many similar documents
- Joint discussion of identified issues
  - Is it a defect?
  - Is it a weakness? (Restricts generality, extensibility, robustness, …)
  - Does it “only” fall short of best practice?
  - Consensus based insight, judgment, and feedback

Inspection Phases

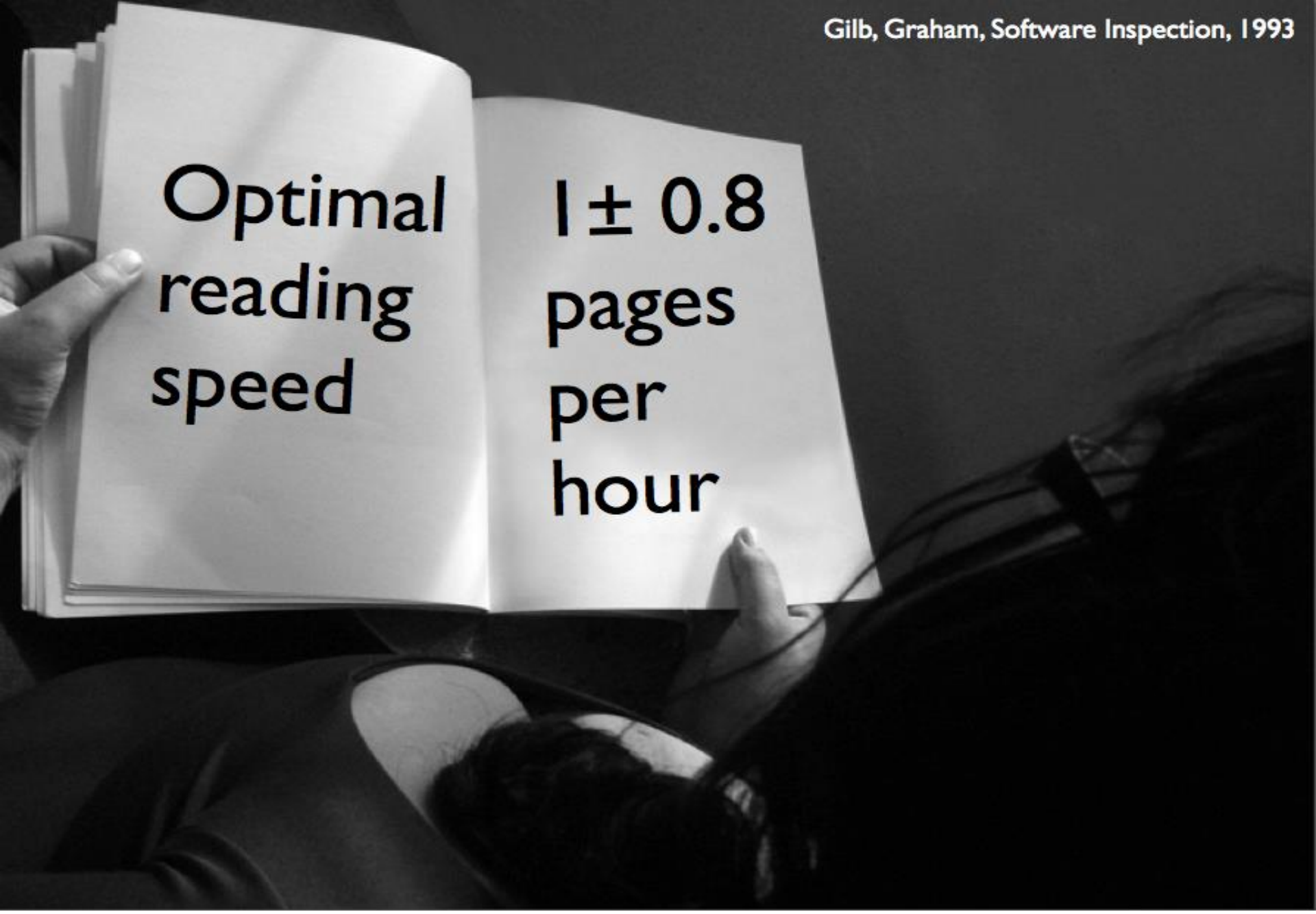
1. Preparation
2. Individual review
3. Group meeting
4. Post processing
5. Process improvement

Inspection: Preparation

- Determine participants and their roles
- Prepare documents and forms
- Determine reading techniques
- Plan schedule
  - Optional: initial meeting (for introduction)
  - If appropriate, partitioning of artifact
    - Group meetings should not exceed 2 hours
    - Focus on important points; ensure active participation

Inspection: Individual review

- Each inspector reviews artifact independently
  - About 1 page per hour per inspector
  - Results from individual reviews must be handled in 1 group meeting
    - i.e. each inspectors handles about 1-4 pages at most
- Use agreed upon reading technique
- Record in writing all issues and exact points in document
- Issues:
  possible defects, suggestions for improvements, questions


Gilb, Graham, Software Inspection, 1993
Optimal reading speed
1 ± 0.8 pages per hour

Inspection: Group Meeting

- Collect individually discovered issues
- Discuss each issue
- Clarify questions
- Record issues detected during discussion
- Collect suggestions for improvements

Alternative opinions: **Don‘t discuss!**

Inspection: Post processing

- List with issues given to document editor
- Editor identifies actual defects and classifies them
- Editor prompts changes to documents
- All issues are addressed
- Changes are checked
- Estimate of remaining defects
  - # of discovered defects ≈ # of undiscovered defects
  - About 1/6 of corrections contain errors/cause new defects
- Document is released when # of estimated defects < N

Inspection: Process Improvement

Process improvement step only taken in some cases

- Improve checklists and perspectives (cf. below)
- Define standards for documents
- Improve defect classification scheme
- Improve forms
- Improve planning and execution

Inspection: Roles

- Inspection leader: directs all phases of inspection
- Moderator: moderates the group meeting
- Inspectors: inspect document(s)
- Editor: classifies and initiates correction of defects (often identical with the author)
- Author: person who produced the document under review
- Secretary: records issues during group meeting

Forms for recording issues (typical aspects)

- Place in document (page, line)
- Description
- Priority/Importance: major/minor
- Type of issue: defect, suggestion, question
- Type of defect
- Fixed: yes/no
- Fix checked: yes/no

Defect classification

- Importance: major/minor
- Type of issue: defect, suggestion, question

- Defect type (NASA orthogonal defect classification)
  - A ambiguous information
  - E extraneous information
  - II inconsistent information
  - IF incorrect fact
  - MI missing information
  - MD miscellaneous defect

Reading techniques – Overview

- Ad hoc: left to reviewer
- Checklists
  - List of questions (about the document under review)
  - Intent is facilitate detection of issues
  - Each question addresses one quality aspect
  - Questions must be checked off
  - Questions depend of type of document, programming technology used (languages, libraries, …)
- Perspectives and Scenarios
  - Instructions on how to check the document
  - Set of questions
  - Perspective addresses quality aspect or type of defect
  - Generally more effective than checklists

Reading techniques -- Checklists

- Guidelines for checklists
  - List at most 1-2 pages, structured
  - Precise questions
  - Questions should point to methods for ensuring high quality
  - Avoid questions that can be checked automatically (by tool)
  - Keep questions up-to-date with technical state of the art

Reading techniques – Checklists: Example (1)

- A possible checklist for Java could be structured into these classes of defects:
  - Declarations: variables, attributes, constants
  - Method definitions
  - Class definitions
  - Computations
  - Comparisons
  - Control flow
  - Input/output
  - Module interface
  - Comments, documentation
  - Modularity
  - Memory use
  - Execution time

Reading techniques – Checklists: Example (2)

- Declarations
  - Are variable, attribute, constant names suggestive of semantics?
  - Do identifiers follow coding standards?
  - Are variables and attributes with similar names likely to be confused?
  - Do all variables and attributes have the correct type?
  - Are all variables and attributes initialized?
  - ….
- Method definitions
  - Are method names suggestive of semantics?
  - Do method names conform with coding standards?
  - Do method return the correct value?
  - Are methods appropriately qualified (static, public, …)
  - …

Reading techniques – Perspectives/Scenarios

- Instead of checklist, adopt a certain perspective and check document for aspects particular to perspective
- Typical perspectives
  - Maintenance
  - Code analysis
  - Testing
  - Design
  - Quality assurance
- Perspective based reading techniques provide
  - Explanation of perspective and its purpose
  - Instructions for reviewing document
  - List of questions about the document

Reading techniques – Perspectives: Example (1)

Example for Maintenance Perspective

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)

Example for Code Analysis Perspective

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.

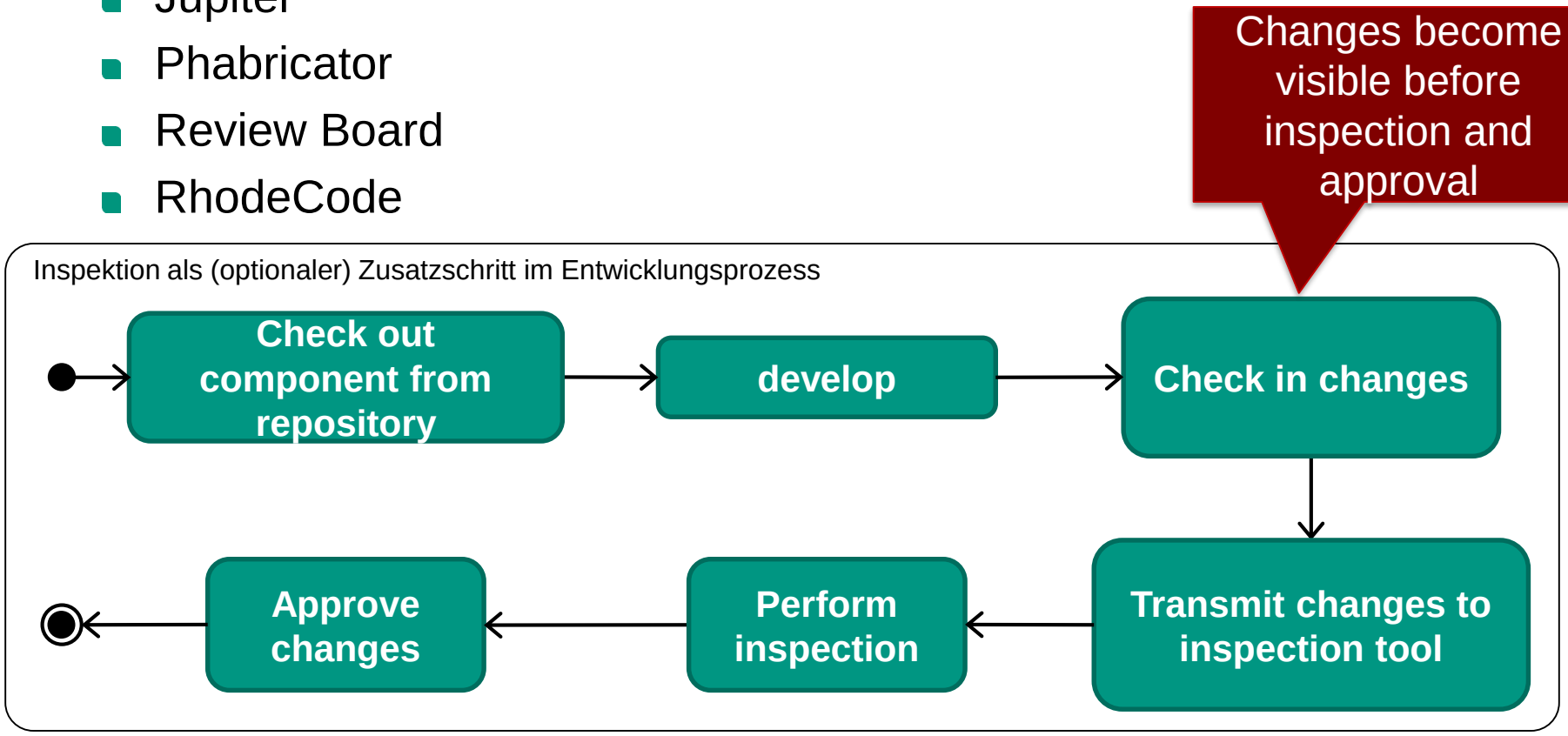
In doing so, take the code document and determine the functions that are implemented in this code module.
Determine the dependencies among these functions and document them in the form of a call graph.
Starting with the functions at the leaves of the call graph, determine the implemented operation of 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?

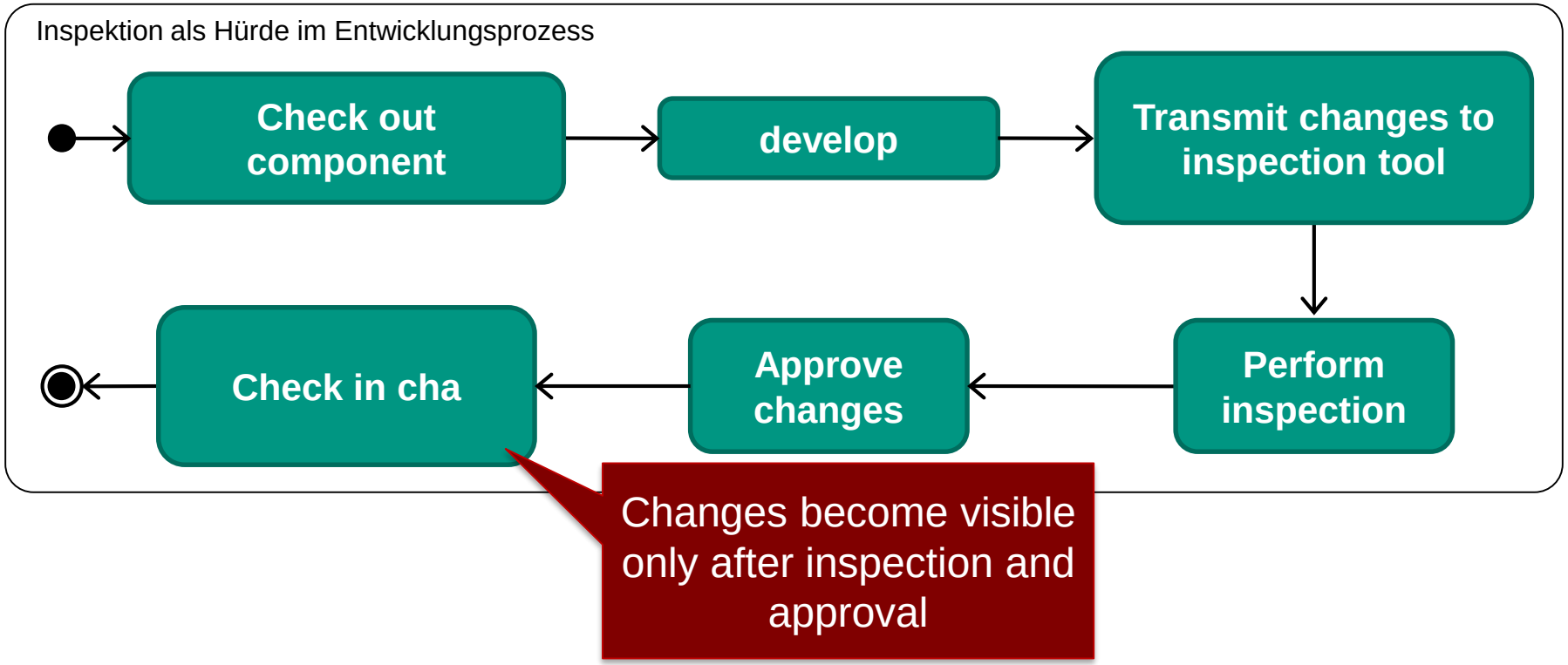
Inspection Support Tools (1)

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

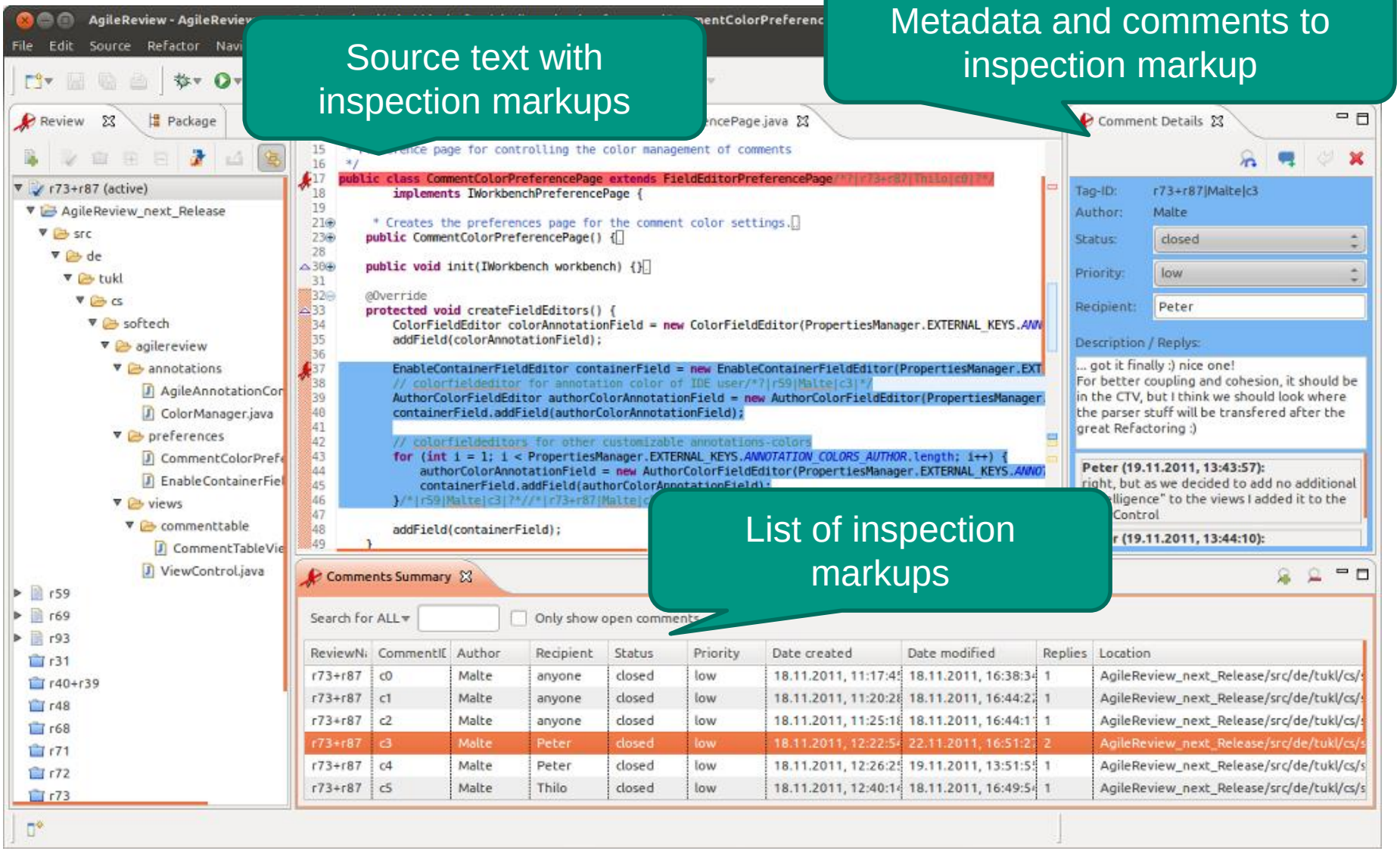


Inspection Support Tools (2)

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



Inspection Support Tools: Example Agile Review



Inspection support tools: Example Agile Review

- Eclipse-Plugin
- Each inspection is project-specific; consists of a set of source text points and associated comments and answers
- Inspections are version-controlled just like the inspected project
- Like the project, an inspection follows the time line of the version control system.

Inspection Support Tools: Example Gerrit

- Gerrit acts as buffer between developer and Git;
  - Delays check-ins until approval
- Compact representation of check-in (time stamp, developer, message, source code fragments and diffs), inspectors, and comments
- Check-in subject to vote, comments, or changes by other developers
- Optional: vote from integration server
- Gerrit checks in approved change into “official” version repository

Overview Matrix

Phase						
Acceptance test						
System test						
Integration test						
Component test	✓		✓	✓	✓	
Method	Control flow oriented	Data flow oriented	Functional tests	Performance tests	Manual V&V methods	Automated V&V tools

Static Analysis

- Static analysis
  - Typically done during compilation of source code
  - Many applications and plug-ins for development environments

Analysis tools

- Warnings and errors
  - Shown in IDE
    - Uninitialized variables
    - Dead code (not reachable by control flow)
    - Unnecessary code
- Style checkers
  - Indentation
  - JavaDoc comments
  - Parameter declarations
- Defect patterns
  - Analysis looks for certain patterns that indicate possible defects
  - Cf. Section 5.2 in Bruegge et al.

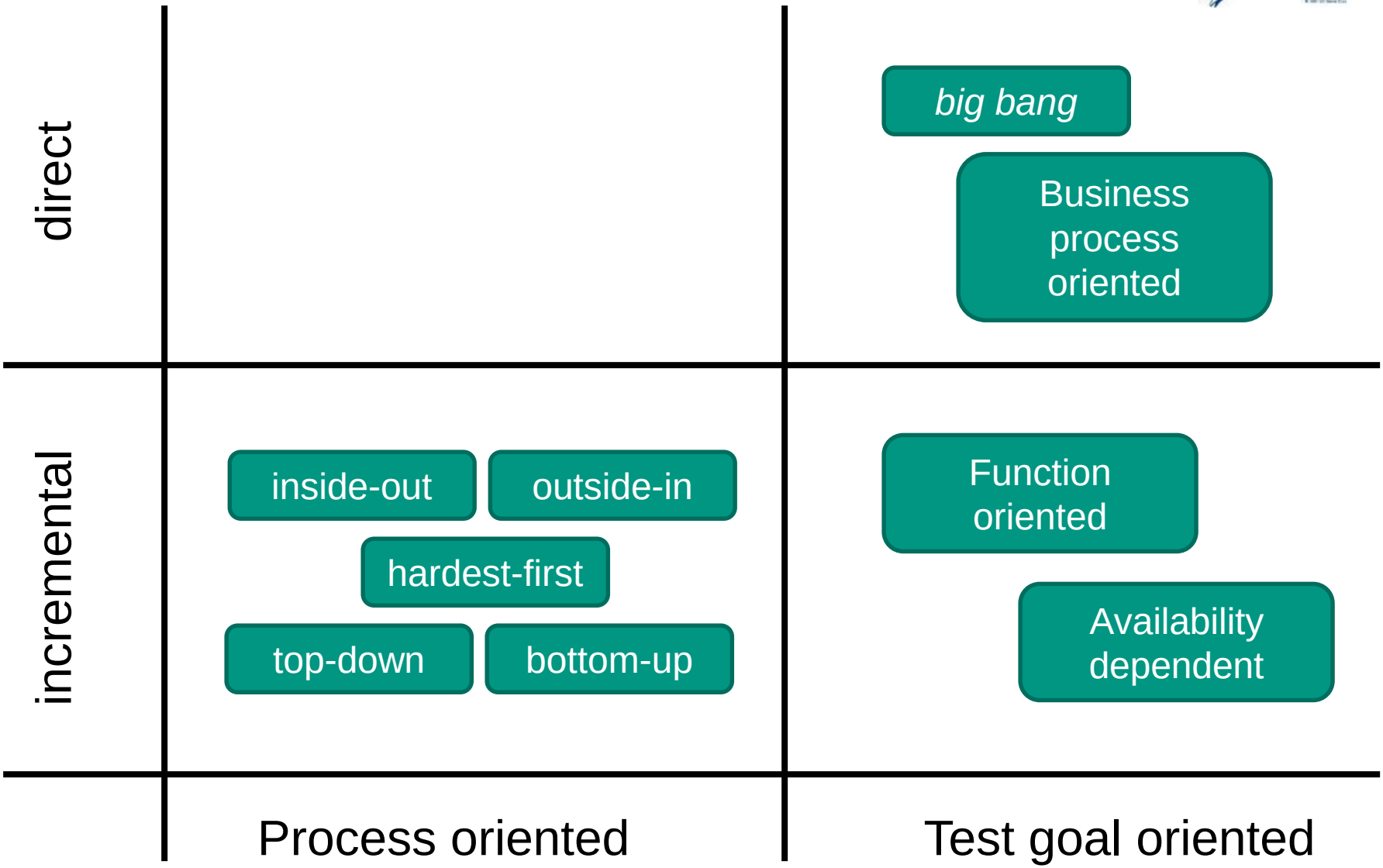
Overview Matrix

Phase						
Acceptance test						
System test						
Integration test						
Component test	✔		✔	✔	✔	✔
Method	Control flow oriented	Data flow oriented	Functional tests	Performance tests	Manual V&V methods	Automated V&V tools

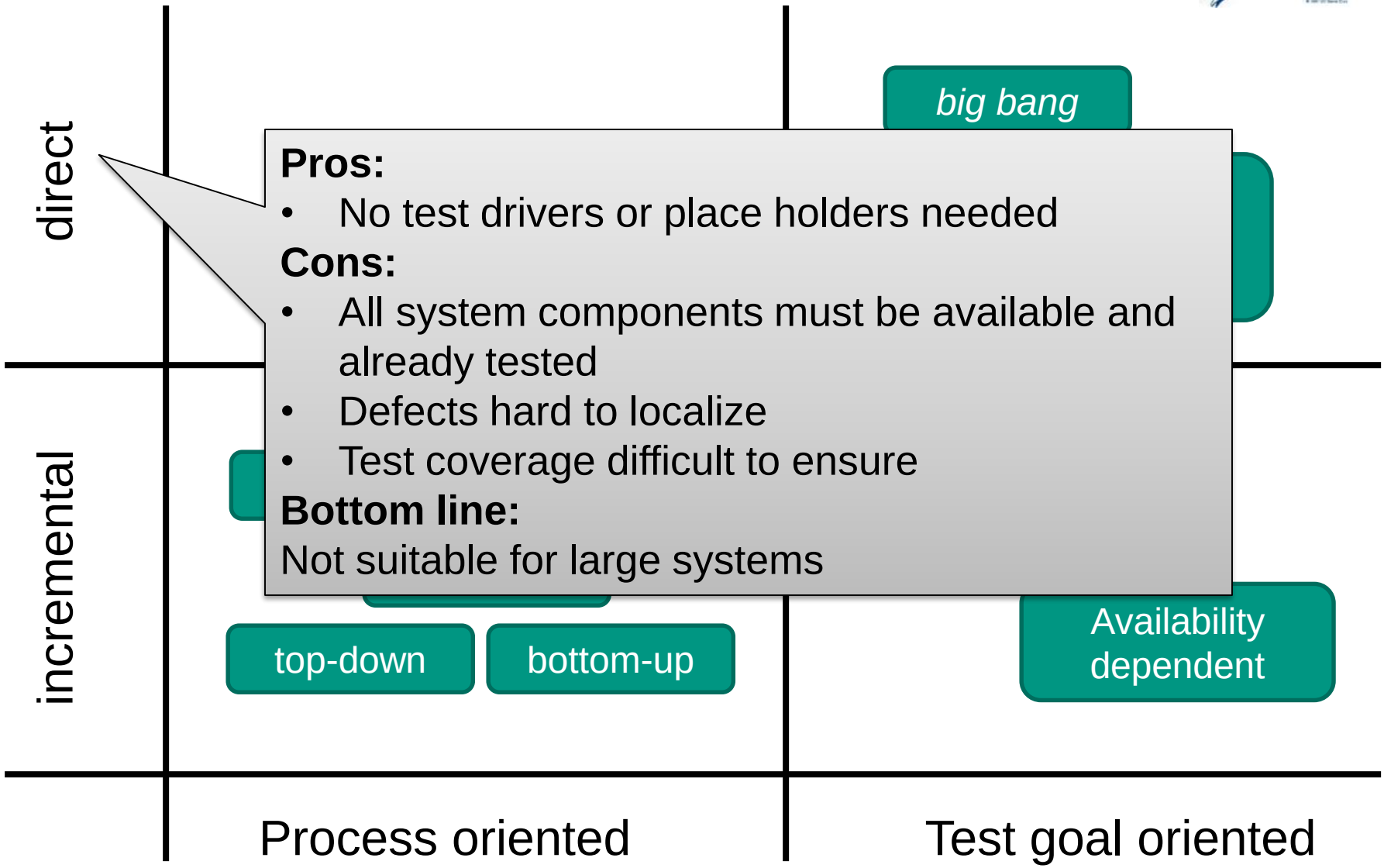
Integration Test

- Prerequisite:
  each component already underwent unit test
- Step-by-step integration
  - Integrate component into integration-tested component set
  - … run new round of integration tests
  - … until entire system is integration tested
  - Successively replace stubs and dummies with actual components

Integration Strategies



Integration Strategies

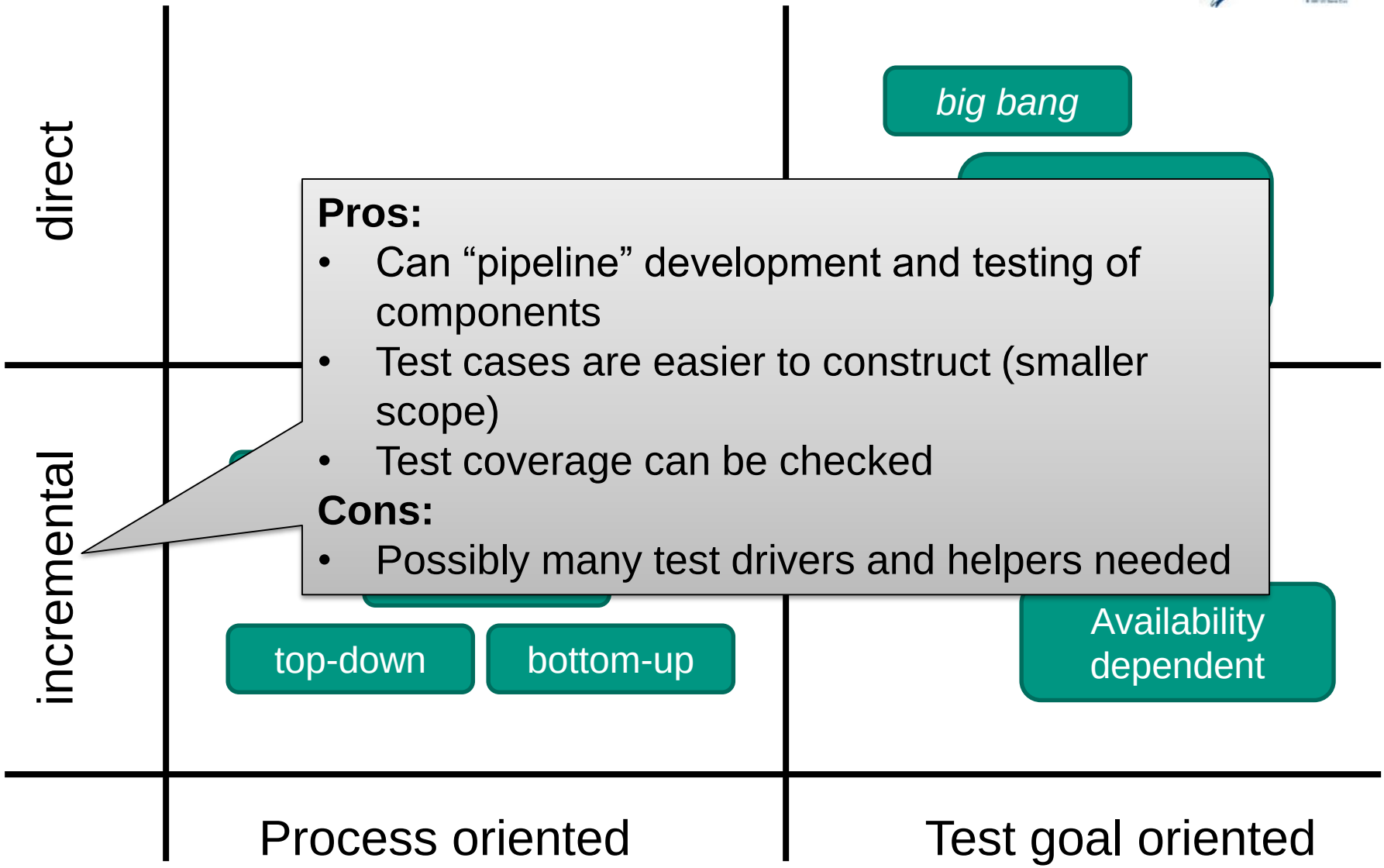


Pros:
- No test drivers or place holders needed

Cons:
- All system components must be available and already tested
- Defects hard to localize
- Test coverage difficult to ensure

Bottom line:
Not suitable for large systems

Integration Strategies



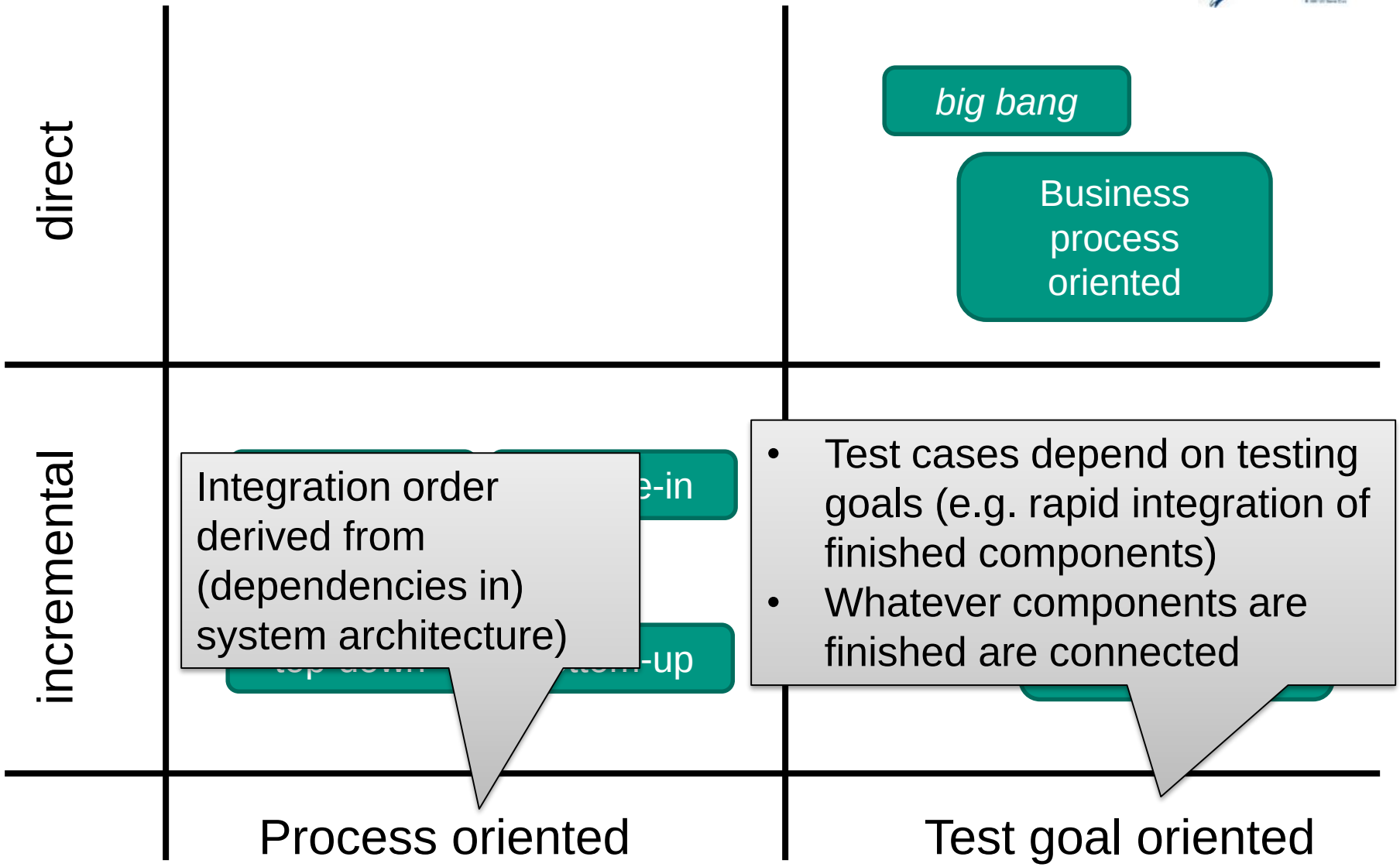
Pros:

- Can "pipeline" development and testing of components
- Test cases are easier to construct (smaller scope)
- Test coverage can be checked

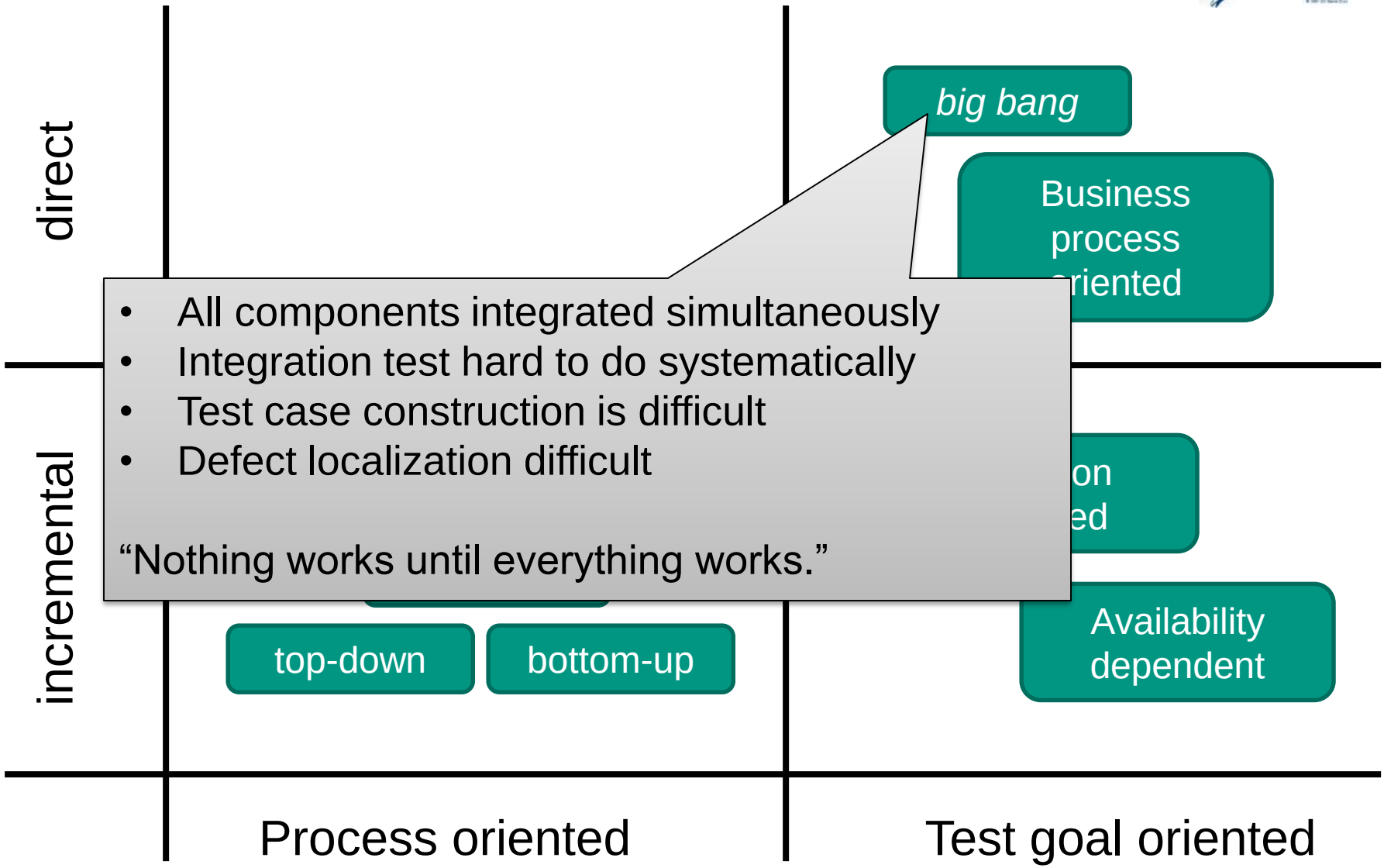
Cons:

- Possibly many test drivers and helpers needed

Integration Strategies



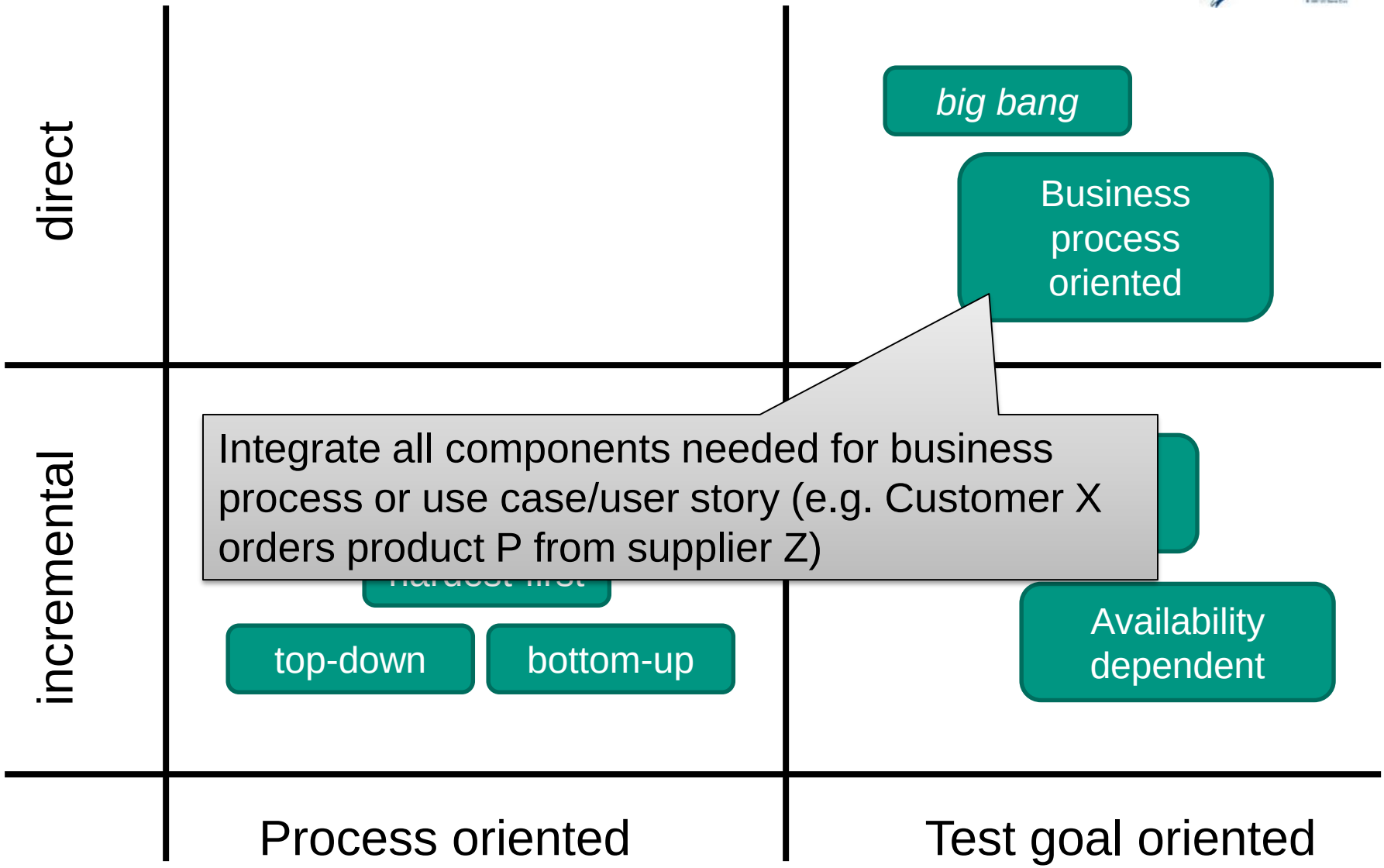
Integration Strategies



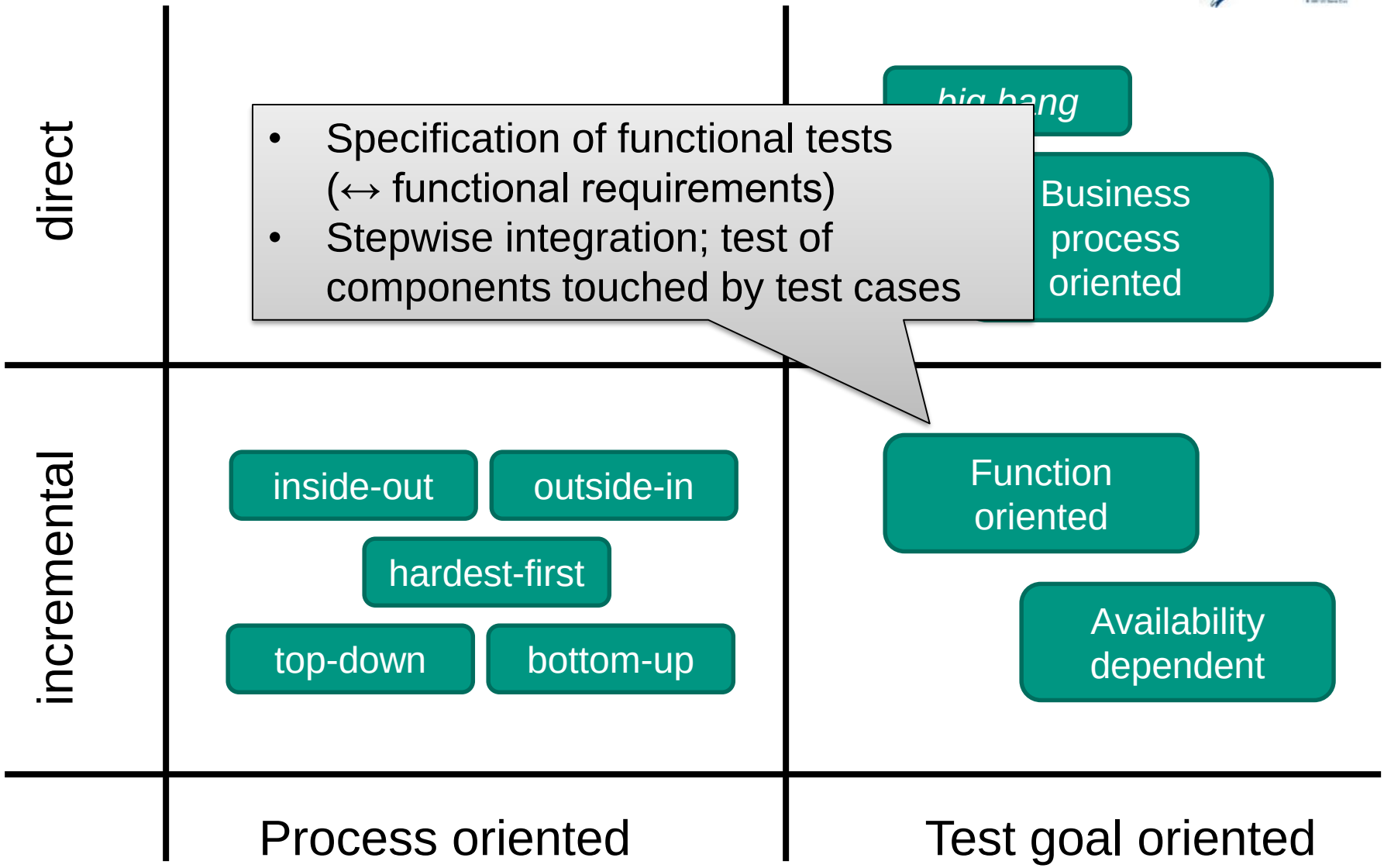
- All components integrated simultaneously
- Integration test hard to do systematically
- Test case construction is difficult
- Defect localization difficult

"Nothing works until everything works."

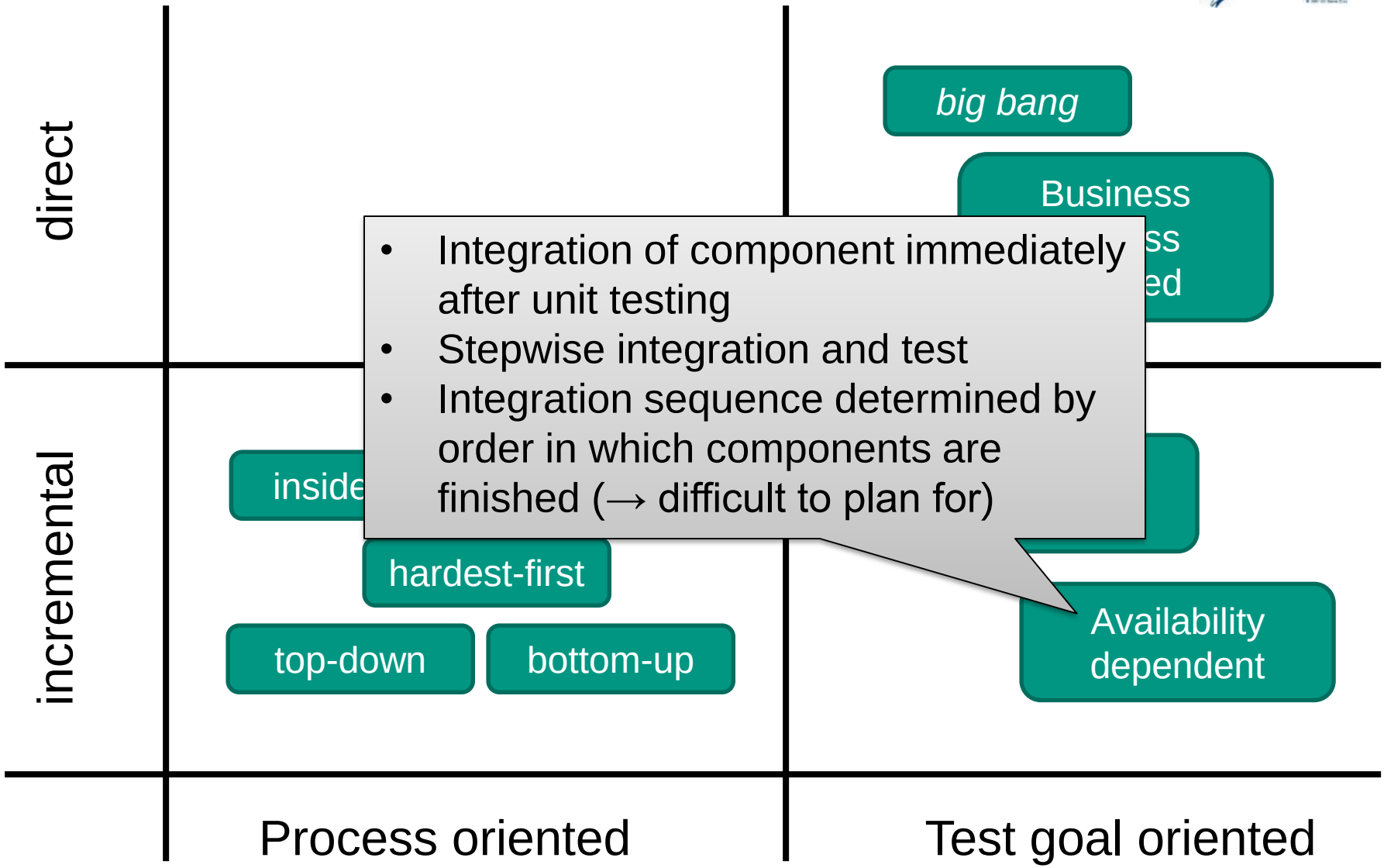
Integration Strategies



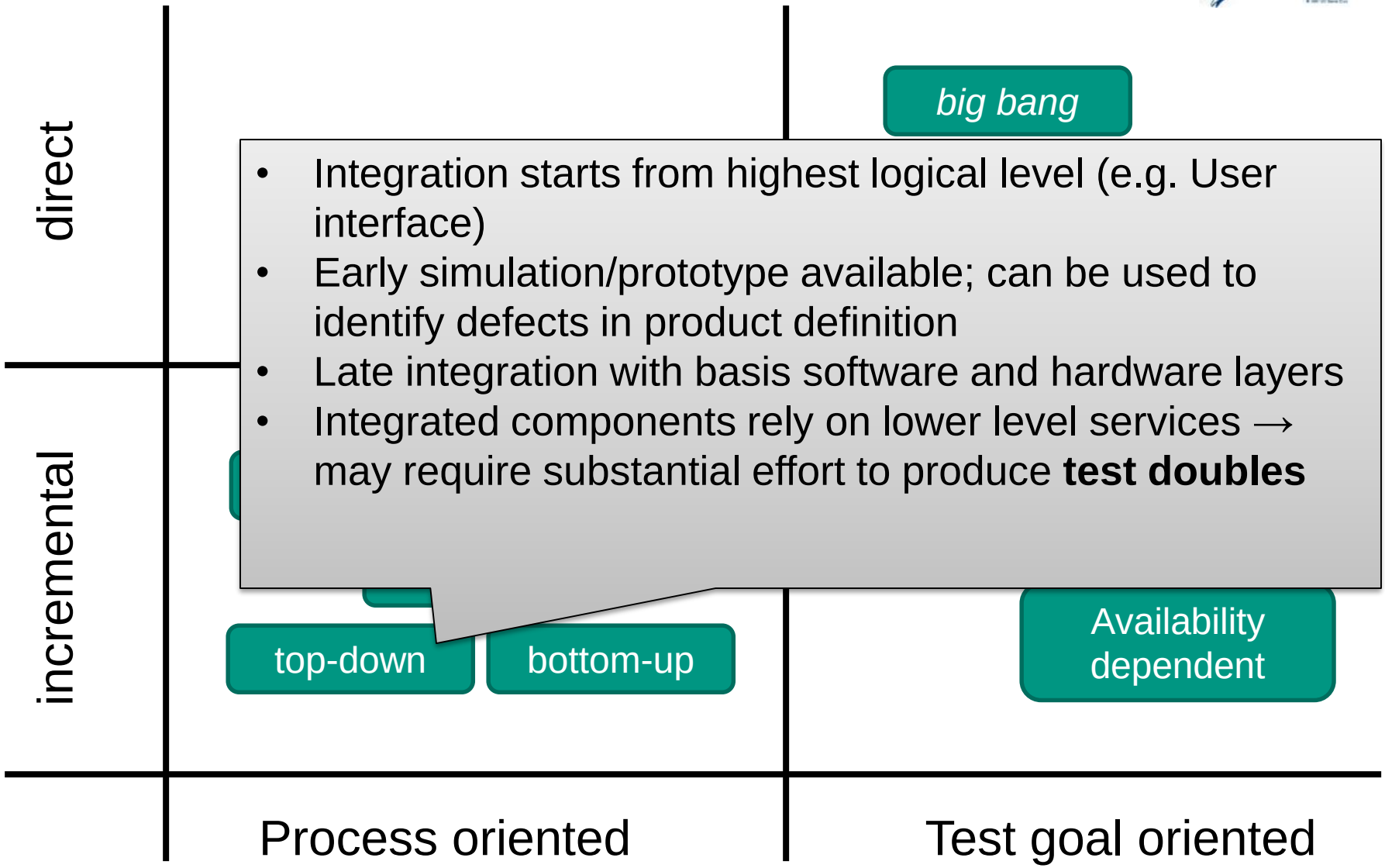
Integration Strategies



Integration Strategies

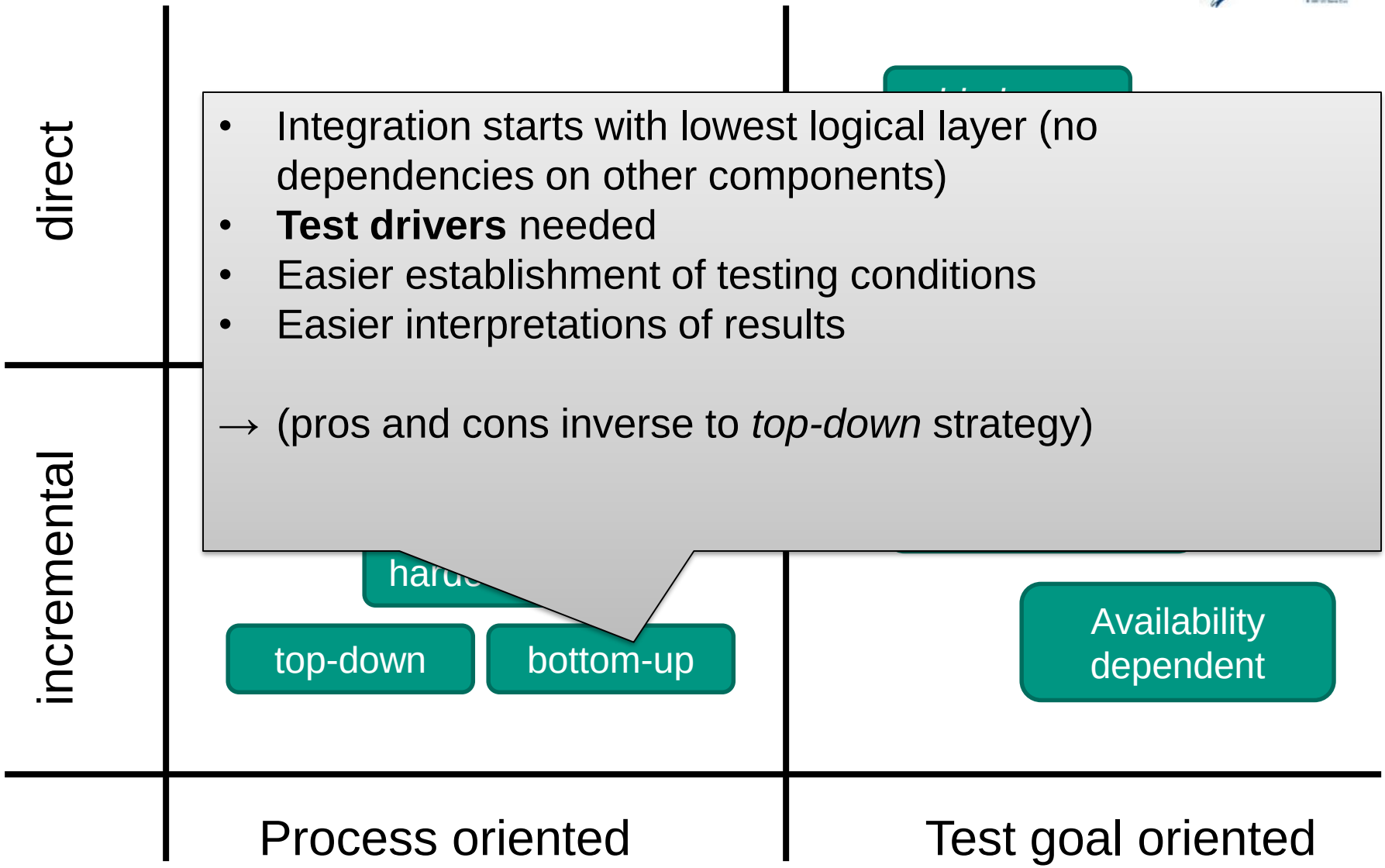


Integration Strategies



- Integration starts from highest logical level (e.g. User interface)
- Early simulation/prototype available; can be used to identify defects in product definition
- Late integration with basis software and hardware layers
- Integrated components rely on lower level services → may require substantial effort to produce **test doubles**

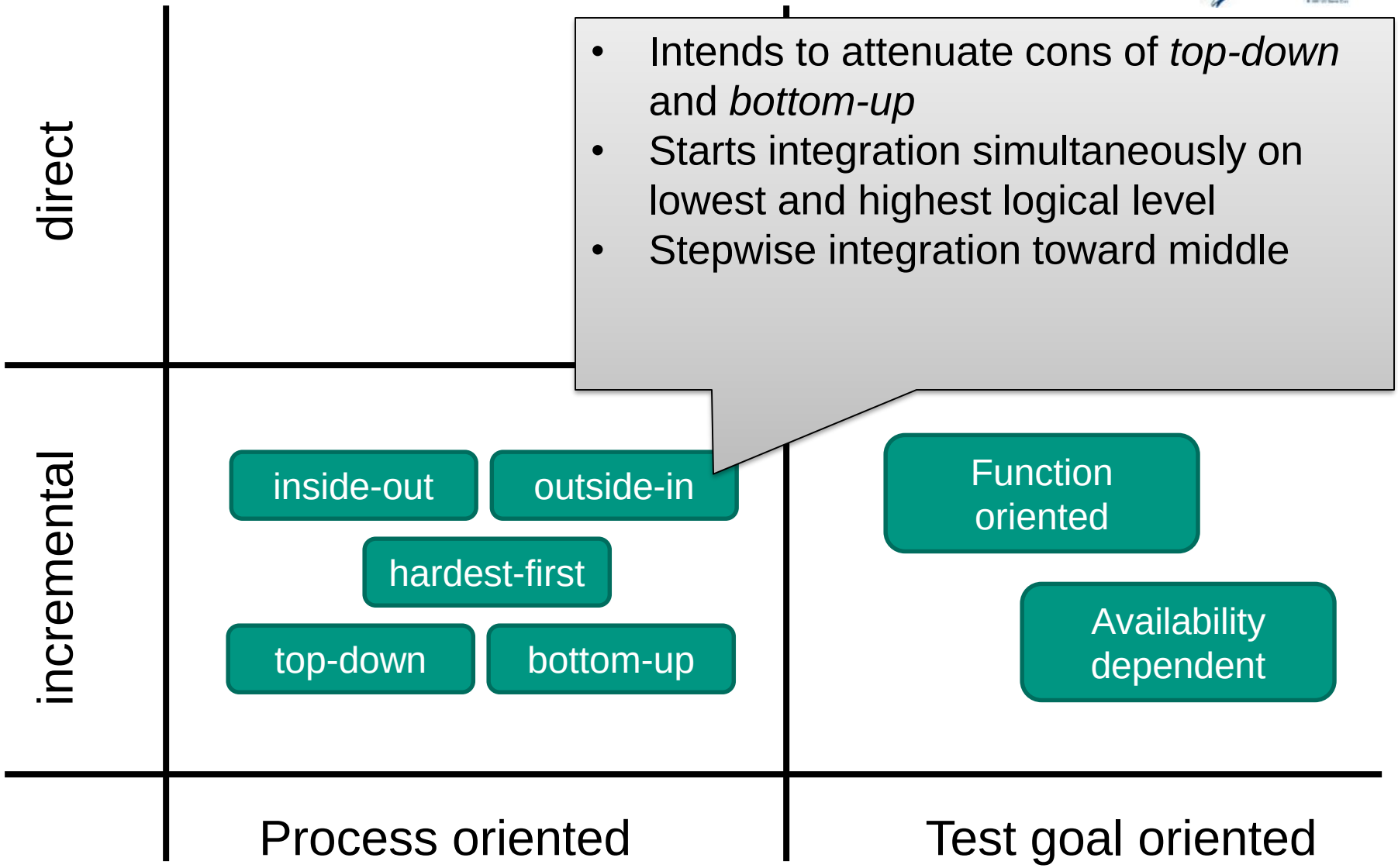
Integration Strategies



- Integration starts with lowest logical layer (no dependencies on other components)
- **Test drivers** needed
- Easier establishment of testing conditions
- Easier interpretations of results

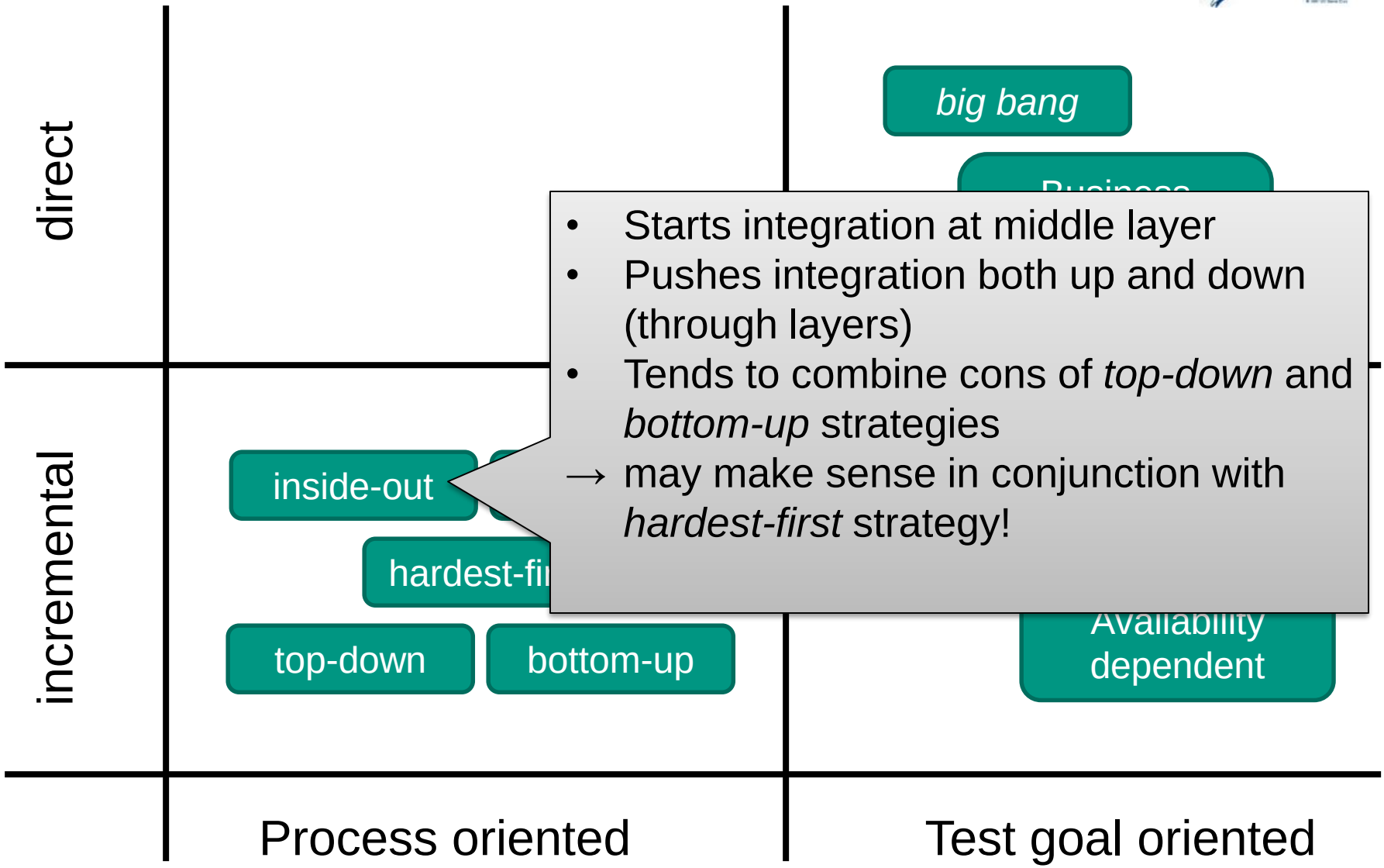
→ (pros and cons inverse to *top-down* strategy)

Integration Strategies



- Intends to attenuate cons of *top-down* and *bottom-up*
- Starts integration simultaneously on lowest and highest logical level
- Stepwise integration toward middle

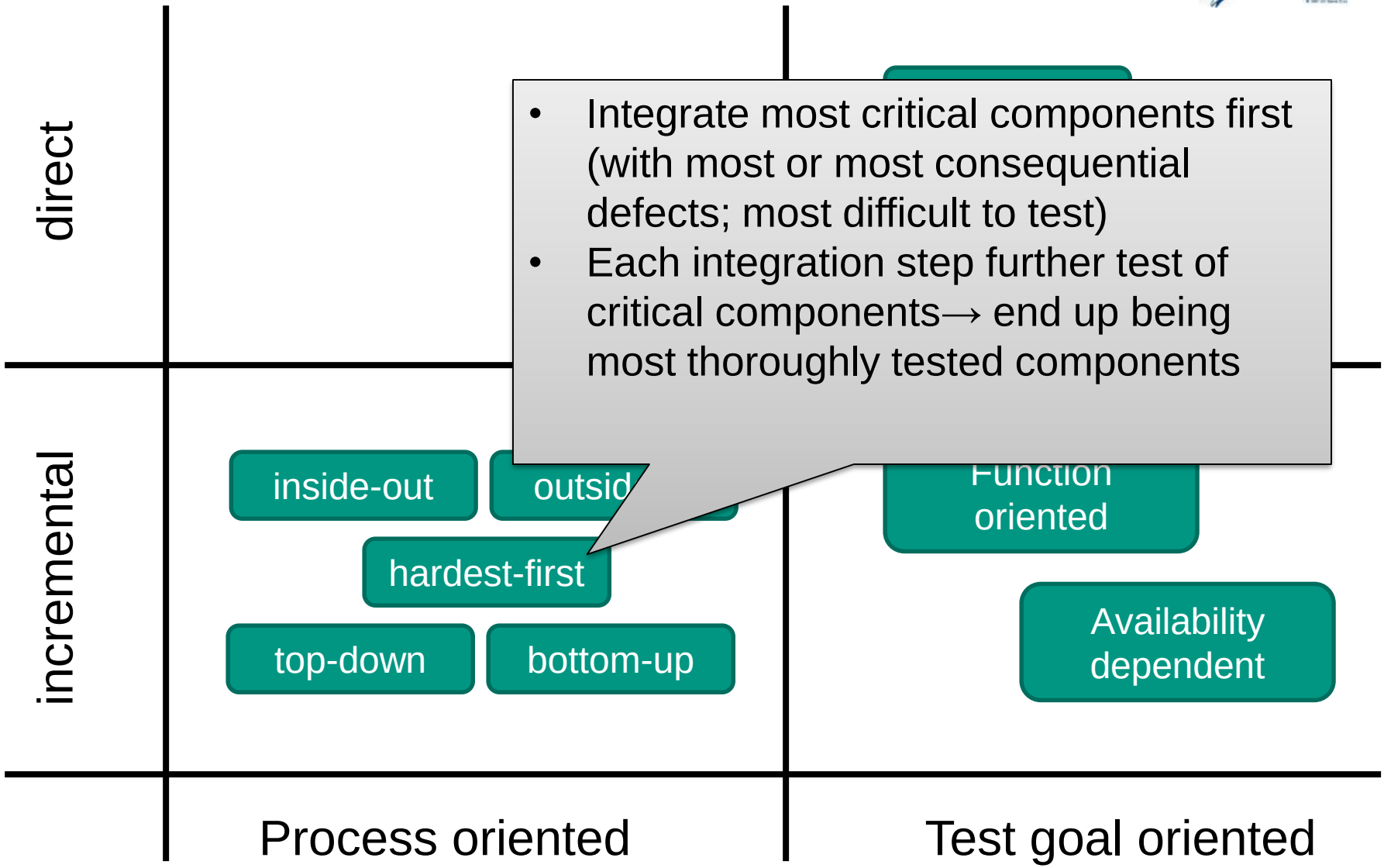
Integration Strategies



- Starts integration at middle layer
- Pushes integration both up and down (through layers)
- Tends to combine cons of *top-down* and *bottom-up* strategies

→ may make sense in conjunction with *hardest-first* strategy!

Integration Strategies



- Integrate most critical components first (with most or most consequential defects; most difficult to test)
- Each integration step further test of critical components→ end up being most thoroughly tested components

Overview Matrix

Phase						
Acceptance test						
System test						
Integration test ✓						
Component test	✓		✓	✓	✓	✓
Method	Control flow oriented	Data flow oriented	Functional tests	Performance tests	Manual V&V methods	Automated V&V tools

System Test

- Check complete system against product definition
  - a verification step: did we realize the specification correctly?
- System viewed as “black box”
  - Only external view of system (actor views)
    - User interface, interfaces to external systems
- Realistic test environment
  - reasonable approximation to employment environment
    - Embedded system, part of mechanical/software system, …
- Often performed by QA team of software development organization, separate from developer team

System Test Classification

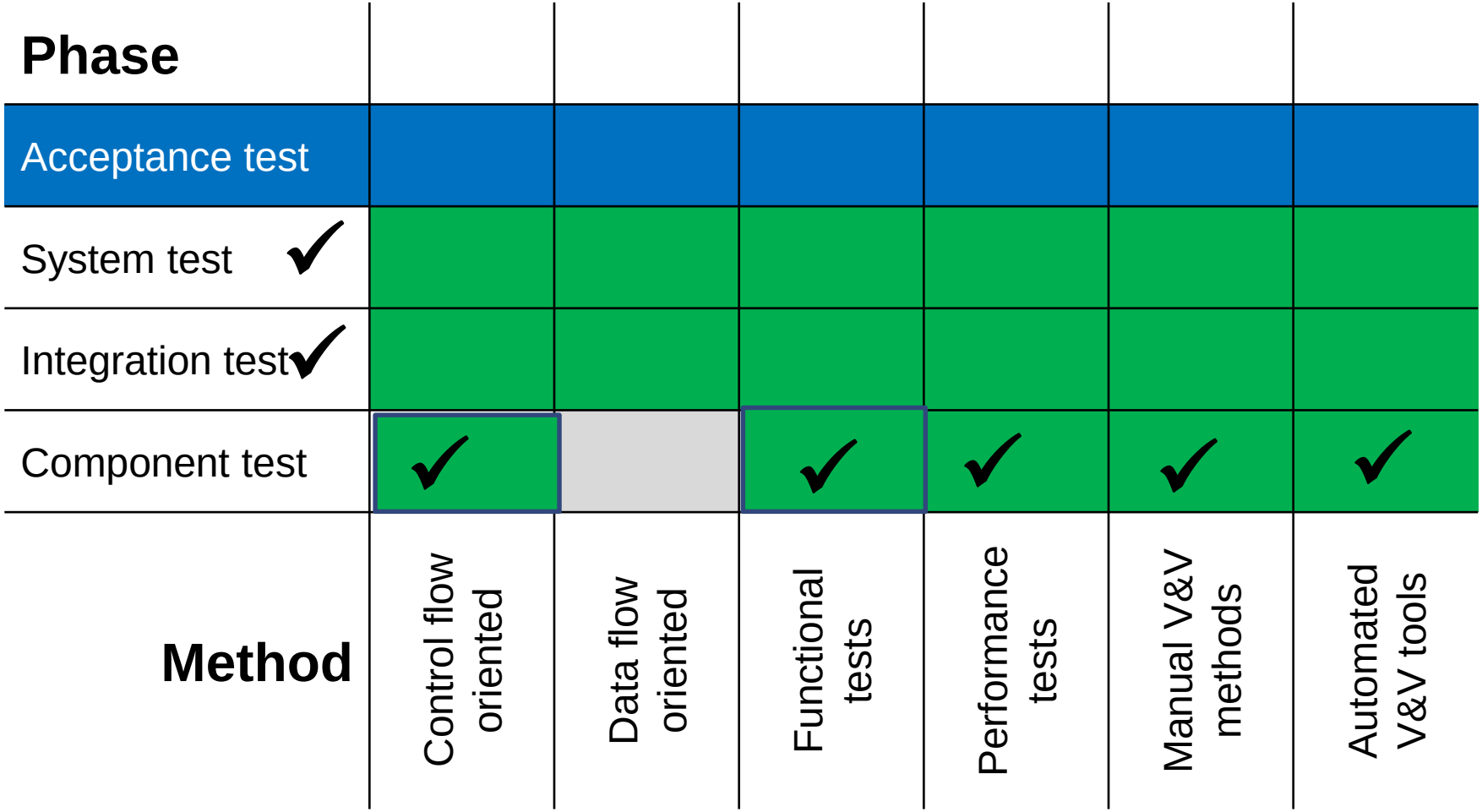
- Classified along functional and non-functional requirements
  - Functional system test:
    checks functional quality attributes
    - Correctness
    - Completeness
  - Non-functional system test:
    checks non-functional quality attributes
    - Security
    - Usability
    - Interoperability
    - Documentation
    - Availability (“up-time”)

... and of course performance tests!

Regression Test

- Definition:
  a regression test is the re-running of a system test suite to re-establish system health after changes
- Purpose:
  make sure improvements did not (re)introduce defects
- Test evaluation:
  compare latest test results with prior test results

Overview Matrix



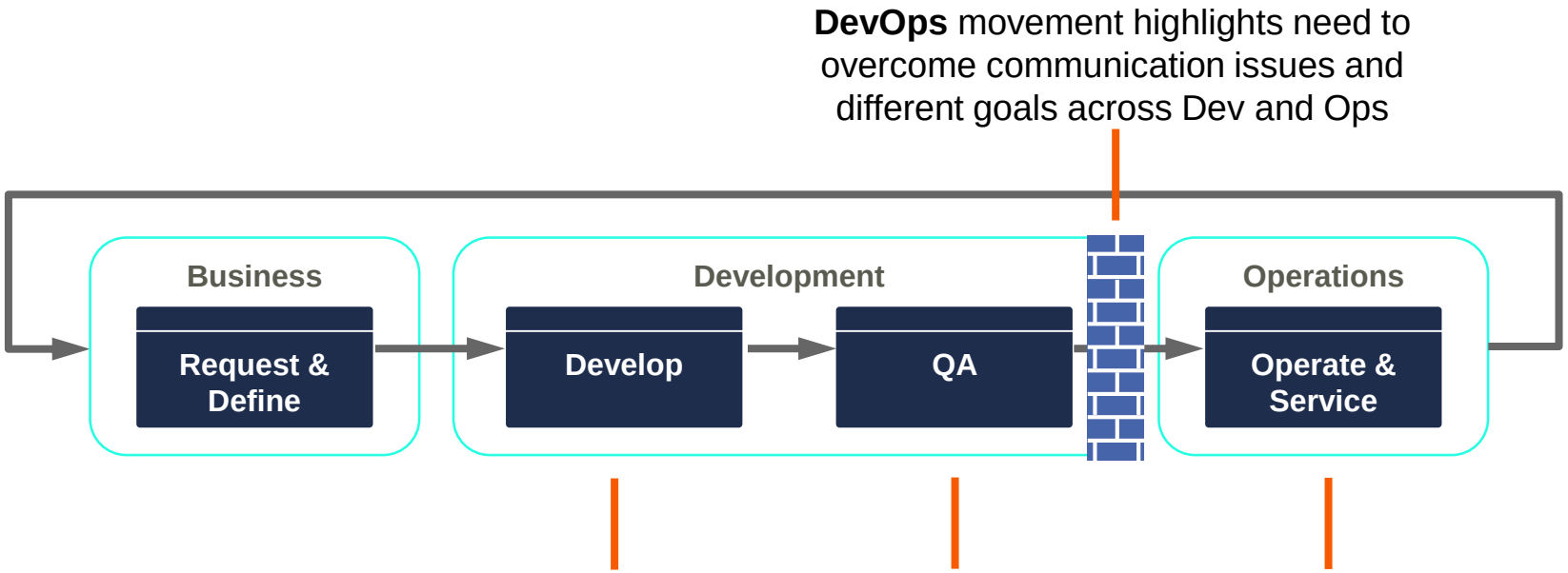
Acceptance Test

- Special form of system test
  - A validation test: customer establishes that system conforms to expectations (what s/he thought she ordered)
  - Customer/end users observe/participate/take lead
  - System tested in actual deployment environment
  - If possible, real (even live) data used
- Acceptance test is the basis for the legally binding acceptance of the product by the customer
  - Acceptance test protocol often established as part of contractual arrangement between customer and development organization

DevOps

- Concept developed over last 10-15 years
- Integration of continuous development, deployment, and operation

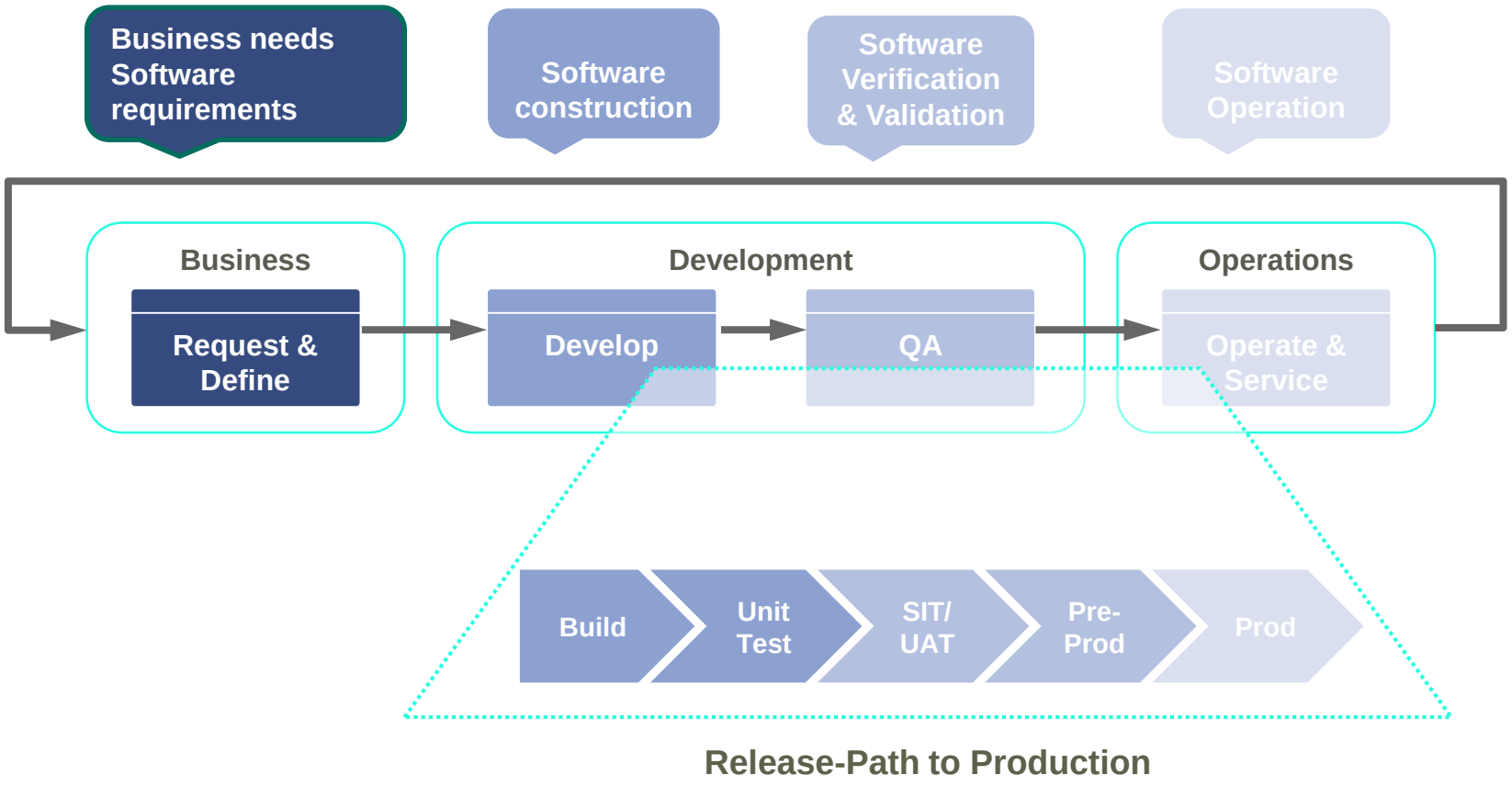
Handoffs and Complexity Create Release Challenges



Enterprise DevOps: Examples

- Salesforce.com
  - 9,000 code changes
  - 1.1 million tests per hour
  - over 600,000 files
- Facebook
  - Single Chef server
  - Full OS and app deployments to 17,000 servers

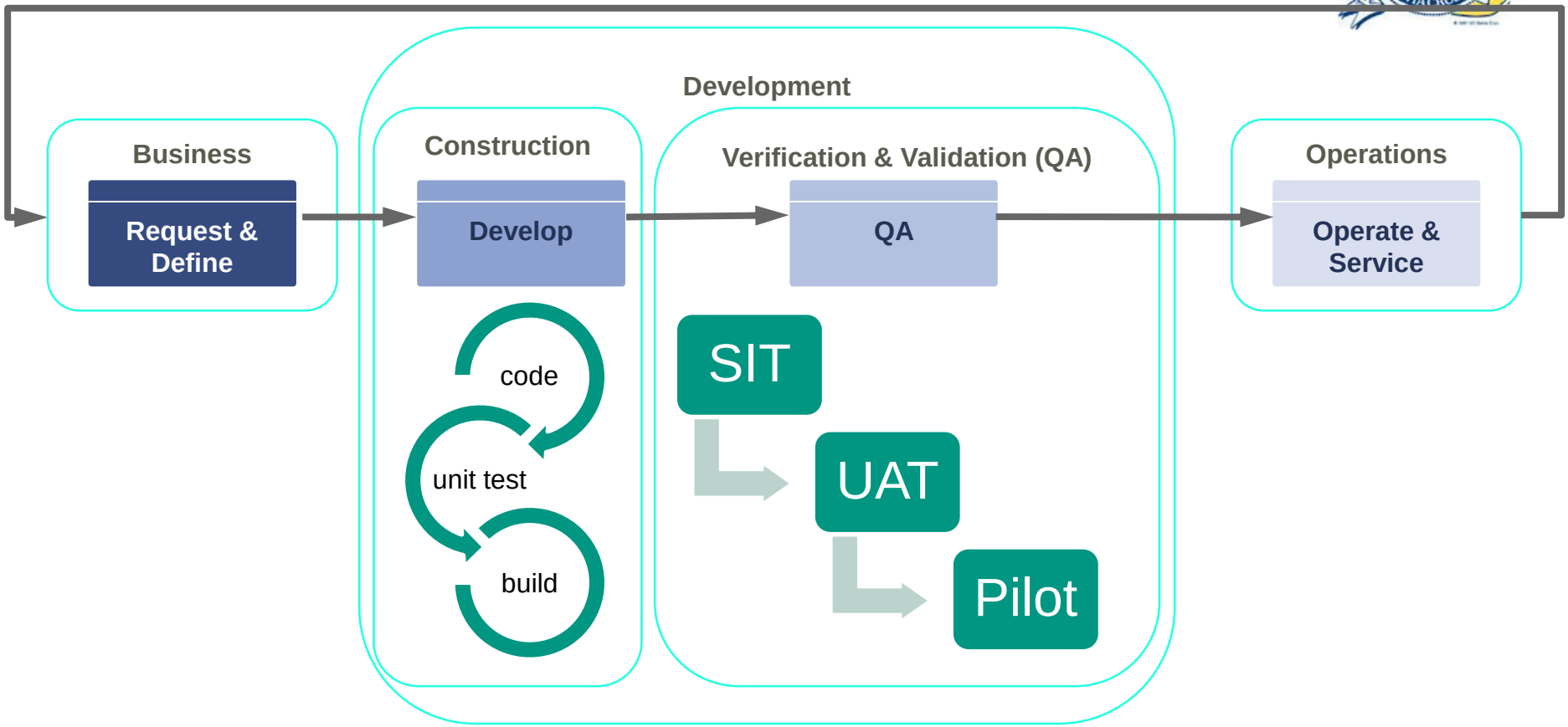
DevOps: Integration of Development and Operation



SIT: system integration test
Integration test: system internal
System test: system/environment
QA: Verification (correctly built)

UAT: user acceptance test
Joint QA/Customer org.
Validation (fitness for purpose)

Enterprise SW Lifecycle



Functional requirements
User stories
➔
Acceptance criteria

Development Environment

Pre-Production Environment

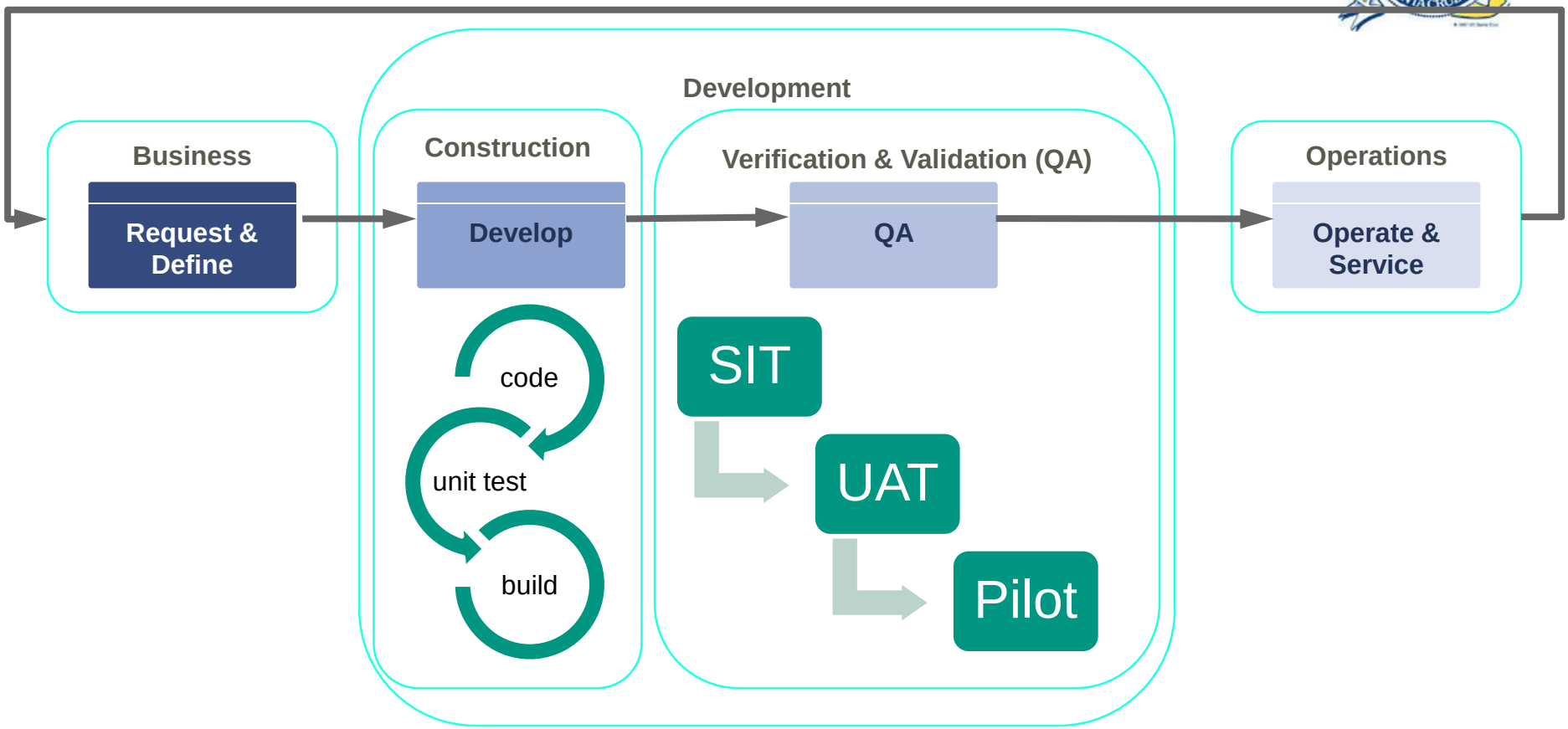
Production Environment

Continuous Build

Continuous Testing

Continuous Deployment

Enterprise SW Lifecycle



Functional requirements
User stories
➔ Acceptance criteria

Continuous Integration (Build)
Tools: (Jenkins)
Version control
Unit test suite
Autom. builds

Continuous Deployment
Tools:
Configuration management
Infrastructure provisioning
Exploratory testing
Automated testing

Continuous Deployment
Tools:
Workflow
History tracking
Change management
Bug tracking