Testing – A Tutorial

Alexander Klaus
Fraunhofer IESE
Dept. Information Systems Quality Assurance

Alexander.Klaus@.iese.fraunhofer.de
0631-6800 2245
Room B3.20

Fraunhofer
IESE

Agenda

- Introduction
- Functional Testing
- Structural Testing
- Unit Testing
- System Testing
- Test Documentation

Agenda

- **Introduction**
- Functional Testing
- Structural Testing
- Unit Testing
- System Testing
- Test Documentation

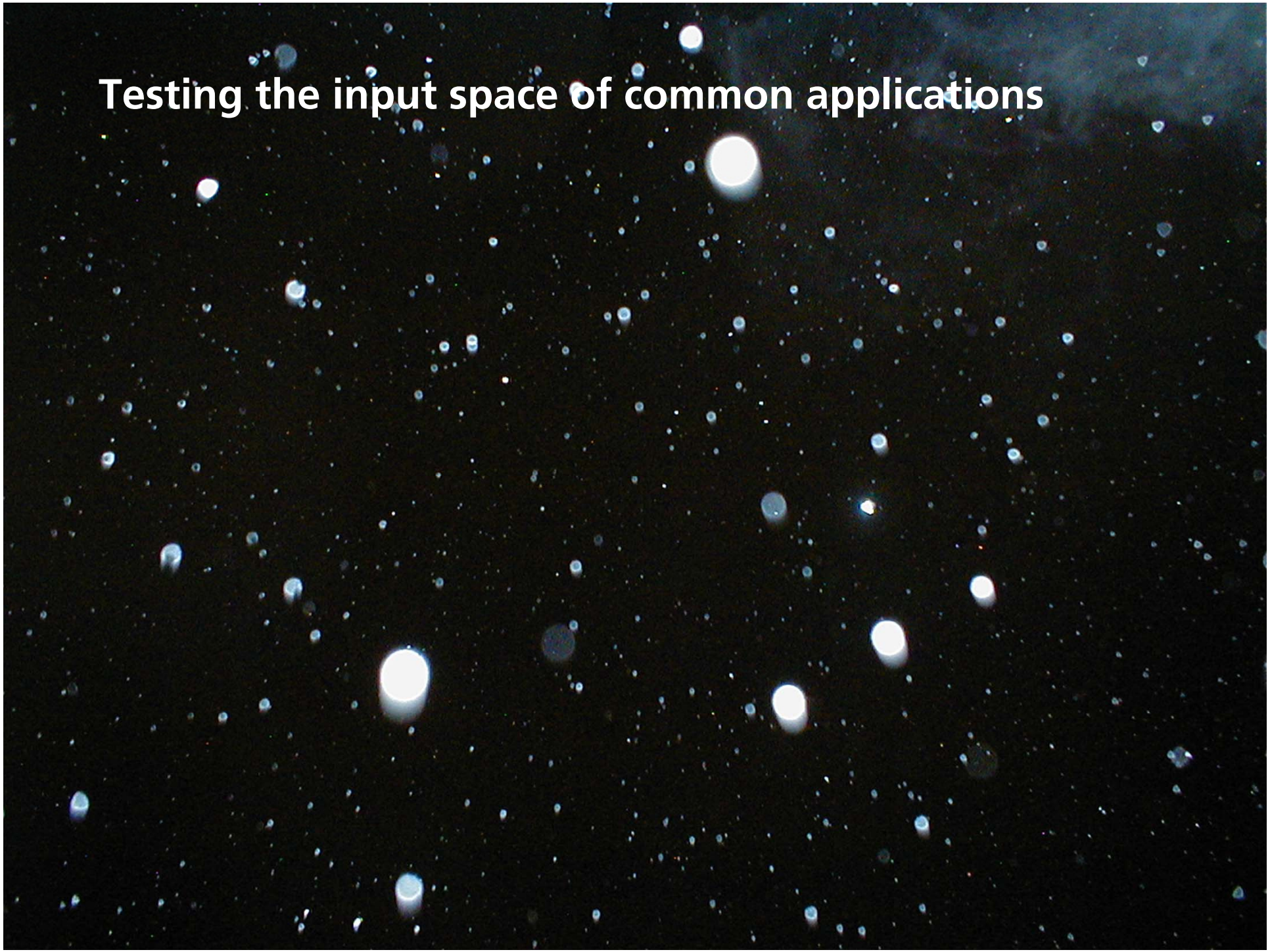
Testing –What is it?

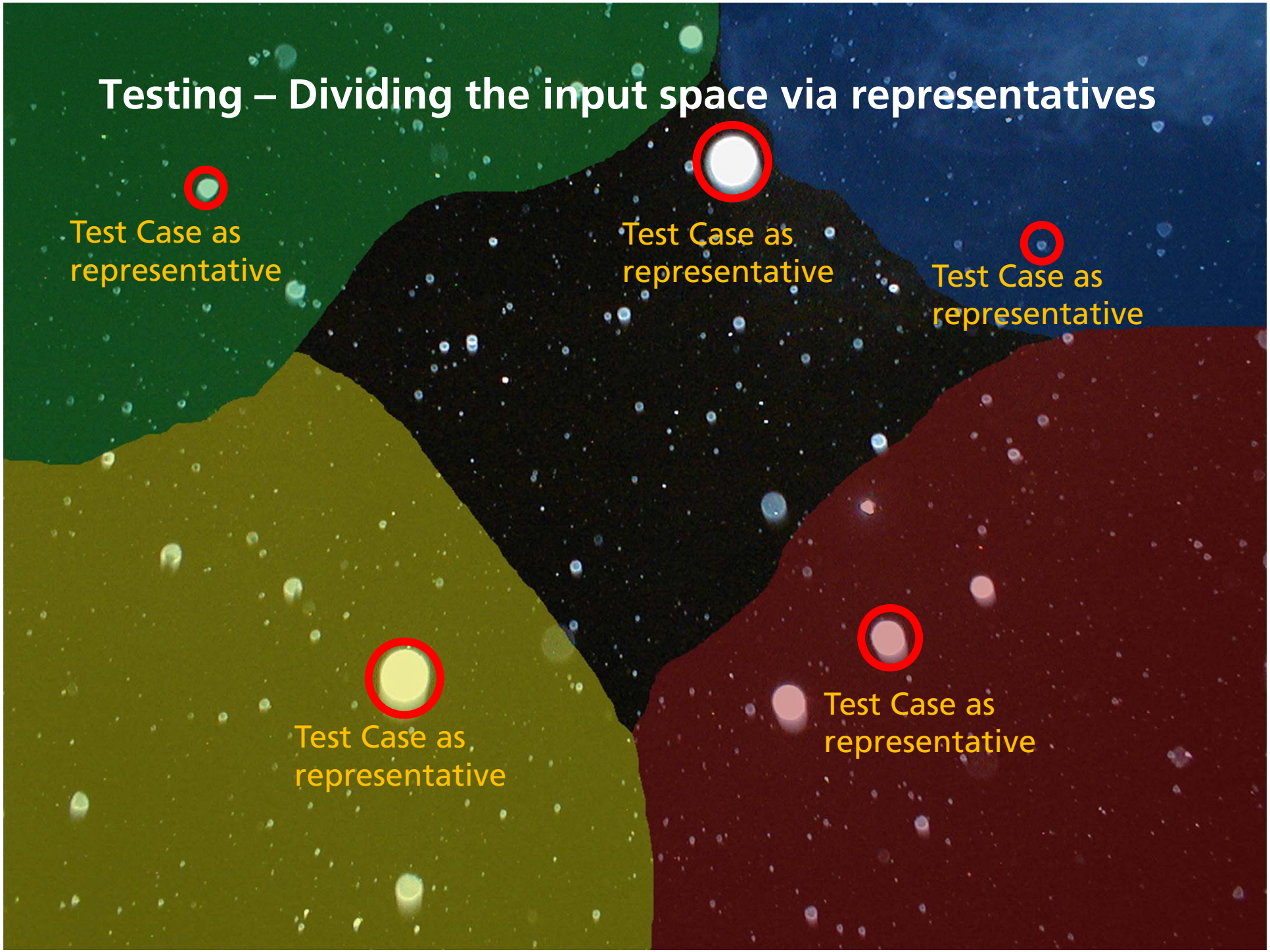
- Testing is the dynamic execution of software with real (i.e., concrete) input
- Testing has sampling character – it is incomplete
  - Most software cannot be tested with all possible combinations of input
- A statement of correctness of the software is possible only for the input used

> Testing can demonstrate the presence of errors, not their absence. (Dijkstra)

- We have to choose our tests as representative as possible
  → one test should represent as many inputs as possible

→ For this we have our test techniques


Testing the input space of common applications


Testing – Dividing the input space via representatives
Test Case as representative
Test Case as representative
Test Case as representative
Test Case as representative
Test Case as representative

Test levels

Acceptance testing

- Test in the customers environment, by the customer

System testing

- Test of functionality and performance of a software against requirements and customer requests
- Against requirements specification

Integration testing

- During step-by-step integration of the modules of a software to a whole system
- Testing of correct interaction and communication between modules
- Against interface specifications

Module / Unit testing

- Isolated check of a single module (and the structure)
- Test of correct and complete realization of the module specification
- Against unit specifications

Testing and object-orientation

Special features of object-oriented software

- Associations („method calls“ / „messages“)
  - Lead to dependencies - not all functions can be tested in isolation
  - Sometimes there is a need for mock-objects and stubs
    - provide functions to be tested with values (no real functionality)
    - enable testing of functions which use other functions not yet implemented
- Inheritance of objects
  - Order of development / testing: from general to special classes
  - Just more functions: ok, add some tests
  - Basis functions are overwritten: new test cases replace old ones for specialized classes

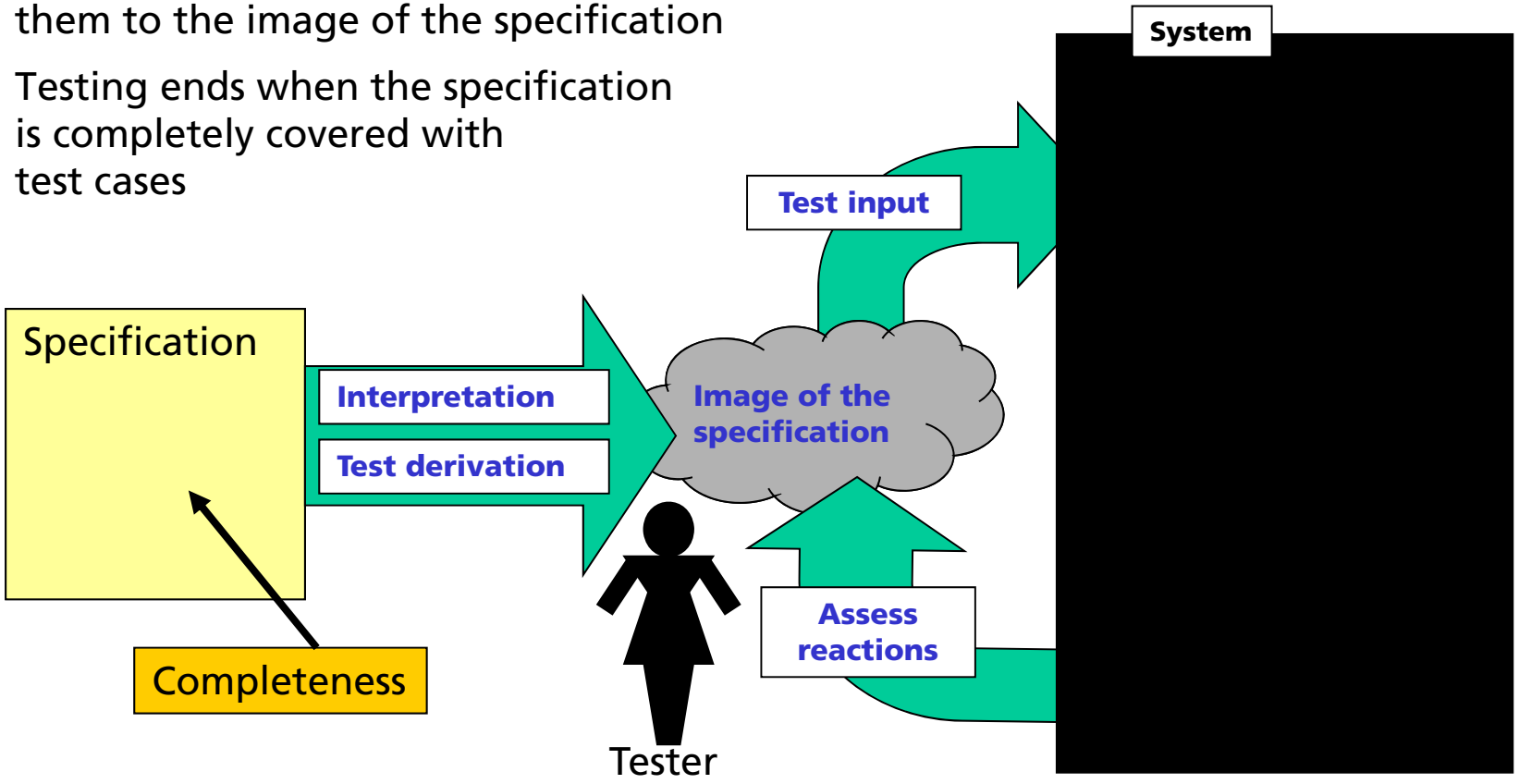
Agenda

- Introduction
- **Functional Testing**
- Structural Testing
- Unit Testing
- System Testing
- Test Documentation

Functional Testing

Black-box testing technique

- Assess completeness of tests by comparing them to the image of the specification
- Testing ends when the specification is completely covered with test cases



Functional equivalence class construction

Principle: Reduce complexity until a rather easy selection of test cases is possible

- Result: Set of “equivalence classes”
  - Set of data, for which the system shows the **same reactions**
  - → Every single value of a certain equivalence class is seen as representing all values in this equivalence class
- Approach: continuous case differentiation

Functional equivalence class construction

Case differentiation

- Regarding input and output conditions

Example: A specification demands positive values as input for a certain function

- Case differentiation
  - Positive input
  - Non-positive input

We can differentiate between **valid and invalid equivalence classes**

- In this example:
  - Positive values = valid equivalence class (i.e., as specified / expected)
  - Non-positive values = invalid equivalence class

Functional equivalence class construction

Case differentiation regarding

- Input: as seen
- Output: differentiation of output classes
  - Choose values as input which create output in the chosen equivalence class
  - Example: Specification: output = 1<= X <= 99
    - **One valid** equivalence class
      - Each input which creates an output between 1 and 99
    - **Two invalid** equivalence classes
      - Each input which creates an output smaller than 1
      - Each input which creates an output greater than 99

Functional equivalence class construction

Based on these equivalence classes choose test cases

Use **Boundary value analysis**

- Test the boundaries of equivalence classes
  - Experience shows: Faulty behavior is often produced when values are on the boundary of an equivalence class
- Possible addition:
  - Test of special values, like “0”
  - Choose values randomly

Functional equivalence class construction

Rules for constructing equivalence classes:

("Myers: The art of software testing, John Wiley & Sons, New York, 1979")

1. If an **input condition** specifies a **value range**, **one valid** and **two invalid** equivalence classes are to be formed

2. If an **input condition** specifies a **number of values**, **one valid** and **two invalid** equivalence classes are to be formed

Functional equivalence class construction

Rules for constructing equivalence classes:

("Myers: The art of software testing, John Wiley & Sons, New York, 1979")

3. If an **input condition** specifies a **quantity of values** which are to be **processed differently**, a **separate valid** equivalence class has to be formed **for each value**. For **all other values** except the valid values **one invalid** equivalence class is to be formed.

- Example: "stringed instruments: guitar, violin, viola, bass"
  - Four valid equivalence classes: guitar, violin, viola, bass
  - One invalid equivalence class: everything else, e.g. piano, drums, …

Functional equivalence class construction

Rules for constructing equivalence classes:

("Myers: The art of software testing, John Wiley & Sons, New York, 1979")

4. If an **input condition** specifies a **certain situation** which **must be fulfilled**, **one valid** and **one invalid** equivalence class are to be formed

Functional equivalence class construction

Rules to be followed

- Functions receiving input from input-/ output-channels
  - Form valid and invalid equivalence classes
  - Reason: error handling routines for invalid input must be implemented
- subordinate functions
  - Comply to limitations regarding input
  - Calling function may already implement error handling routines
    - Subordinate function is likely to NOT implement an additional error handling
      - Would lead to dynamically unreachable code
- Dependencies between input values
  - Some combinations of input values may not be applicable

Functional equivalence class construction

Building test cases

- Two rules

a) Derive test cases regarding **valid** equivalence classes by selecting test data from **as many** valid equivalence classes as possible
   - Reduces number of test cases regarding valid equivalence classes to a minimum

b) Derive test cases regarding **invalid** equivalence classes by combining test data from exactly **one** invalid equivalence class with test data from valid equivalence classes
   - When using two or more invalid input values at the same time, which value caused the error handling to be processed?

Functional equivalence class construction

Example – Specification

A routine of a booking system computes invoices from private and business customers.

For computing, the system requires the customer ID, the invoice amount, and the receipt of payment.

Invoice amount and receipt of payment are positive numbers (minimum is one) without leading zeros, limited to six digits.

Customer IDs start with a “P” for private or a “B” for business customers.

The routine produces true, if the receipt of payment is equal to or greater than the invoice amount. The routine produces false, if the receipt of payment is less than the invoice amount.

Functional equivalence class construction

Example – Specification

A routine of a booking system computes invoices from private and business customers.

For computing, the system requires the **customer ID**, the **invoice amount**, and the **receipt of payment**.

Invoice amount and receipt of payment are *positive numbers (minimum is one) without leading zeros, limited to six digits*.

Customer IDs start with a "P" for private or a "B" for business customers.

The **routine produces** *true, if the receipt of payment is equal to or greater than the invoice amount*. The routine produces *false, if the receipt of payment is less than the invoice amount*.

Description: **Blue** and **Bold**: values; *Red* and *Italic*: conditions

Functional equivalence class construction

Example – Constructing equivalence classes

Variable	Valid equivalence classes	Invalid equivalence classes
Customer ID	1)P123 2)B135	3) D123
Invoice amount	4) 1 <= IA <= 999999	5) IA <= 0 6) IA > 999999
Receipt of payment	7) 1 <= RoP <= 999999	8) RoP <= 0 9) RoP > 999999
Routine	10) True (RoP >= IA)	11) False (RoP < IA)

Functional equivalence class construction

Example – Creating test cases with boundary value analysis

Test case	Equivalence classes used	Customer ID	Invoice amount	Receipt of Payment	Routine
1	1, 4L, 7U, 10	"P123"	1	999999	True
2	2, 4U, 7L, 11	"B135"	999999	1	**False**
3	3	**"D123"**	1	1	-
4	5	"P123"	**0**	1	-
5	6	"P123"	**1000000**	1	-
6	8	"P123"	1	**0**	-
7	9	"P123"	1	**1000000**	-

Most test cases are negative (i.e., they produce an error)!

U: Upper bound
L: Lower bound
Red and **Bold**: Values from invalid equivalence classes

Functional Testing

Remaining problem

- State-based software
  - Reactions are different dependent on the state
  - → State-based testing
- Basis: state charts
    - UML (Unified modeling language)
    - In UML, state charts are used to describe the behavior of classes
  - Here, we use state charts to derive test cases
- State-based testing aims at complete test coverage of state charts

State-based Testing

Derivation of state chart diagrams

Text based specifications for state based systems are likely to be incomplete

- Text based specifications are not sufficient for state-based systems
  - Different reactions based on actual state

In the following: example from

> "Liggesmeyer: Software Qualität. Testen, Analysieren und Verifizieren von Software, Spektrum Akademischer Verlag, Heidelberg, 2002"

State-based Testing

Example: Text based specification

Connection establishment and termination between a calling and a called telephone extension is to be realised. Initially, the connection is ***disconnected***. Whenever the receiver is put down, the software is in this state.

If the connection establishment has begun, but is not yet completed, the software is ***dialling***. If the connection establishment succeeded, the software is ***connected***.

Successful connection establishment always starts with lifting the receiver, followed by dialling several digits which form a valid telephone number.

Putting down the receiver always terminates the connection. If a ***timeout*** occurs during ***dialling***, only putting down the receiver enables returning to the initial state, ***disconnected***.

State-based Testing

This specification is incomplete

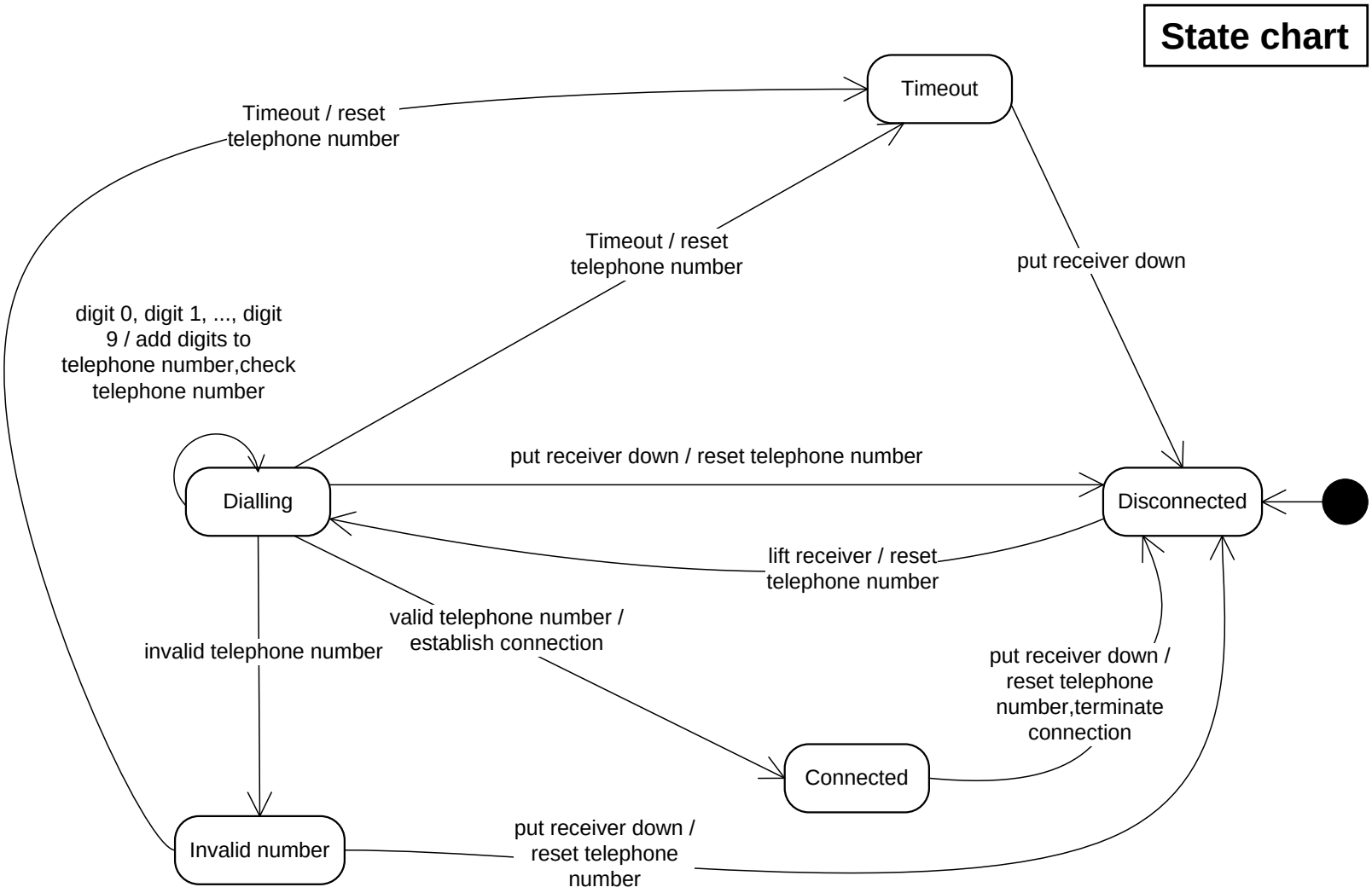
- No state regarding invalid telephone numbers

Transforming this textual specification into a graphical specification enables checking for consistency and completeness

Notation used

- State transitions: directed edges
- Causing event: written in front of “/”
- Associated action: written behind “/”
  - → Event / action
- → Initial state: filled dot

State-based Testing

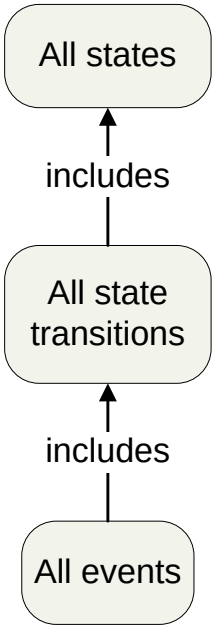


State-based Testing

Derive test cases by walking through the state chart

Test completeness criteria

- Cover all states
  - Does not guarantee all state transitions to be tested
- Cover all state transitions
  - Does guarantee all states to be processed at least once
- Cover all events
  - Reasonable, if state transitions may be caused by multiple, different events
    - Example: digits during dialling



State-based Testing

Example: Test cases for state transition coverage

(Notation: *state*, event → state, event, …)

1. *Disconnected*, lift receiver → *Dialling*, put receiver down→ *Disconnected*
2. *Disconnected*, lift receiver → *Dialling*, Timeout → *Timeout*, put receiver down → *Disconnected*
3. *Disconnected*, lift receiver → *Dialling*, Digit 0 … Digit 9 → *Dialling*, Digit 0 …Digit 9 → *Dialling*, valid telephone number → *Connected*, put receiver down → *Disconnected*
4. *Disconnected*, lift receiver → *Dialling*, Digit 0 … Digit 9 → *Dialling*, Digit 0 …Digit 9 → *Dialling*, Digit 0 … Digit 9 → *Dialling*, Digit 0 … Digit 9 → *Dialling*, invalid telephone number → *Invalid number*, put receiver down→ *Disconnected*
5. *Disconnected*, lift receiver → *Dialling*, Digit 0 … Digit 9 → *Dialling*, Digit 0…Digit 9 → *Dialling*, Digit 0 … Digit 9 → *Dialling*, Digit 0 … Digit 9 → *Dialling*, invalid telephone number → *Invalid number*, Timeout → *Timeout*, put receiver down → *Disconnected*

State-based Testing

This state chart is not complete

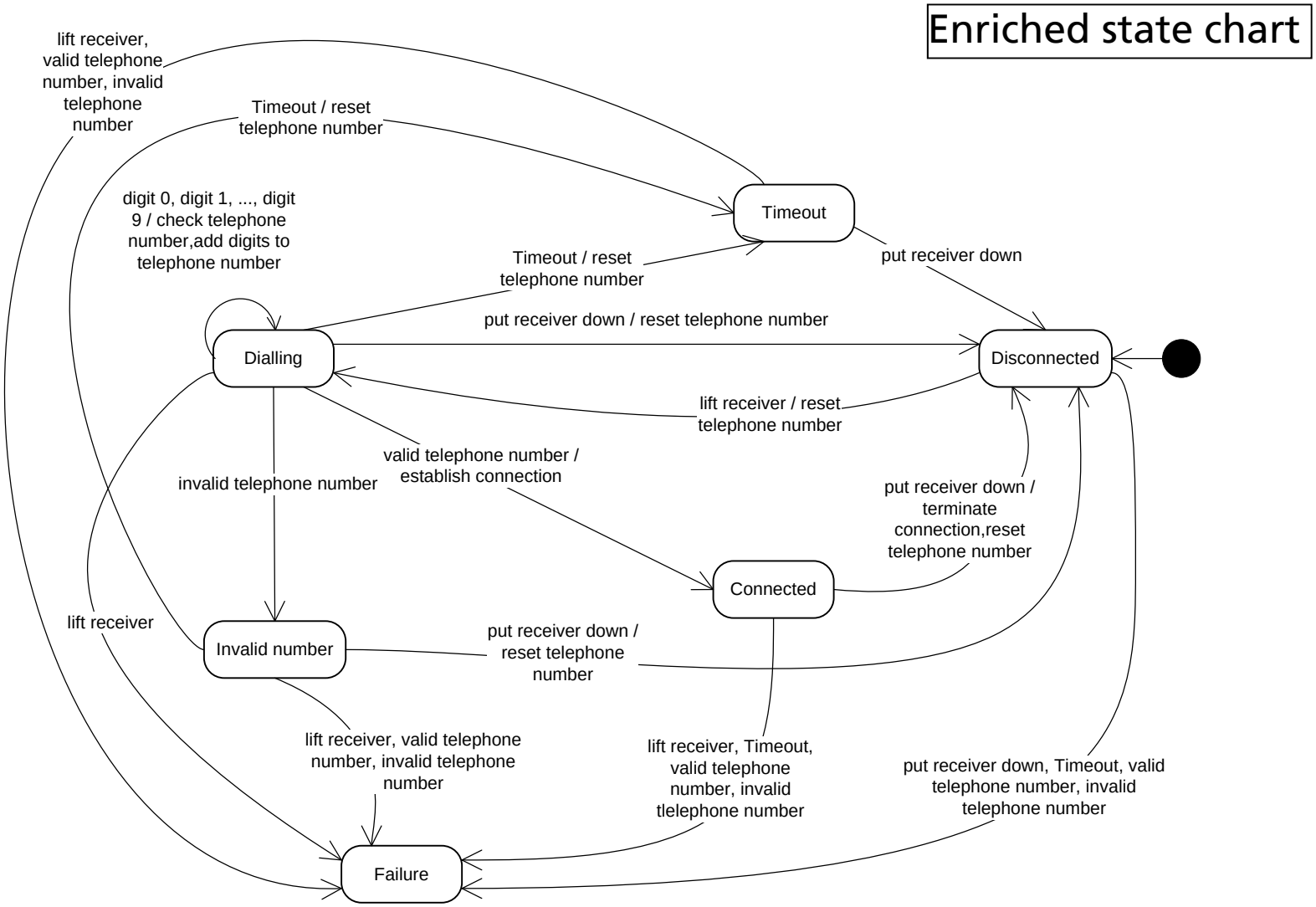
- Describes desired behavior
- → All test cases describe regular cases

Test cases for error situations are not included and therefore likely to be neglected

Mitigation: add an error state to the state chart

- All events not shown are treated as if they were ignored
  - E.g.: dialling digits when disconnected

State-based Testing



State-based Testing

Now we could test all combinations

- This is often not possible due to resource limitations
- How to select test cases?

Normally, three situations are distinguished for the selection

- Events that cause an action / transition if they appear in a certain state
- Events that may be ignored if they appear in a certain state
- Events that require an error handling if they appear in a certain state

The last category is covered by test cases generated from the enriched state chart

State-based Testing

Suitable for module and integration testing

Disadvantage

- Complex systems: State explosions
  - Only weak coverage is possible

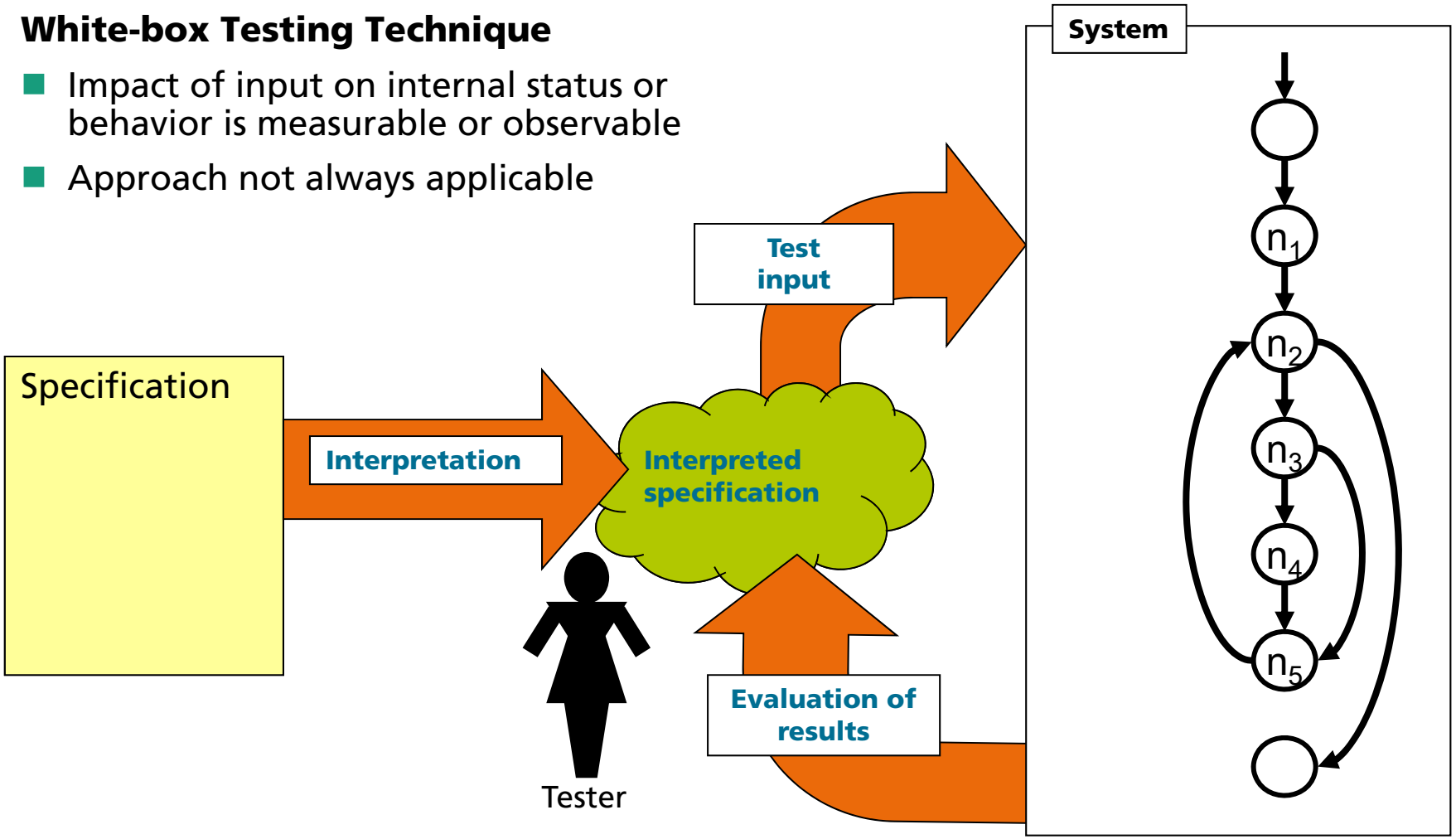
Agenda

- Introduction
- Functional Testing
- **Structural Testing**
- Unit Testing
- System Testing
- Test Documentation

Structural Testing

White-box Testing Technique

- Impact of input on internal status or behavior is measurable or observable
- Approach not always applicable



Structural Testing

- Evaluation of adequacy, completeness (=test quality) and test case development based on the module (program code) structure
- Evaluation of test results according to the module specification

Control flow-oriented techniques

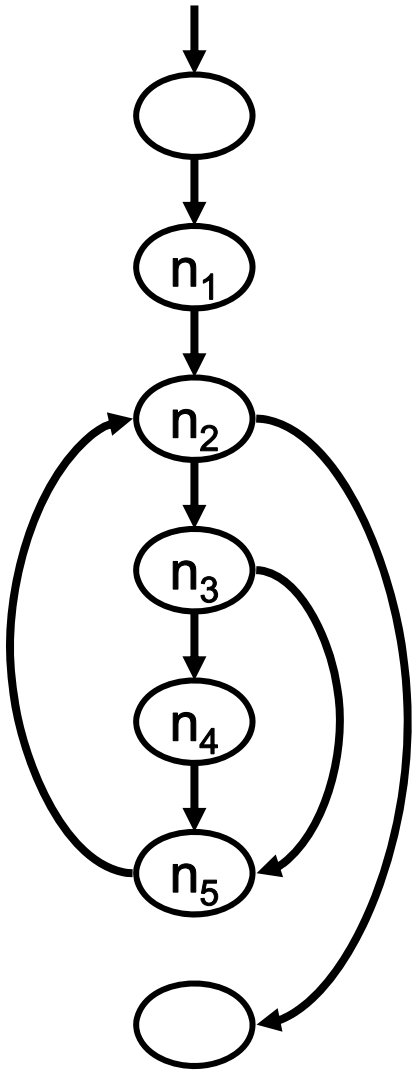
- Based on statements, paths, branches, conditions

Data flow-oriented techniques (not covered here)

- Based on dependencies between definition and use of data/variables

> *Advantage*
> Consideration of the structure of the implementation
>
> *Disadvantage*
> Not realized but specified functions will not be detected

Example of control flow




```
void countCharacter(int &vowelNumber,  int &totalNumber)
// precondition: vowelNumber <= totalNumber

{   char Character;
    cin>> Character;

    while ((Character >= 'A')&&(Character <= 'Z')&&
                (totalNumber < INT_MAX))
    {
          totalNumber = totalNumber +1;
         if((Character == 'A')||(Character == 'E')||
             (Character == 'I')||
             (Character == 'O')||(Character == 'U'))
         {
              vowelNumber = vowelNumber + 1;
         }

         cin>> Character;
    } //end while
}
```

Structural Testing

Types of structural testing

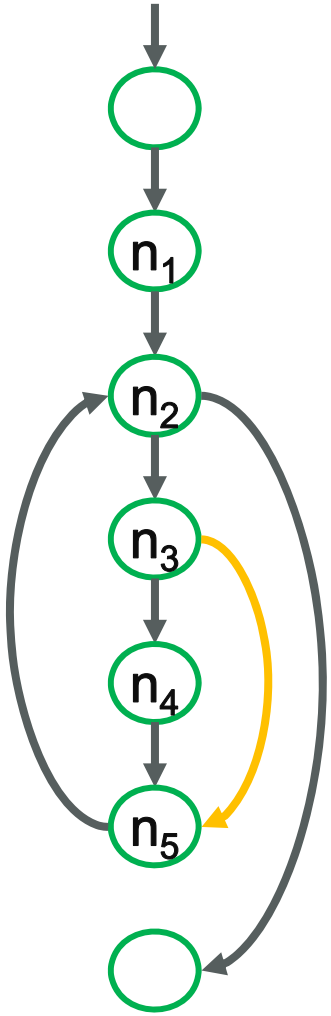
- **Control flow-oriented techniques**
  - **Statement coverage**
  - Branch coverage
  - Decision coverage

Statement coverage test

Simplest control flow-oriented test technique

- Notation: C_0
- Execution of each statement / node of the control flow graph at least once
- Coverage metric: relation of executed and total statements

Statement coverage test case



vowelNumber=0, totalNumber=0, input='A', '1'

```
void countCharacter(int &vowelNumber, int &totalNumber)
// precondition: vowelNumber <= totalNumber

{   char Character;
    cin>> Character;

    while ((Character >= 'A')&&(Character <= 'Z')&&
                (totalNumber < INT_MAX))
    {
          totalNumber = totalNumber +1;
         if((Character == 'A')||(Character == 'E')||
             (Character == 'I')||
             (Character == 'O')||(Character == 'U'))
         {
              vowelNumber = vowelNumber + 1;
         }

         cin>> Character;
    } //end while
}
```

Structural Testing

Types of structural testing

- **Control-flow oriented techniques**
  - Statement coverage
  - **Branch coverage**
  - Decision coverage

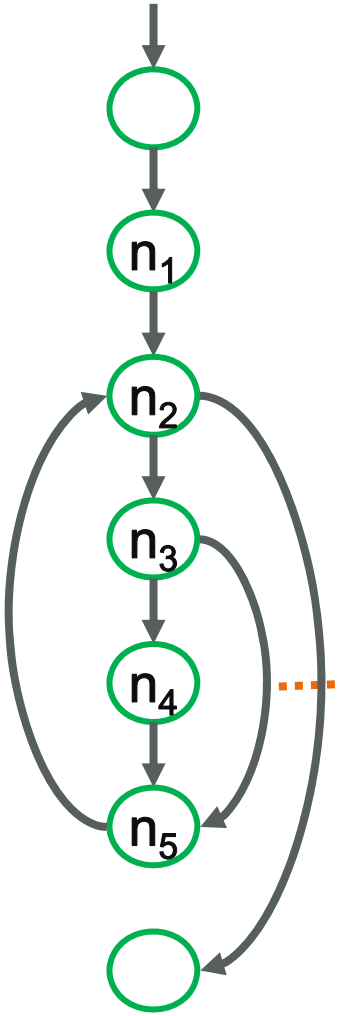
Branch coverage test

Coverage of every branch of the program

- Notation: C_1
- **Subsumes the statement coverage test**
- Widely accepted **minimum criterion** for structural testing

Branch coverage test case

vowelNumber=0, totalNumber=0, input='A', 'B', '1'




```
void countCharacter(int &vowelNumber, int &totalNumber)
// precondition: vowelNumber <= totalNumber

{   char Character;
    cin>> Character;

    while ((Character >= 'A')&&(Character <= 'Z')&&
                (totalNumber < INT_MAX))
    {
        totalNumber = totalNumber +1;
        if((Character == 'A')||(Character == 'E')||
            (Character == 'I')||
            (Character == 'O')||(Character == 'U'))
        {
            vowelNumber = vowelNumber + 1;
        }

        cin>> Character;
    } //end while
}
```

Structural Testing

Types of structural testing

- **Control-flow oriented techniques**
  - Statement coverage
  - Branch coverage
  - **Decision coverage**

Condition coverage test

Question: Is branch coverage useful for testing complex decisions?

Example:

- Simple decision:
  - if (x > 5)

- Complex decision:
  - if (credit card defect || credit card invalid || PIN wrong || Timeout)
  - if (((u == 0) || (x > 5)) && ((y < 6) || (z == 0)))
    [= if ((A||B)&&(C||D)]

MC/DC – modified condition / decision coverage

Each partial decision influences the complex decision

Pairwise test cases

- One partial decision changes
- The other partial decisions keep constant
- Complex decision should change
- Suitable if there are many complex decisions in your code

Effort of (n+1) test cases for n partial decisions

		A	B	C	D	(A‖B)&&(C‖D)
I	1	F	F	-	-	F
	2					
	3					
	4					
II	5	F	W	F	F	F
III	6	F	W	F	W	W
IV	7	F	W	W	-	W
	8					
V	9	W	-	F	F	F
	13					
VI	10	W	-	F	W	W
	14					
VII	12	W	-	W	-	W
	11					
	15					
	16					

Agenda

- Introduction
- Functional Testing
- Structural Testing
- **Unit Testing**
- System Testing
- Test Documentation

Unit Testing

- How to test?
  - Concentrate on black-box techniques, and use white-box techniques only if required
  - Black-box testing is faster and is easier
- Where to obtain data for the test cases?
  - From the unit specifications
- Unit testing is also called „developer testing“
- Sometimes there are no requirements for this stage to derive test cases
  - This is not considered good development style!
  - Information out of the code is then needed

Unit Testing

- When should unit testing start?
  - As soon as possible; developers should program and test nearly in parallel. A class can only be considered “ready” if it has been tested.
    - Write test cases as soon as the resp. (unit) requirements specification is ready
  - Some approaches even propose to write test cases before the resp. code is written (“Test-driven development”).
  - System testing cannot start, if unit testing has not been finished.
  - Unit testing should finish in parallel to development.
  - The less code you write between two test runs, the less you have to check in case on an error.

Unit Testing

Features of units

- Several different types of methods
- Visibilities
- Modalities

Unit Testing

- Testing method types
  - Just like in development, there is a „best“ order
    1. Contructors
    2. Get-Methods
    3. Is-Methods
    4. Set-Methods
    5. Iterator-Methods (next element)
    6. Calculation methods
    7. Other methods
    8. Destructors

Unit Testing

- Visibilities
  - We test from restrictive to non-restrictive
    1. Private (see next slide)
    2. Protected
    3. Public
- Modalities
  - Some methods require a certain order of calls (and/or a state) to work properly, e.g., stack-methods (modal methods)
  - Some methods do not (non-modal)
  - We test the correct order of calls first, then the incorrect order

Testing private methods

- Testing private methods is not indisputable

> If there is significant functionality that is hidden behind private or protected access, that might be a warning sign that there's another class in there struggling to get out. When push comes to shove, however, it's probably better to break encapsulation with working tested code that it is to have a good encapsulation of untested, non-working code.
>
> (Hunt, A., Thomas, D.: Pragmatic Unit Testing : In Java with JUnit. The Pragmatic Starter Kit - Volume II, Raleigh, The Pragmatic Programmers, 2004)

- If you have to test private methods, use
  - a nested test class
  - reflection

Agenda

- Introduction
- Functional Testing
- Structural Testing
- Unit Testing
- **System Testing**
- Test Documentation

System Testing

- Based on user interface and software requirements
  - No code as basis for tests
- Where to obtain data for the test cases?
  - Requirements documents, use cases, system functionalities, ...

System Testing

Order of test derivation for use cases

- For each use case, do
  - Execute the „regular“ processing (legal values, expected results)
  - For each step in a use case, do
    - Check, if there are exceptions to the regular case
      - What happens? → New test case
    - end
- End

- Compare with expected results (post condition) AND quality requirements

System Testing

- Not only functionality in focus

- Various specialized activities
  - Security Testing
  - Performance Testing
  - Usability Testing
  - Memory Management

  …

  What is important for your product?

System Testing

- When should system testing start?
  - Write test cases as soon as the requirements specification for your software (system) is ready.
    - This can be done as part of the requirements inspection.
  - Some of the test cases you have written do not require the whole system for executing
    - Those can be executed as soon as all relevant parts of the system are done (possible overlap with integration testing)
    - The others should be executed as soon as the system is completely developed

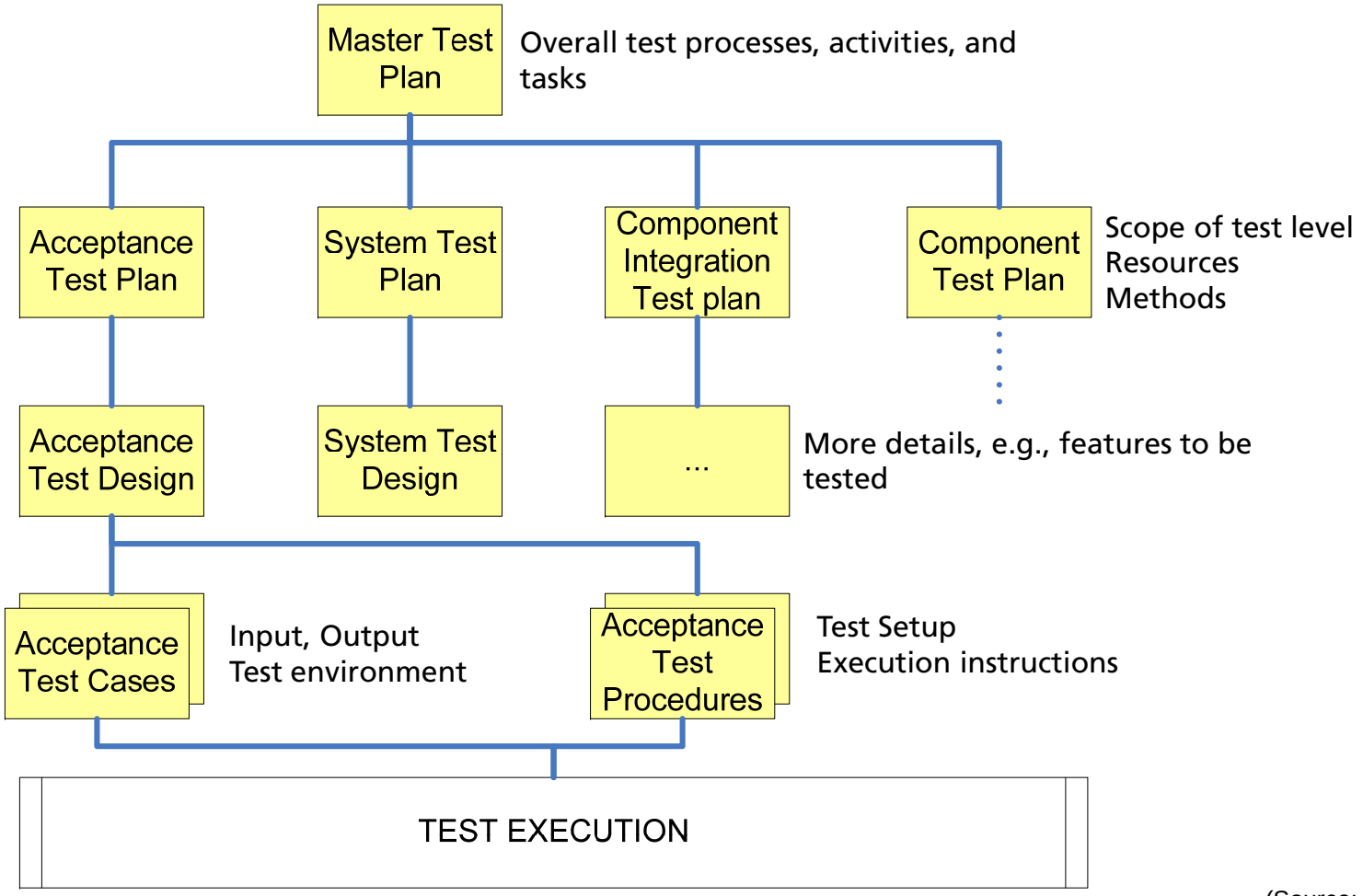
Agenda

- Introduction
- Functional Testing
- Structural Testing
- Unit Testing
- System Testing
- **Test Documentation**

Test documentation using IEEE 829

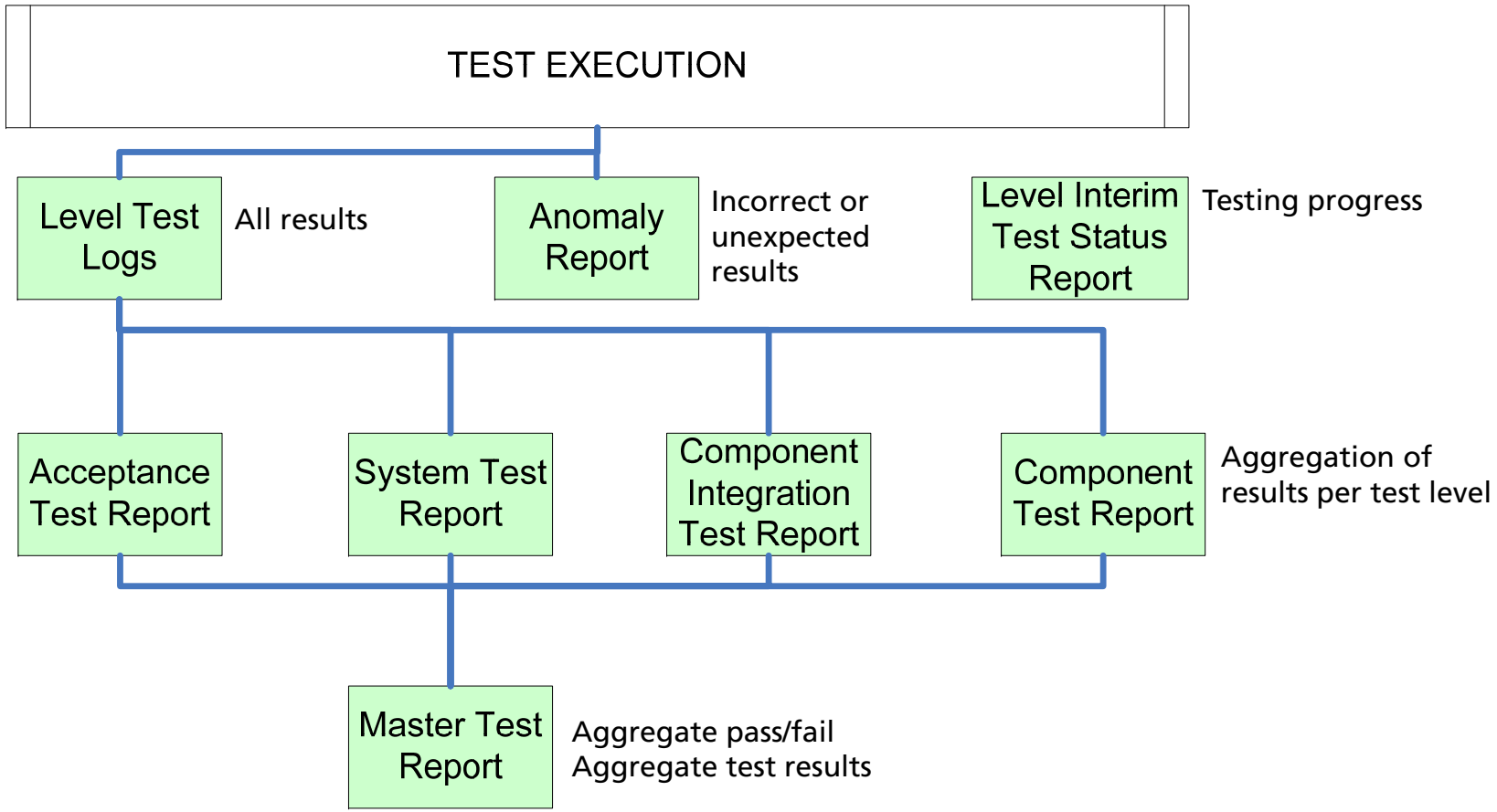
- IEEE 829: Test documentation provides documentation templates
  - Define test tasks, required inputs, and required outputs
  - **Requirements Traceability Matrix** (Requirements ←→ Test cases)
  - Document what you did and why
  - Document deviations from the plan
- Provides structure for the whole testing process
  - Breaks down overall plan into smaller parts
    - Here: testing is split up into unit testing and system testing
    - Separate documents for planning and test cases

IEEE Standard for Software and System Test Documentation



(Source: IEEE 829:2008)

IEEE Standard for Software and System Test Documentation



(Source: IEEE 829:2008)

Test Documentation

What do you need?

- You don´t need all of these documents (no need to comply to IEEE 829)
- Testing has to be reproducible and traceable (others have to be convinced)
- Use the Requirements Traceability Matrix
- Show how you structured your tests for breakdown
  (from system-wide to unit- (function-) wide)

- Use the same defect management as during inspections (*.xls?)
- Use some of the test report templates for structuring and presenting the results

- If information is already written down somewhere else, just reference it!
  (There is no need for filling pages – quality counts, not quantity)

Standards – IEEE 829

Example: Master Test Plan

- Main test planning and test management document
- Set objectives for each part
- Set the division of labor (time, resources) and interrelationships between the parts

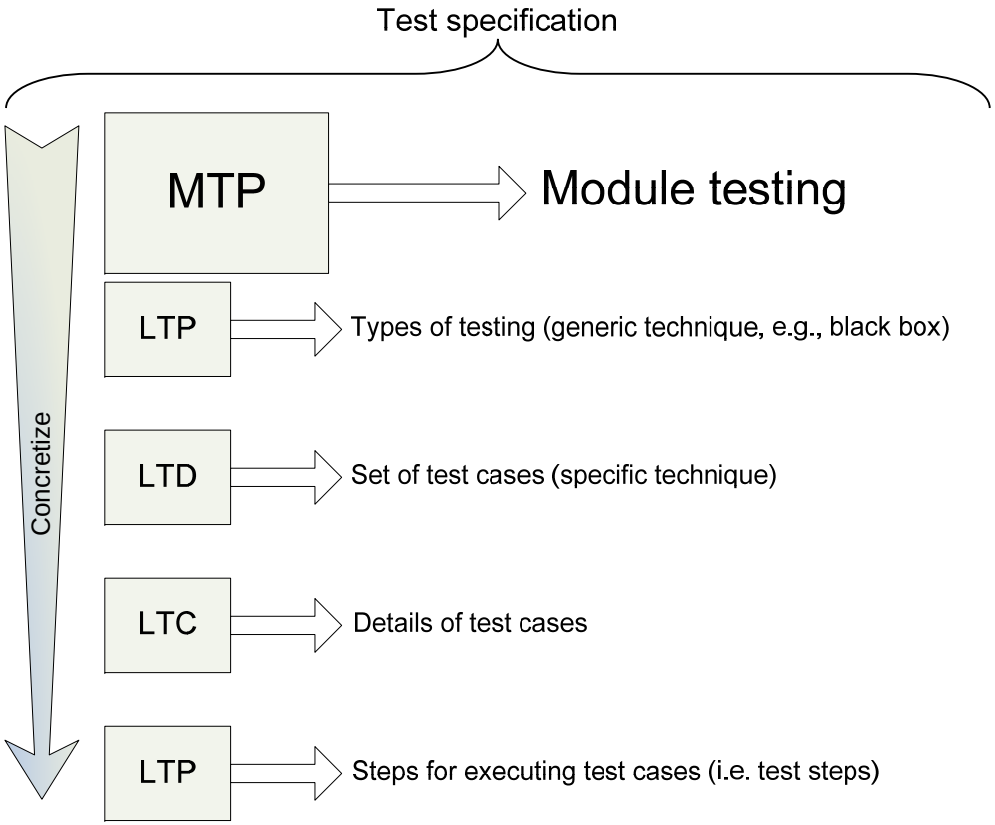
Identify

- Number of levels of test
- Overall tasks to be performed
- Documentation requirements

→ Appendix: more information

Standards – IEEE 829: Example

(The Master Test Plan of course also covers the other test levels)



Structure:

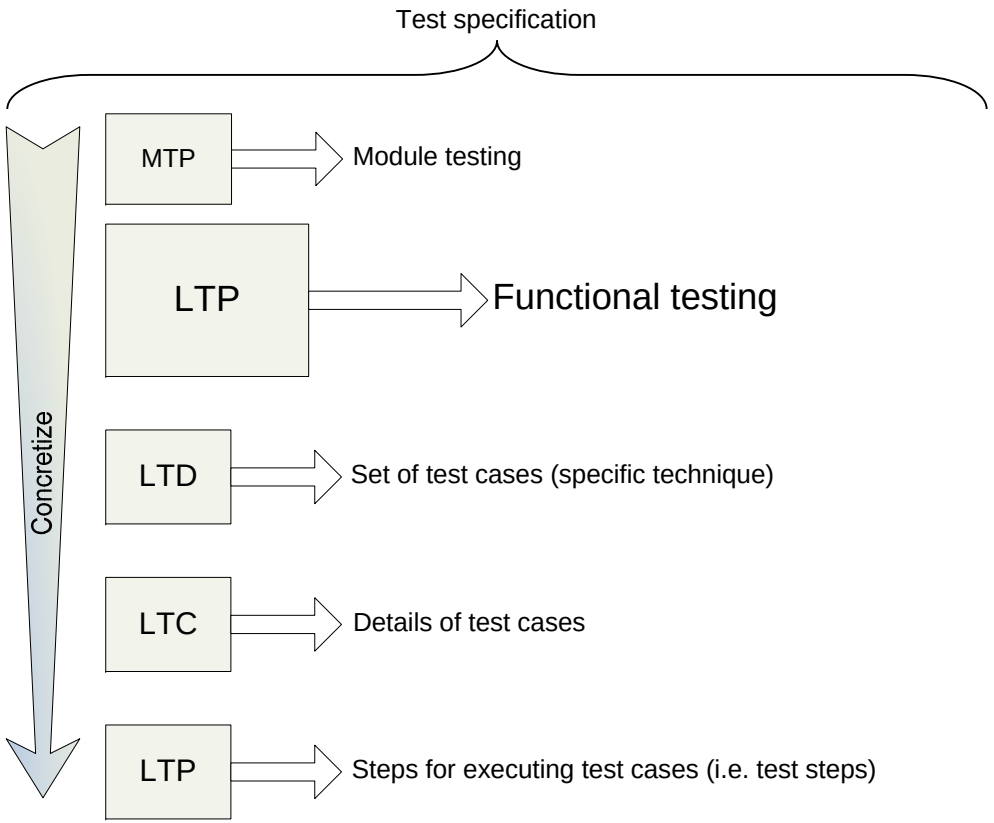
For each level of testing there is a specific LTP

For each type of testing there is a specific LTD

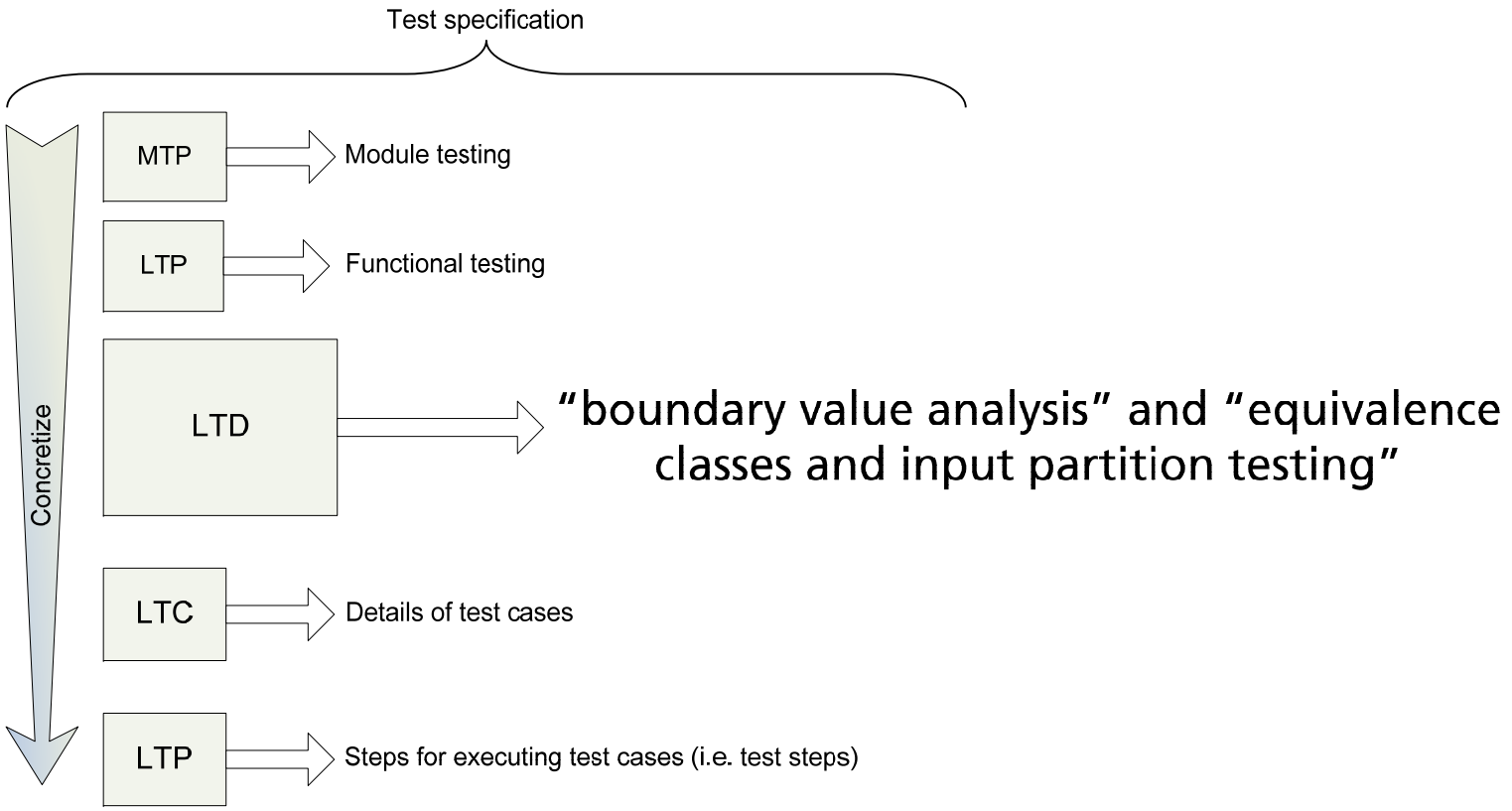
For each set of test cases there is a specific LTC

For each test case there are specific LTPs

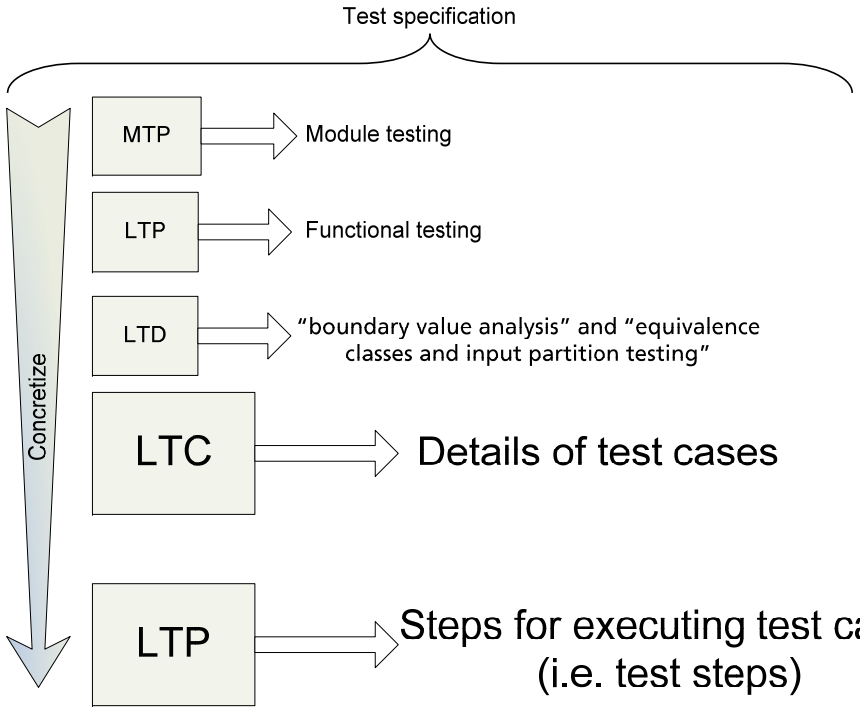
Standards – IEEE 829: Example



Standards – IEEE 829: Example



Standards – IEEE 829: Example



Variable	Valid equivalence classes	Invalid equivalence classes
Customer ID	1)P123 2)B135	3) D123
Invoice amount	4) 1 <= IA <= 999999	5) IA <= 0 6) IA > 999999
Receipt of payment	7) 1 <= RoP <= 999999	8) RoP <= 0 9) RoP > 999999
Routine	10) True (RoP >= IA)	11) False (RoP < IA)

Test case	Equivalence classes used	Customer ID	Invoice amount	Receipt of Payment	Routine
1	1, 4L, 7U, 10	"P123"	1	999999	True
2	2, 4U, 7L, 11	"B135"	999999	1	**False**
3	3	**"D123"**	1	1	-
4	5	"P123"	**0**	1	-
5	6	"P123"	**1000000**	1	-
6	8	"P123"	1	**0**	-
7	9	"P123"	1	**1000000**	-

Closing remarks

Some hints

- Test the performance and the memory management on the iPad
  (Have you tried to provoke memory warnings?)
- System testing should be done manually at least the first time a test case is executed
  - You can capture your test runs, and replay them later
    (e.g., when new functionality is added, code has changed, …)
- Test the most riskiest parts first
  You may not have enough time to test all you want

  Which parts of the implementation were the most challenging?
  Which parts of the implementation are the most important ones?

Literature

[1] Liggesmeyer, Peter: Software-Qualität. Testen, Analysieren und Verifizieren von Software, 2. Aufl. Heidelberg : Spektrum Akademischer Verlag, 2009. ISBN 978-3-8274-2056-5 (in German)

[2] Craig, Rick D. ; Jaskiel, Stefan P.: Systematic Software Testing Boston : Artech House, 2002. (Artech House Computing Library). - ISBN 1-58053-508-9

[3] IEEE Computer Society: IEEE 829: IEEE Standard for Software and System Test Documentation, New York, 2008

[4] Myers, Glenford J.: The Art of Software Testing New York : John Wiley & Sons, 1979. (Business Data Processing). - ISBN 0-471-04328-1

Good Luck!

Any Questions?

Fraunhofer
IESE

Dipl.-Wirtsch.-Inf. Alexander Klaus

Fraunhofer Institute for Experimental Software Engineering IESE

Fraunhofer-Platz 1 | 67663 Kaiserslautern
Phone +49 631 6800-2245 | Fax -9 2245

Alexander.Klaus@iese.fraunhofer.de
www.iese.fraunhofer.de