Software Quality Assurance

Chapter VI: Testing

Prof. Dr. Gregor Engels

FG Database and Information Systems

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Software Quality Assurance

- Constructive approaches
  - Languages
    - Syntax, Semantics
    - DSLs
    - DMM, Visual Contracts
  - Patterns / Anti-Patterns
  - Architectural styles
  - MDD
    - Transformation
    - Refinement
  - Product Lines
  - Packaged vs. Custom SW
  - Open Source Software
  - Product standards

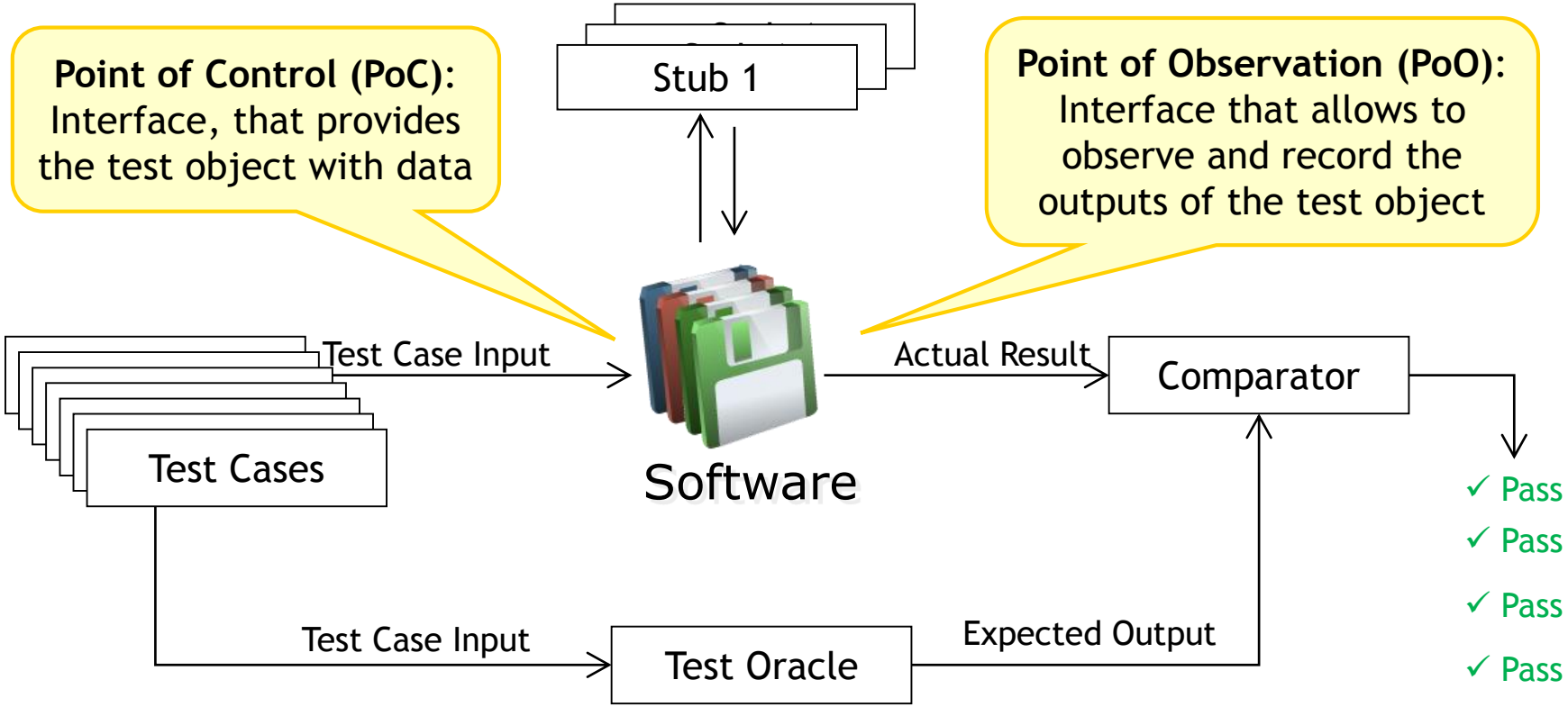
Software

- Process-based approaches
  - Software development processes
    - Classic
    - Agile methods
  - Standards
    - ISO 9000, CMM

- Analytic approaches
  - Static analysis
    - Reviews, Inspections
    - Model / Code Analysis
    - Model Checking
      - General properties
      - Specific properties, Patterns
  - Dynamic analysis
    - Testing
      - General test strategies
      - Test process/types of tests
      - Black box testing
      - White box testing

Large-Scale Example: SOA and SQA

Testing

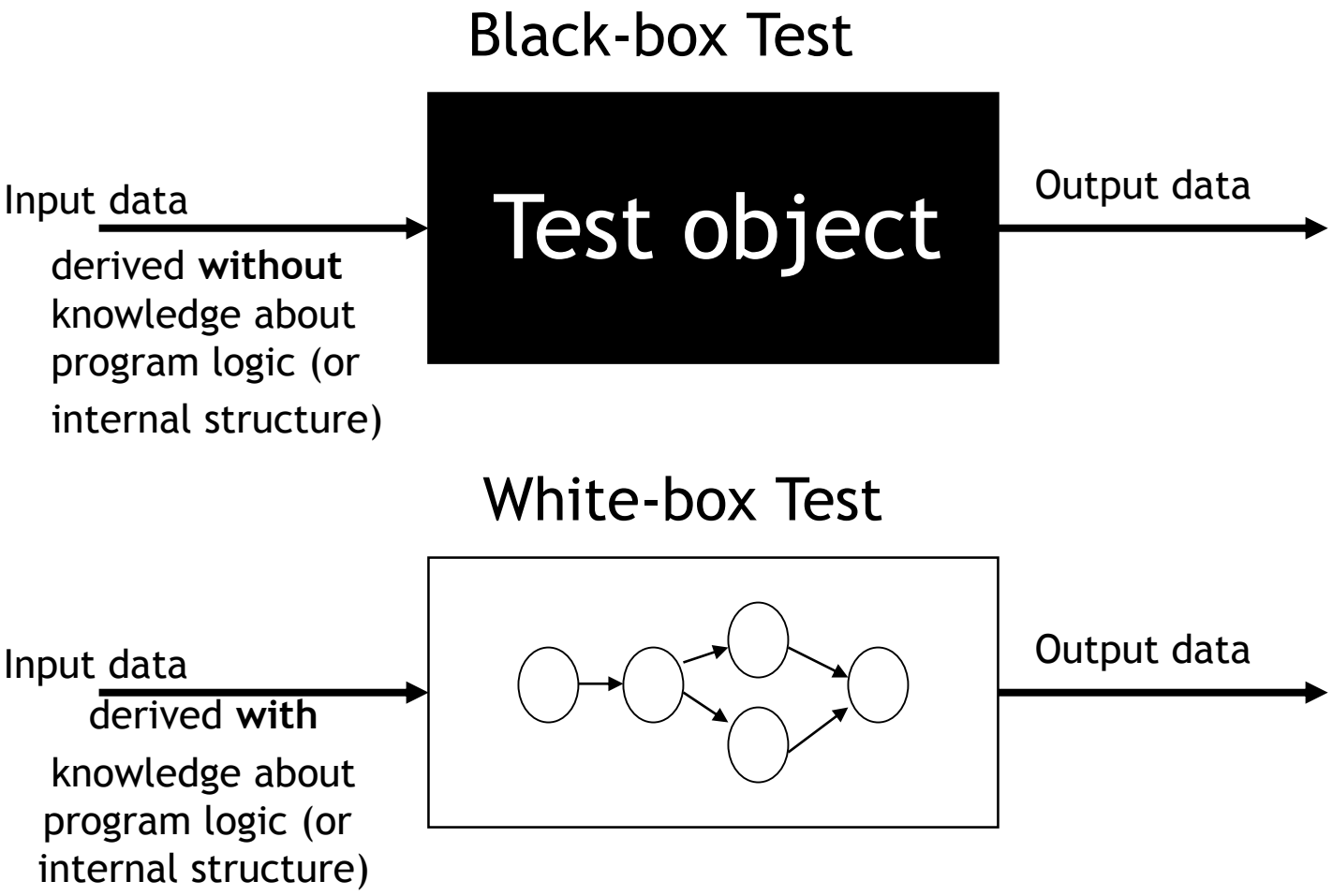


What do you do, if all test cases passes?
Do you trust in your software under test?

ISTQB Definitions: Test Driver/ Test Stub

- **Test Driver:** A software component or test tool that replaces a component that takes care of the control and/or the calling of a component or system.
- **Test Stub:** A skeletal or special-purpose implementation of a software component, used to develop or test a component that calls or is otherwise dependent on it. It replaces a called component.

Black-box Test vs. White-box Test (1)



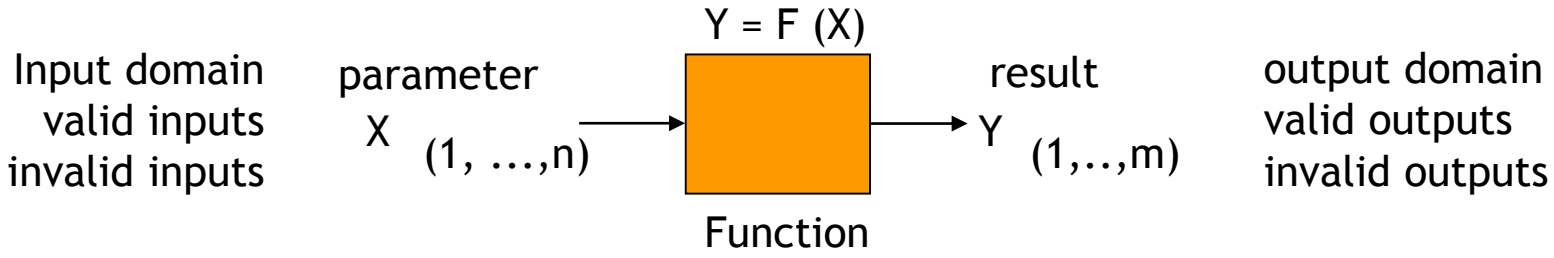
Black-box testing (test object is opaque)

- Black-box procedures in general
  - No information about the program code and the inner structure
  - The behavior of the test object is observed from outside (PoO - Point of Observation is outside of the test object)
  - Control of the test object only through the choice of input test data (PoC - Point of Control is outside of the test object)
- Specification-based approaches („functional“ test approaches)
  - Models and specifications for the test object, whether formal or informal, are the basis for the specification of the problem to be solved, the software or its component.
  - Systematic test cases can be derived from these models.
- Experience-based approaches
  - Use knowledge and experience of people to derive test cases.
  - Knowledge of testers, developers, users, domain experts, and other stakeholders about the software, its use and environment
  - Knowledge about likely errors and their distribution

White-box testing (test object is transparent)

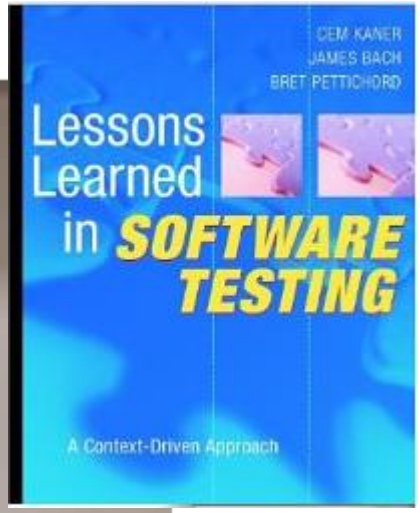
- Test cases can be derived based on the program structure of the test object (»structure-based« test approaches)
  - Information about the structure of the software is used to derive test cases, for example the code and the algorithm.
  - Degree of coverage of the code can be determined for the test cases.
  - Additional test cases to increase the degree of coverage can be derived systematically.
  - The inner order of events/execution steps of the test object can be analyzed while test object is executed (Point of Observation within the test object)
  - Direct manipulation of the execution of the test object possible, for example if the failure condition for a negative test cannot be triggered from the component's interface (Point of Control can be within the test object)

Black-Box Testing: Specification-based Approaches



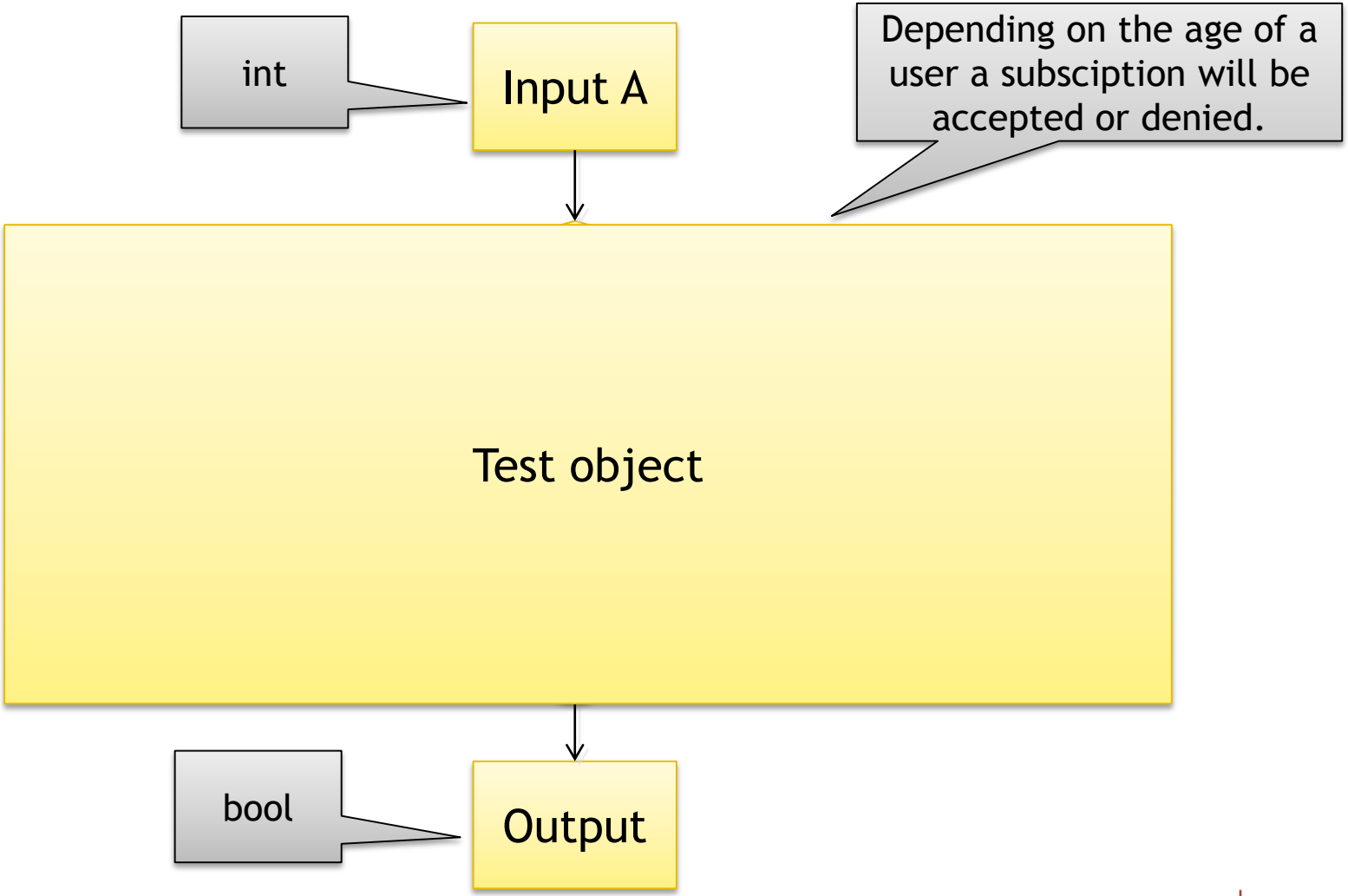
- Equivalence partitioning
  - representative inputs
  - valid inputs
  - invalid inputs
- Boundary value analysis
  - input/output domains
  - domain borders
- State-based test
  - complex (inner) states and transitions
- Use case-based test
  - scenarios of system usage

Do you know this guy?

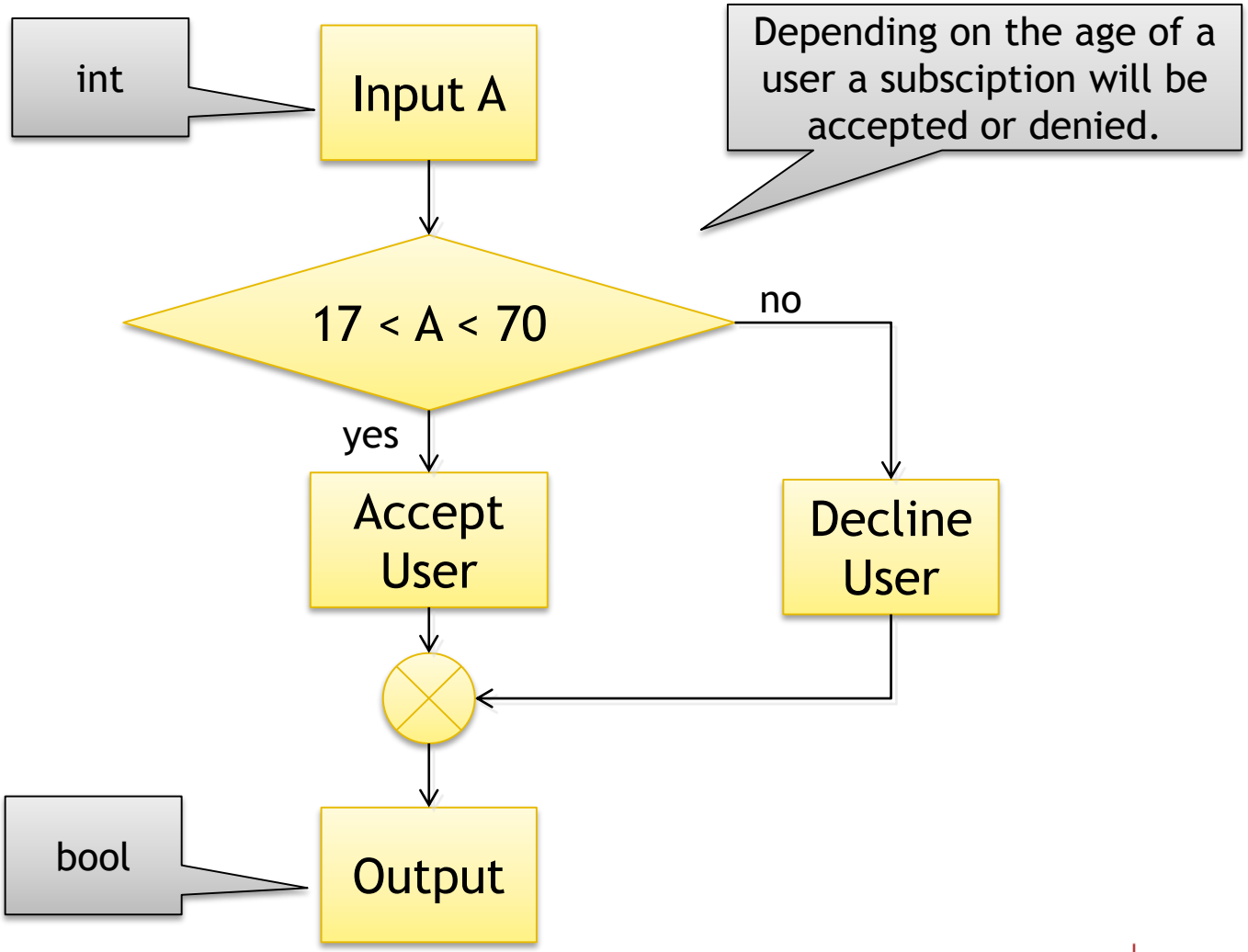


http://www.youtube.com/watch?v=ILkT_HV9DVU

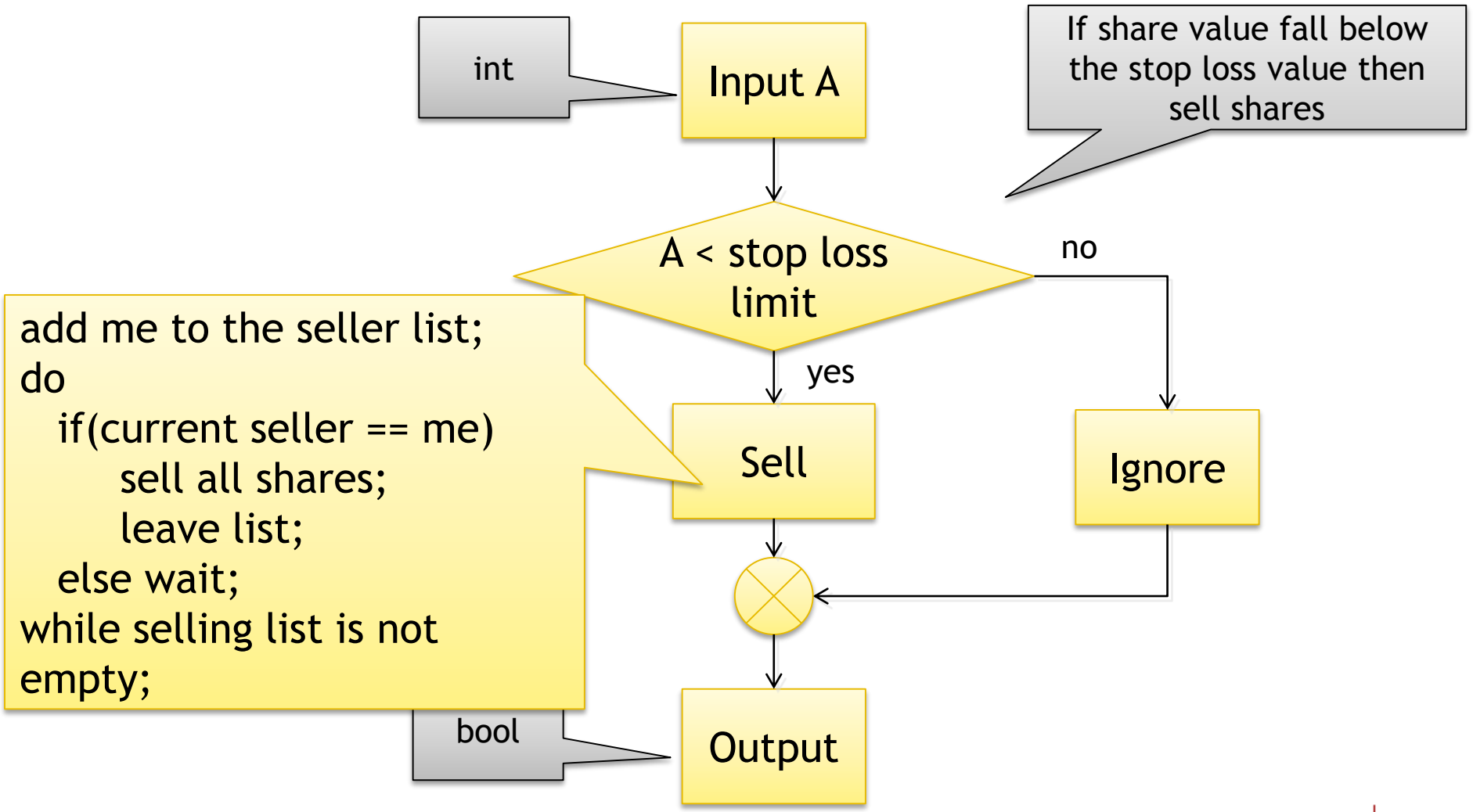
Let's test!


int
Input A
Depending on the age of a user a subsciption will be accepted or denied.
Test object
bool
Output

Let's test!

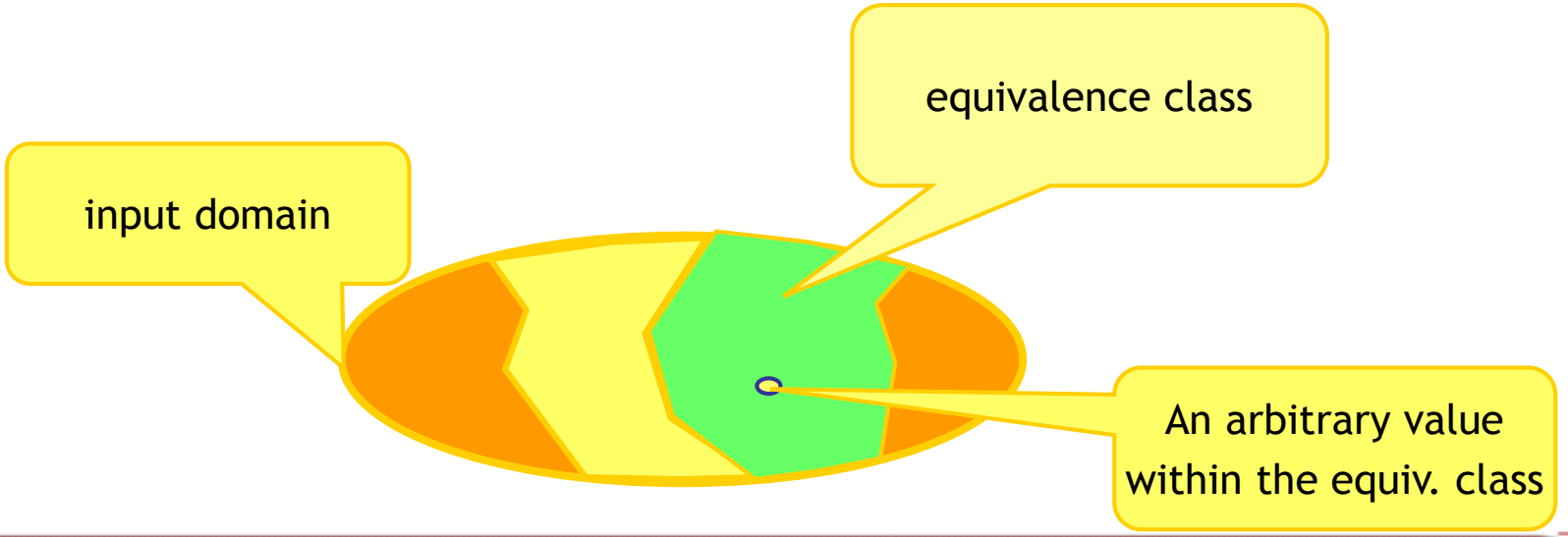

int
Input A
Depending on the age of a user a subsciption will be accepted or denied.
17 < A < 70
no
yes
Accept User
Decline User
bool
Output

Let's test!


int
Input A
If share value fall below the stop loss value then sell shares
A < stop loss limit
no
yes
add me to the seller list;
do
if(current seller == me)
sell all shares;
leave list;
else wait;
while selling list is not empty;
Sell
Ignore
bool
Output

Basic idea of equivalence partitioning

- The domains of the inputs and outputs are partitioned into equivalence classes, such that all values of a class result into an equivalent behavior of the test object.
  - If a value in an equivalence class reveals an error, it is expected that any other value within the same equivalence class reveals the same error.
  - If a value in an equivalence class reveals no error, it is expected that no other value within the same equivalence class reveals an error.



Equivalence Partitioning: Example: function "gcd"

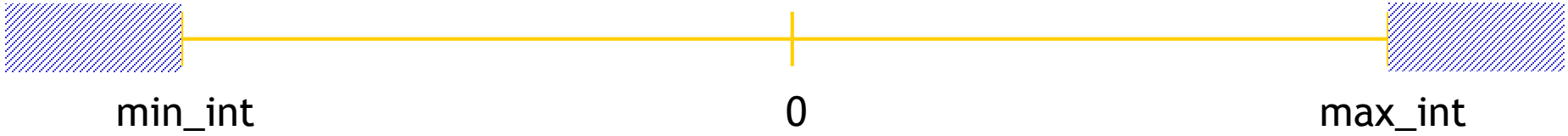
- Determines the *greatest common divisor*, *gcd* of two natural numbers m and n
  - gcd(4,8)=4; gcd(5,8)=1; gcd(15,35)=5
- Specification (mathematical)
  - gcd: $\mathbb{Z} \times \mathbb{Z} \nrightarrow \mathbb{N}$
    gcd (m,n) | m $\wedge$ gcd (m,n) | n $\wedge \forall o \in \mathbb{N}: o > gcd(m,n) \Rightarrow (\neg(o \mid m) \vee \neg(o \mid n))$
- Specification in UML / Java

```
public int gcd(int m, int n) {
// pre:      true
// post:     m@pre.mod(return) = 0 and
//           n@pre.mod(return) = 0 and
//           forall(i : int | i > return implies
//              (m@pre.mod(i) > 0 or n@pre.mod(i) > 0)
...
)
```

- Are both specifications equivalent?

Equivalence classes for gcd

- Domains of inputs and outputs
  - Input parameter: `int`
  - Output parameter: `int`
- Distinguish valid from invalid subsets of the domains



- General procedure:
  - Make equivalence class for each restriction
    here: one valid, two invalid (<min_int, >max_int)
  - If there is a specification for values that are treated differently
    → make different equivalence classes
  - If there is a situation that needs to be given
    → make equivalence class where situation is given and where not

Equivalence class building – more examples

- If there is a restriction in a domain
  - The specification of a test object expresses that natural numbers between 1 and 100 are valid inputs.
  - Input domain: 1 <= x <= 100
  - Equivalence class (valid) 1 <= x <= 100
  - Equivalence classes (invalid) x < 1 and x > 100

- If there is a set of values defined that are treated differently
  - Specification: in a sports club there are these disciplines
    - Soccer, hockey, baseball, volleyball, tennis
  - Equivalence classes (valid) soccer
hockey
baseball
volleyball
tennis
  - Equivalence class (invalid) every other discipline

Generation of test cases

- For each equivalence class (valid: vEC1, invalid iEC1)

	TC1	TC2	...	TCn
vEC1	x			
vEC2		x		
...			x	
iEC1				
iEC2				x
...				

- At least two equivalence classes per parameter
  - one with valid values
  - one with invalid values
- With n parameters and m_i equivalence classes (i=1..n) there are

$$\prod_{i=1..n} m_i \text{ different combinations (test cases)}$$

Further strategies for equivalence partitioning-based testing

- Sort test cases according to relative frequency (consider typical use)

- Do not combine representatives of invalid equivalence classes with other representatives of invalid equivalence classes
  - Example: input domain: 1 <= value <= 99; color IN (red, green, yellow)
  - Equivalence classes:

    value_vEC1: 1 <= value <= 99
    value_iEC1: value < 1
    value_iEC2: value > 99
    color_vEC1: color IN (red, green, yellow)
    color_iEC1: color NOT IN (red, green, yellow)
  - Test data: value=0, color=black => value_iEC1 and color_iEC1
  - Invalid treatment of color=black remains potentially unrevealed!

Equivalence classes for gcd

```
public int gcd(int m, int n)
```

Input parameter m: int

```
vEC1_1 : min_int ≤ m < 0
vEC1_2 : m = 0
vEC1_3 : 0 < m ≤ max_int
iEC1_1 : m < min_int
iEC1_2 : m > max_int
```

Input parameter n: int

```
vEC2_1 : min_int ≤ n < 0
vEC2_2 : n = 0
vEC2_3 : 0 < n ≤ max_int
iEC2_1 : n < min_int
iEC2_2 : n > max_int
```

Difficult to determine appropriate test data for output equivalence classes!

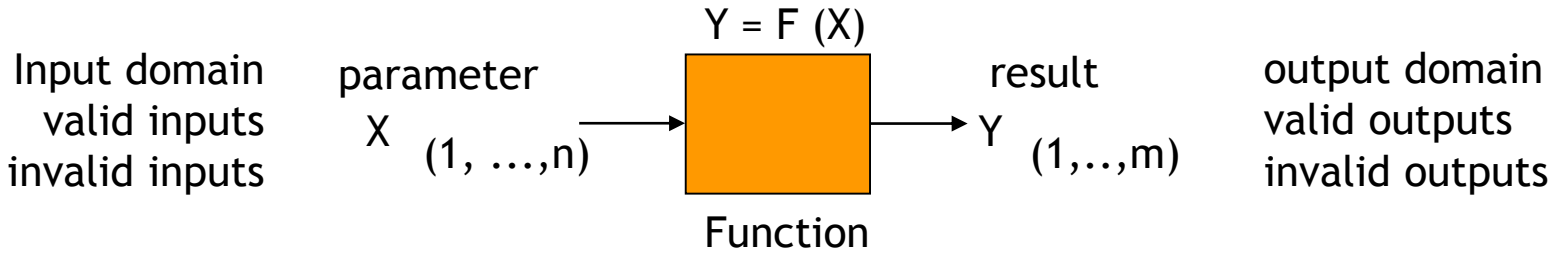
Output value gcd: int

```
vEC3_1 : 1 < gcd ≤ max_int
vEC3_2 : gcd = 1
iEC3_1 : gcd = 0
iEC3_2 : gcd < 0
iEC3_3 : gcd > max_int
```

Equivalence partitioning: advantages and disadvantages

- Advantages
  - Less test cases compared to unsystematic approaches
  - Appropriate for programs with many input and output conditions
- Disadvantages
  - Considers conditions for single input and output parameters, but interactions and dependencies between conditions are hard to consider
- Advice
  - Use to determine effective test data.
  - Combine with error-oriented approaches like boundary value analysis

Black-Box Testing: Specification-based Approaches



- Equivalence partitioning
  - representative inputs
  - valid inputs
  - invalid inputs
- Boundary value analysis
  - input/output domains
  - domain borders
- State-based test
  - complex (inner) states and transitions
- Use case-based test
  - scenarios of system usage

Boundary value analysis (1)

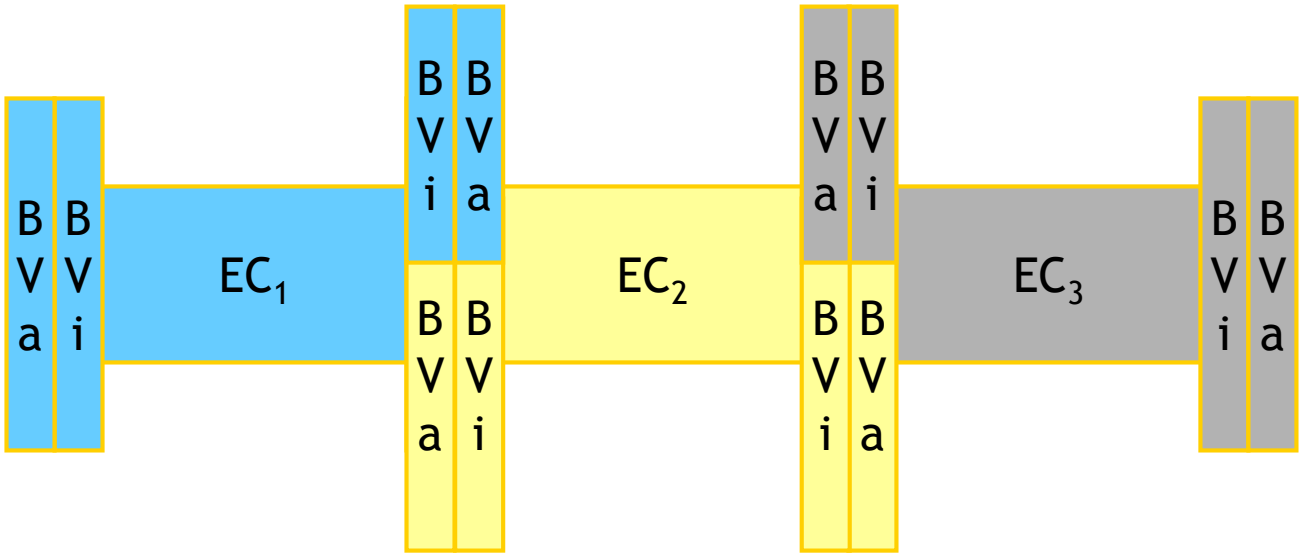
- Idea: In decisions and loops there are often boundary areas, for which a condition just applies (or just not any more).
  - Such decisions tend to contain errors (*off by one*).
  - Test data that checks such boundaries is more likely to reveal failures than other test data.
- Best results when combined with other approaches.
- In particular with equivalence partitioning:
  - Test borders of equivalence classes (smallest and biggest values).
  - Every border of an equivalence class must occur in at least one test case.

Boundary value analysis (2)

- Typically, the border value as well as the values directly above or below are tested.
- Atomic, ordered domains (integer, real, char)
  - Values at the borders of the domain
  - Values left or right of the borders (invalid values, smaller or bigger than the domain border)
- Set-based domains (e.g. limited number of input or output values)
  - Smallest and biggest valid number of elements
  - Smallest and biggest invalid number of elements

Boundary value analysis (3)

- Combination of boundary values of adjacent equivalence classes



EC – equivalence class
BV – boundary value:
i – within the EC
a – outside of EC

Boundary value analysis: Example

data type	boundaries	larger	smaller
integer	0 min_int max_int	1 min_int + 1 max_int + 1	-1 min_int - 1 max_int - 1
char[5]	"" "xxxxx"	"x" "xxxxxx"	`null` (if possible) "xxxx"
double	0.0e0 min_double (-∞) max_double (+∞) NaN (not a number)	ε min_double + ε max_double + ε ??	-ε min_double - ε max_double - ε ??

Boundary value analysis: hints (1)

- Consider first and last element of ordered sets
  - e. g. sequential file, linear list, table, …
- Test empty sets for complex data structures
  - e. g. empty list, null matrix, …
- For numerical calculations
  - choose values that are either close together or far away from each other

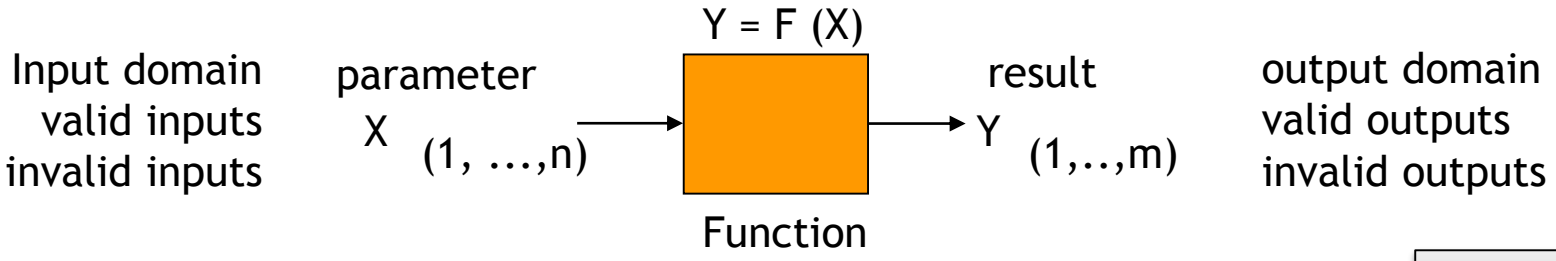
Boundary value analysis: advantages and disadvantages

- Advantages
  - At the boundaries of equivalence classes errors are more likely to be found
  - “Boundary analysis is, if appropriately applied, one of the most useful approaches for test case generation.”

    Myers, Glenford J.:
    Methodisches Testen von Programmen
    Oldenbourg, 2001 (7. Auflage)
  - Can be efficiently combined with other approaches that allow some freedom in the choice of test candidates.

- Disadvantages
  - Hard to provide a “recipe” for the choice of test data
  - Determination of relevant boundary values is sometimes difficult
  - Creativity is needed to find good test data
  - Often not efficiently applied, because it seems to be too simple.

Black-Box Testing: Specification-based Approaches



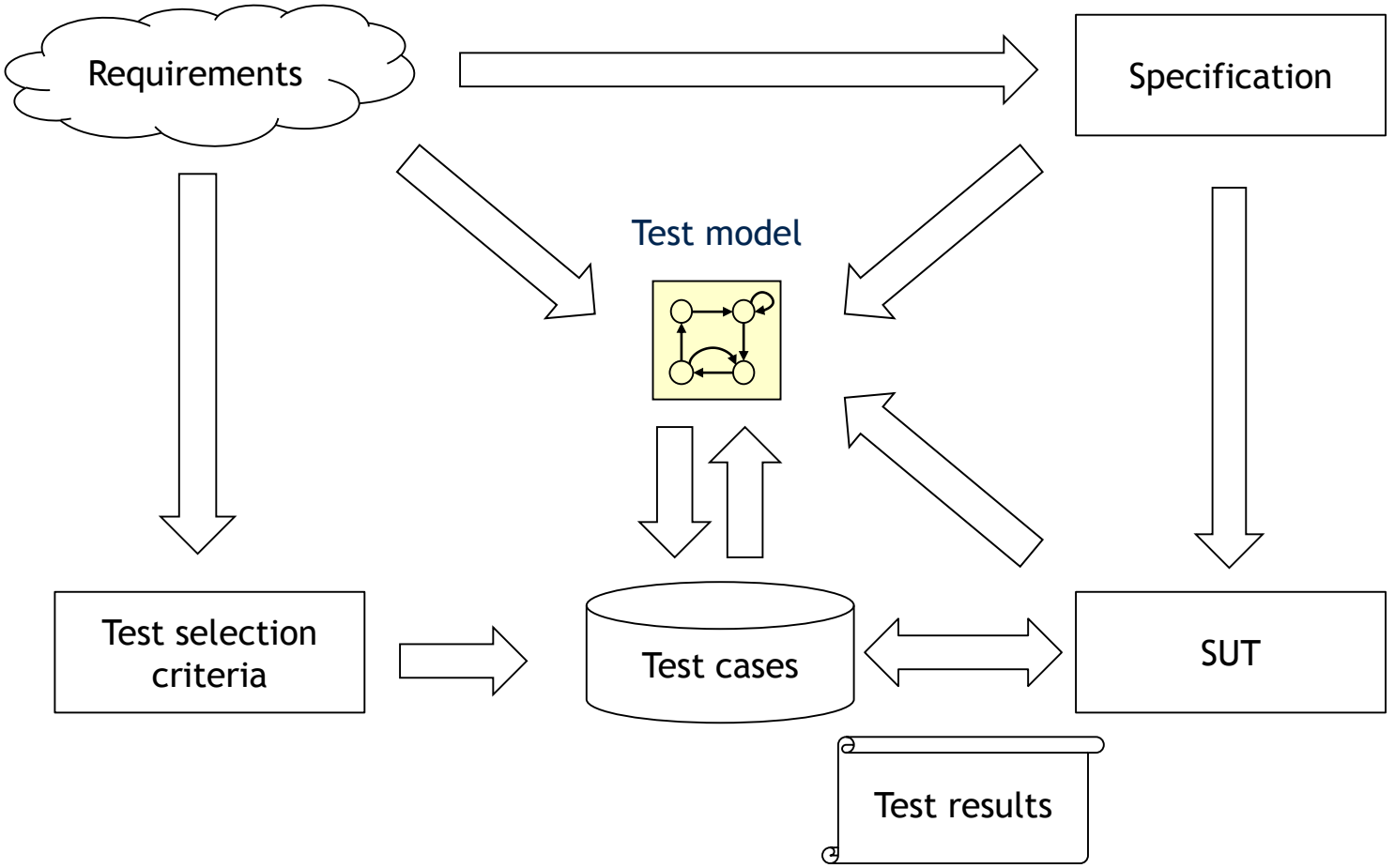
- Equivalence partitioning
  - representative inputs
  - valid inputs
  - invalid inputs
- Boundary value analysis
  - input/output domains
  - domain borders

- State-based test
  - complex (inner) states and transitions

Model-based Testing

- Use case-based test
  - scenarios of system usage

Model-based Testing: Different Scenarios



[Pretschner, A., Philips, J.: Methodological Issues in Model-Based Testing. 2005]

State-based tests

- With many software systems, the expected reaction to an event depends on the current state of the system, i.e., the inputs of the system in the past
- Behavior of such systems can e.g. be modeled with UML State Machines
  - Start state is initial state of system
  - State transitions are caused by events, e.g. method calls
- Idea of state-based tests: Check conformance of State Machine and actual system
  - Precise definition of State Machine‘s behavior needed

Example: Stack

Class Stack

State preserving operations

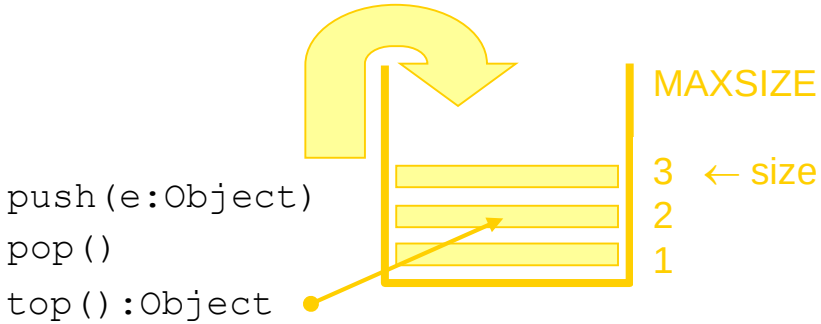
```
size():integer; // # of elements in stack
MAX():integer;  // maximum # of elements
top():Object;   // pointer to top element
```

State changing operations

```
Stack(Max:integer);     // Constructor
~Stack();               // Destructor
push(element:Object);   // adds element to stack
pop();                  // removes top element
```

Three states:

```
empty:  size() = 0;
filled: 0 < size() < MAX();
full:   size() = MAX();
```



Goals of state-based tests

- Show that system behaves conform to State Machine
  - Expected events
- Show that system does behave robust
  - Unexpected events

Steps of state-based testing

0. Analysis of state machine
1. Compute transition tree
2. Extend transition tree for robustness check
3. Generate sequences of events (including parameters carried by event, if needed)
4. Perform tests, measure coverage of transition tree

0. Analysis of State Machine

Three states:

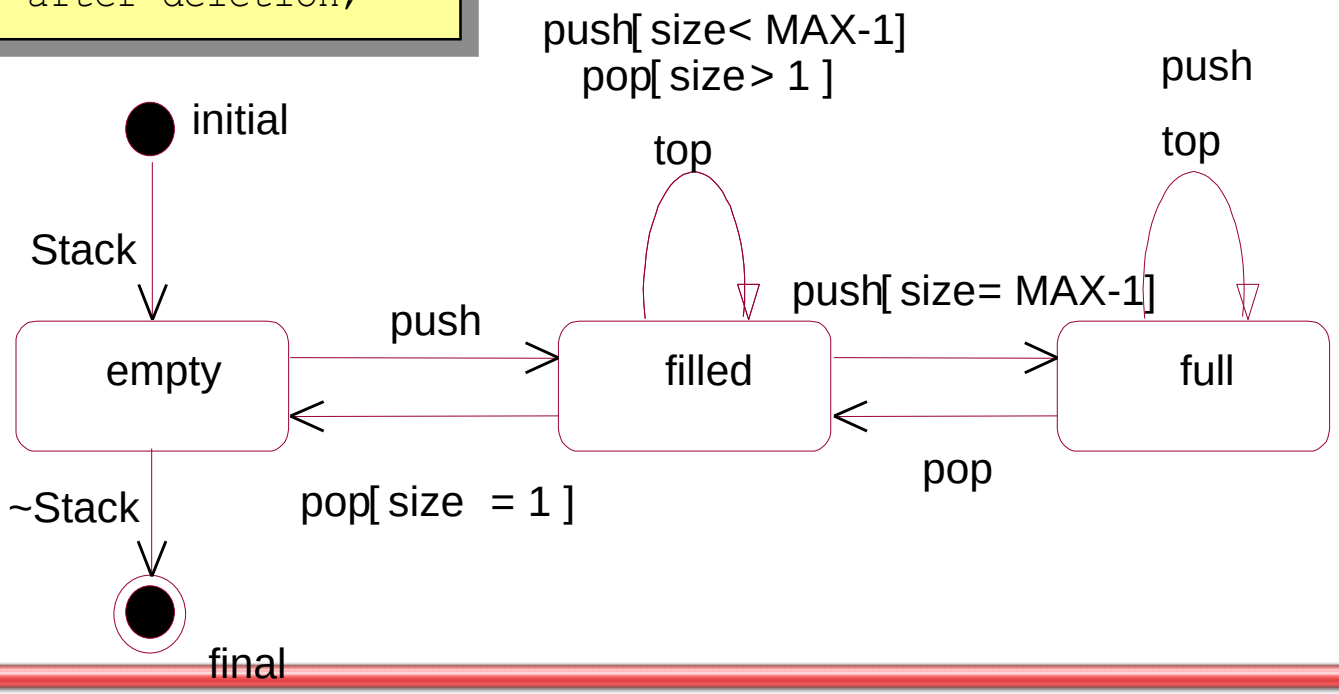
```
empty:   size() = 0;
filled:  0 < size() < MAX();
full:    size() = MAX();
```

Two „pseudo states“:

```
initial: before creation;
final:   after deletion;
```

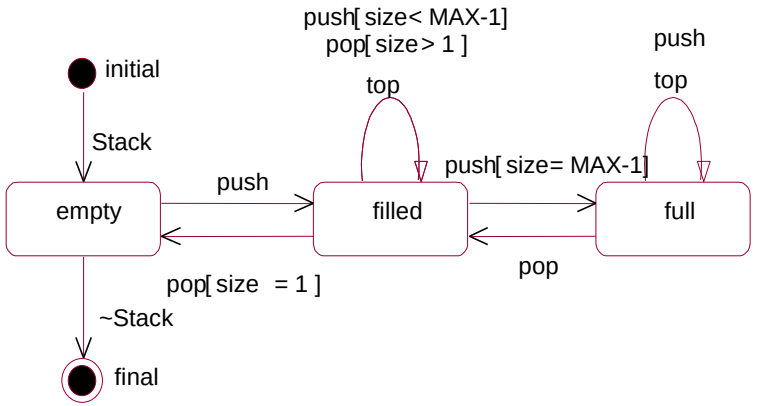
Eight state transitions:

```
initial → empty; empty → final
empty → filled; filled → empty (cycle!)
filled → full; full → filled   (cycle!)
filled → filled; full → full   (cycle!)
```



0. Check on completeness

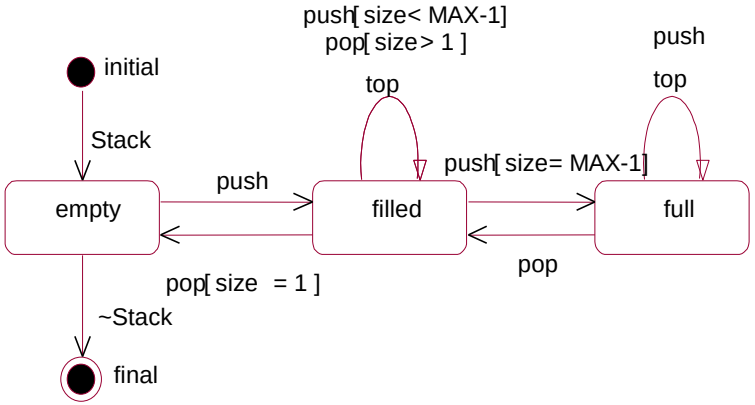
- Check whether state machine is complete
  - Use state transition table
  - Check guards w.r.t. completeness and consistency
  - Discuss non-specified State/Event pairs

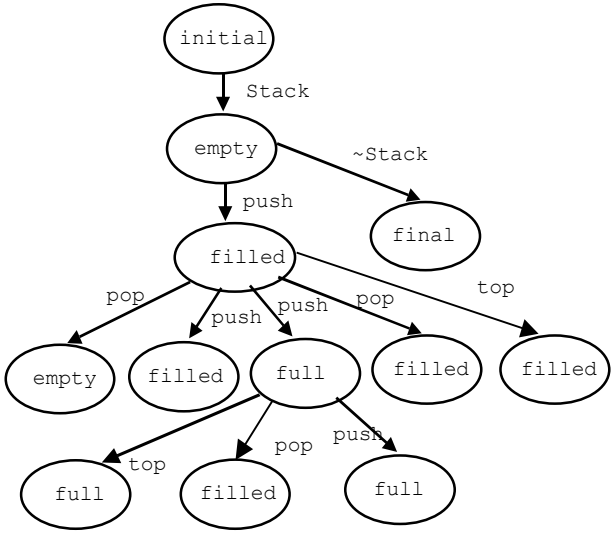


State / Event	initial	empty	filled	full
Stack()	empty	N/A	N/A	N/A
~Stack()	N/A	final	?	?
push()	N/A	filled	filled, full	full
pop()	N/A	?	empty, filled	filled
top()	N/A	?	filled	full

1. Computation of transition tree

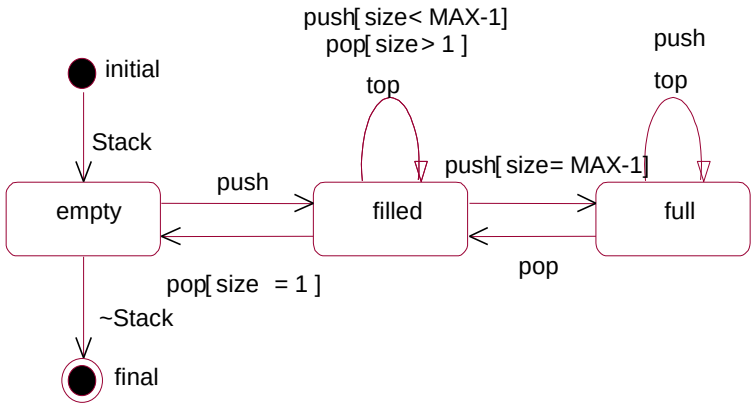
1. Initial state becomes root of tree
2. For all transitions from root to another state:
   Add that transition and the reached state to tree (label transition with event name)
3. Repeat last step with every leaf of the current tree until one of the following conditions holds:
   - Upper part of tree contains state which is equivalent to the leaf
   - The leaf is a final state

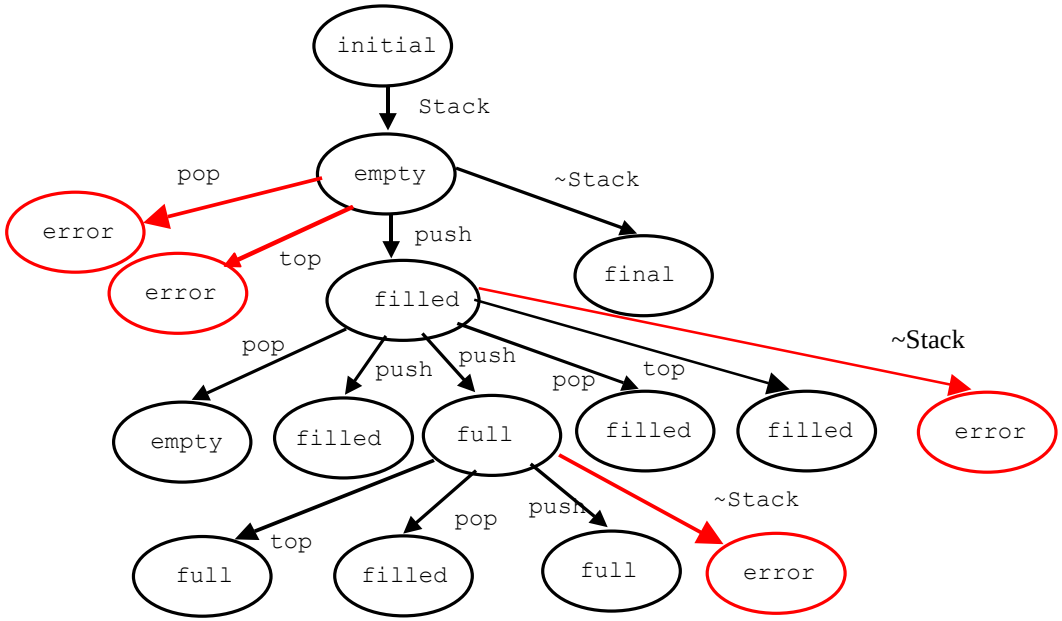




2. Extend transition tree for robustness check

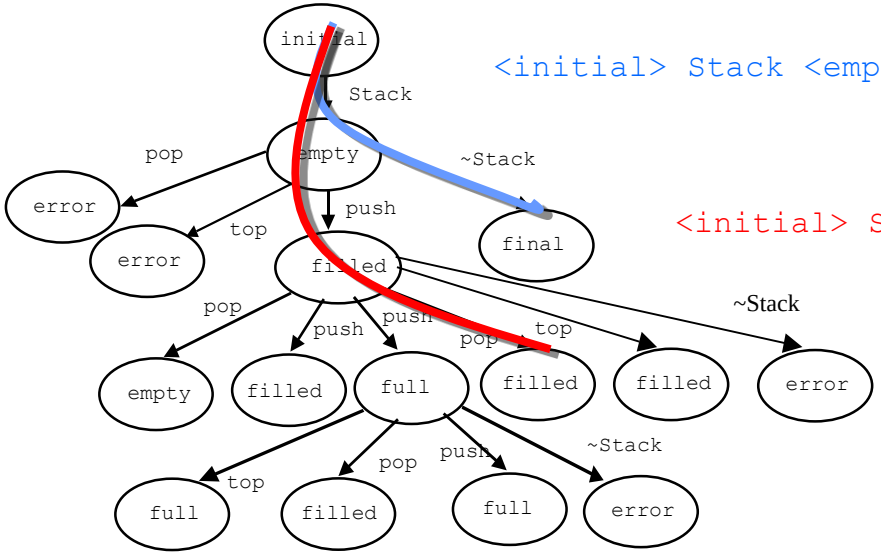
- Goal: Test robustness under sequences of events which *do not* conform to State Machine
- Introduce error state which is reached through transitions which are not in State Machine (see transition table!)





3. Generate test cases (1)

- Paths from root to leaves are sequences of events
- Stimulation of test object with paths covers all states and transition of State Machine
- Parameters might have to be created!



<initial> Stack <empty> ~Stack <final>

<initial> Stack <empty> push <filled> pop <filled>

3. Generate test cases (2)

State conformance test:

```
C1 = <initial> new Stack() <empty> ~Stack() <final>
C2 = <initial> new Stack() <empty> push() <filled> pop() <empty>
C3 = <initial> new Stack() <empty> push() <filled> push() <filled>
C4 = <initial> new Stack() <empty> push() <filled> pop() <filled>
...
C8 = <initial> new Stack() <empty> push() <filled> push() <full> push() <full>
```

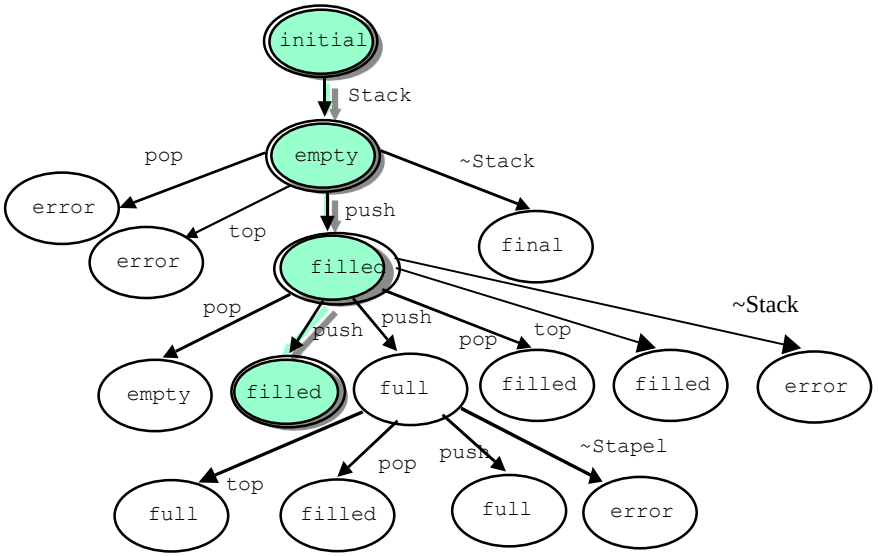
State robustness test:

```
R1 = <initial> new Stack() <empty> pop() <error>
R2 = <initial> new Stack() <empty> top() < error >
R3 = <initial> new Stack() <empty> push() <filled> ~Stack() < error >
R4 = <initial> new Stack() <empty> push() <filled> push() <full> ~Stack() < error >
```

4. Execute tests

- Test cases, i.e. sequences of events are encapsulated in test scripts, which are then executed by a test driver
- During execution, states are monitored by using state preserving operations

```
C3' = //<initial>
        Stack OUT = new Stack(5)
      //<empty>
        OUT.push(new Object())
      //<filled>
        OUT.push(new Object())
      //<filled>
        if (OUT.size() != 2) then
         throw WrongStateException;
```



State based tests: state coverage

- State explosion: state space of a software system is typically infinite => not all paths can be tested
- Minimal criterium: Every state of the state machine at least once

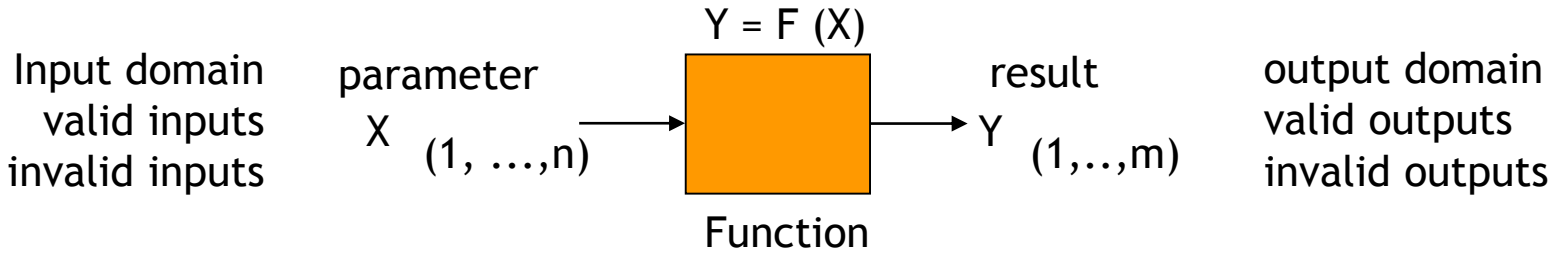
State coverage = # of states tested / # of all states

- Other criteria:
  - Every transition within the state machine at least once
  - All non-conforming transitions have been triggered
  - Every kind of event (i.e., every operation) is executed at least once
- In case of critical applications, test
  - all transitions with all states, also several times
  - if possible: All paths through transition tree

Evaluation of state-based testing

- Often the state depends on a number of variables
  - State space very complex
  - Generating test cases can be very time-consuming
- Use state-based tests where the functionality depends on the current state of the SUT
  - The testing approaches we have seen before do not respect the system states
- State-based tests are especially suited für object-oriented systems!
  - Objects typically have different states, behavior of their operations often depends on that state
  - This aspect of OO systems is respected by state-based tests

Black-Box Testing: Specification-based Approaches



- Equivalence partitioning
  - representative inputs
  - valid inputs
  - invalid inputs
- Boundary value analysis
  - input/output domains
  - domain borders
- State-based test
  - complex (inner) states and transitions
- Use case-based test
  - scenarios of system usage

Use case tests

- Use cases describe the interaction between actors and system
- Every use case has preconditions to be performed succesfully and postconditions which hold after it has been performed
- Additionally, a use case typically has a main scenario and several alternative scenarios (e.g. to handle exceptions)
- Therefore, use case are predetermined for deriving test cases which can be used to identify flaws in the system‘s processes
- Additionally, tests based on example scenarios – which have been identified in conjunction with dedicated users of system – are very helpful when designing acceptance tests

Use case Disbursement **for model** CashMashine
Actors Customer, main frame
Precondition card reader ready **AND** keyboard locked
Main scenario

1. Customer logs into system (includes UC „Login“)
2. Custome chooses „Disbursement“
3. Customer enters amount
4. CashMashine checks amount, reports to main frame
5. CashMashine updates card of customer
6. Customer takes card
7. CashMashine provides money
8. Customer takes money

Alternative scenario

4.a Amount to high, customer must enter new amount
4.b Amount can‘t be provided, customer must enter new amount

Postcondition Account of customer reduced by amount
AND Money in CashMashine reduced by amount
AND card reader ready **AND** keyboard locked

Exceptional scenario

6.a Card is not taken within 60 seconds

Postcondition CashMashine has confiscated card **AND** card reader ready **AND** keyboard locked

Exceptional scenario

1.-4. Customer cancels process
5.a CashMashine updates card of customer
6.a Customer takes card

Postcondition card reader ready **AND** keyboard locked
END Disbursement.

Designing use case tests

- Choose test cases such that the required coverage of use case is given
  - Main scenario
  - Alternative scenarios
  - Exceptional scenarios
  - Possible repetitions of scenarios

Other Approaches: Random test and smoke test

- Random test
  - Test cases are created by randomly choosing input values
  - If a certain distribution is given or assumed for the input (e.g. gaussian distribution), it should be respected when calculating input values (makes tests more realistic)
- Smoke test
  - Primary goal: Test that system runs at all („try out“)
  - No test oracle => no expected results
  - Smoke test tries to „crash system“
  - Often used for testing readily installed systems

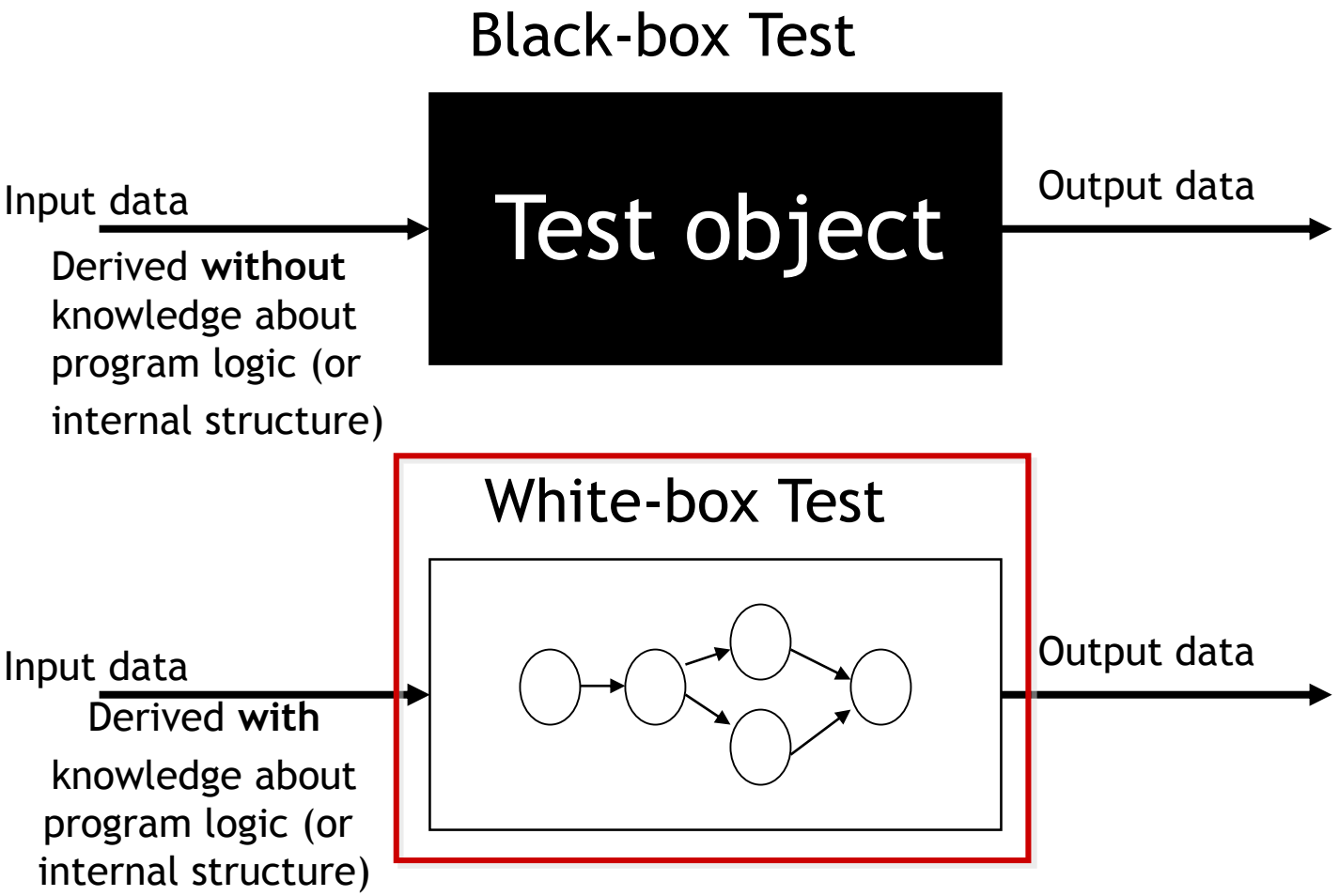
Black box testing: Discussion

- System specification is foundation of all black box tests
  - Errors in specification can not be found!
  - Succesfully tested system behaves as specification requires, even if this is not desired
- Additional functionality not covered
  - Functionality not contained in specification => functionality not tested
  - Coverage criteria only based on specification
- Main point of black-box approaches is testing of the system‘s functionality
  - Since this is the most important aspect of a system, black-box tests are very important in basically all kinds of testing efforts

Summary

- Dynamic tests *execute* the test object
- Special testing framework needed
- Not everything can be tested => test cases have to be chosen carefully
- Black-box do not require internal knowledge of system for choosing test cases
- Functional tests are based on specification of system
- Use equivalence classes in combination with boundary values to create test cases
- If state of system is important: Use state-based tests
- Use use case tests to test scenarios of usage

Black-box Test vs. White-box Test



White-box Testing – Terminology

- White-box Test
  - Testing based on an analysis of the internal structure of the component or system.
- White-box test design technique
  - Procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system.
  - Aka glass box testing, structure-based testing, structural testing, logic-driven testing, code-based testing

Structural testing of programs

- Based on control flow
  - Statement coverage (C0 coverage, all nodes of control flow graph)
  - Branch coverage (C1 coverage, all outgoing edges of every node in the control flow graph)
  - Branch condition coverage (branch coverage with additional checking of each subcondition with true and false values)
    - Example: … if (x>0 or a[x]=20) then …)
      - Test case x=0 and x=1 will not detect error in case of array a[1..10] and a[x]=20
    - Refinements: branch condition combination coverage, modified branch condition decision coverage
  - Path coverage (C∞ coverage, all paths through control flow graph)
- Based on data flow (not in this lecture)
  - All definitions (all defs)
  - All definition-use pairs (all def-uses)

Control flow based white-box testing

- Statement coverage
- Branch coverage
- Path coverage

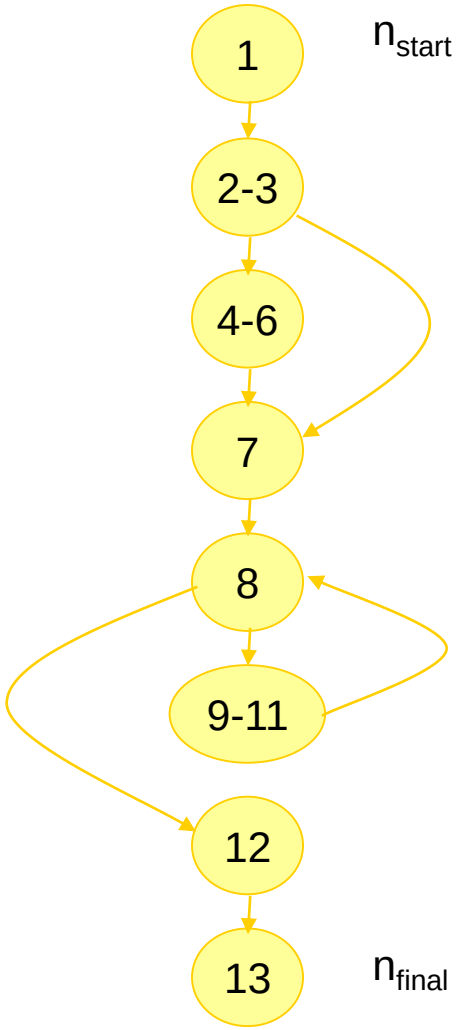
Control flow graph of gcd(int, int)

```
1. public int gcd (int m, int n) {
   // pre: m > 0 and n > 0
   // post: return > 0 and
   // m@pre.mod(return) = 0 and
   //...
2.     int aux, rem;
3.     if (n > m) {
4.         aux = m;
5.         m = n;
6.         n = aux;
       }
7.     r = m mod n;
8.     while (rem != 0) {
9.         m = n;
10.        n = rem;
11.        rem = m mod n;
       }
12.    return n;
13. }
```

Block

Block

Block



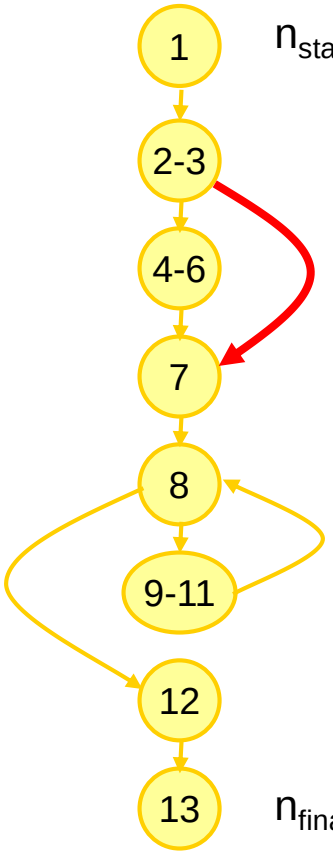
Statement coverage

- 100% statement coverage requires that every statement is executed at least once

$$\text{Statement coverage} = \frac{\text{\# of executed statements}}{\text{\# of all statements}}$$

- Every test case determines a path through the control flow graph
  - Statements are executed in order determined by nodes on path
- Calculation of coverage only considers *whether* a statement has been executed, not *how often*
- For every test case pre- and postconditions, expected results, and behavior have to be determined in advance
- Test ends when the targeted degree of coverage is reached

Example: Statement coverage for gcd()




```
1.  public int gcd(int m, int n) {
    // pre: m > 0 and n > 0
    // post: return > 0 and
    // m@pre.mod(return) = 0 and
    //...
2.      int aux, rem;
3.      if (n > m) {
4.          aux = m;
5.          m = n;
6.          n = aux;
        }
7.      rem = m mod n;
8.      while (rem != 0) {
9.          m = n;
10.         n = rem;
11.         rem = m mod n;
        }
12.     return n;
13. }
```

Test data = (2,3)

One path is sufficient to fulfill statement coverage: (1, 2-3, 4-6, 7, 8, 9-11, 8, 12, 13)

Statement coverage: Discussion

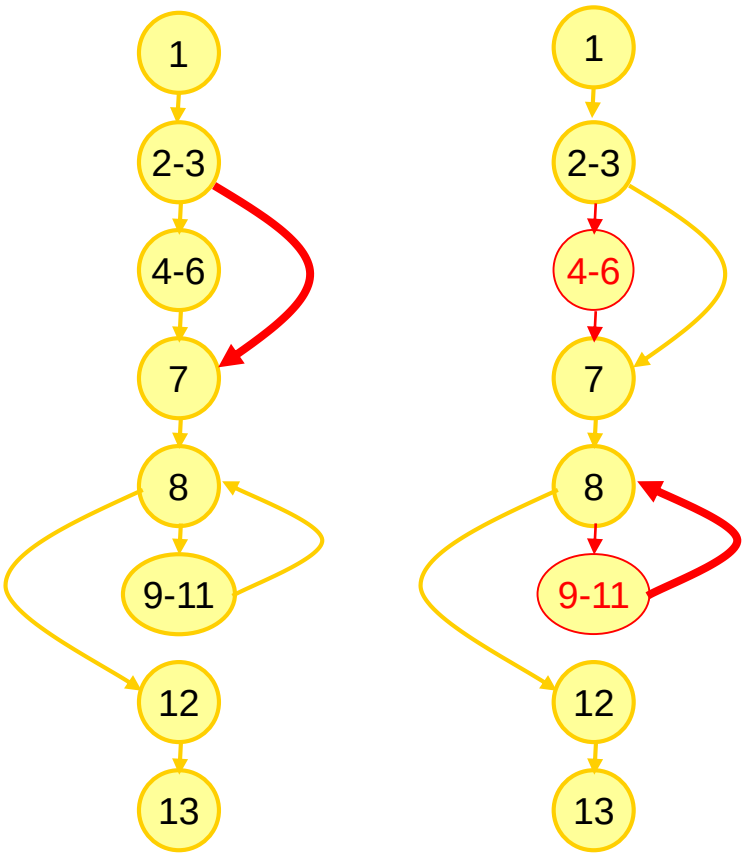
- Statement coverage is a relatively weak criterium
  - Statement coverage doesn‘t care about „empty“ edges, i.e., edges which only bridge some nodes
  - Examples: ELSE edge (between IF and ENDIF) with empty ELSE part; jump to beginning of REPEAT loop; BREAK
  - Possibly missing statements in the contained program part are not discovered!
- 100% statement coverage is not always achievable
  - E.g. if program contains exceptional conditions which can not be simulated while testing
  - Can also be sign for unreachable („dead“) code
    => check with static analysis

Branch coverage

- Requires that all decisions of the test object are tested for all possible outcomes
- Decision is program part where control flow branches (branch = node with more than one outgoing edge)
- Test case selection:
  - Every possible outcome of a decision must be evaluated (IF/WHILE/FOR: true and false, CASE: all alternatives)
- Calculation of coverage only considers *whether* a decision has been investigated, not *how often*
- As always, pre- and postconditions, expected results, and behavior have to be determined in advance for every test case

$$\text{Branch coverage} = \frac{\text{\# of tested decision results}}{\text{\# of all decision results}}$$

Example: Branch coverage for gcd()




```
1.  public int gcd (int m, int n) {
    // pre: m > 0 and n > 0
    // post: return > 0 and
    // m@pre.mod(return) = 0 and
    //...
2.      int aux, rem;
3.      if (n > m) {
4.          aux = m;
5.          m = n;
6.          n = aux;
        }
7.      rem = m mod n;
8.      while (rem != 0) {
9.          m = n;
10.         n = rem;
11.         rem = m mod n;
        }
12.     return n;
13. }
```

Path 1 = (1, 2-3, 4-6, 7, 8, 9-11, 8, 12, 13)

Path 2 = (1, 2-3, 7, 8, 12, 13)

Branch coverage: Discussion

- Branch coverage is stronger than statement coverage
  - Branch coverage implies statement coverage
  - In contrast to statement coverage, branch coverage can discover missing statements (e.g. in empty ELSE parts)
- Conclusion:
  - Most important coverage criterium
  - Aim at 100% branch coverage (not always achievable!)

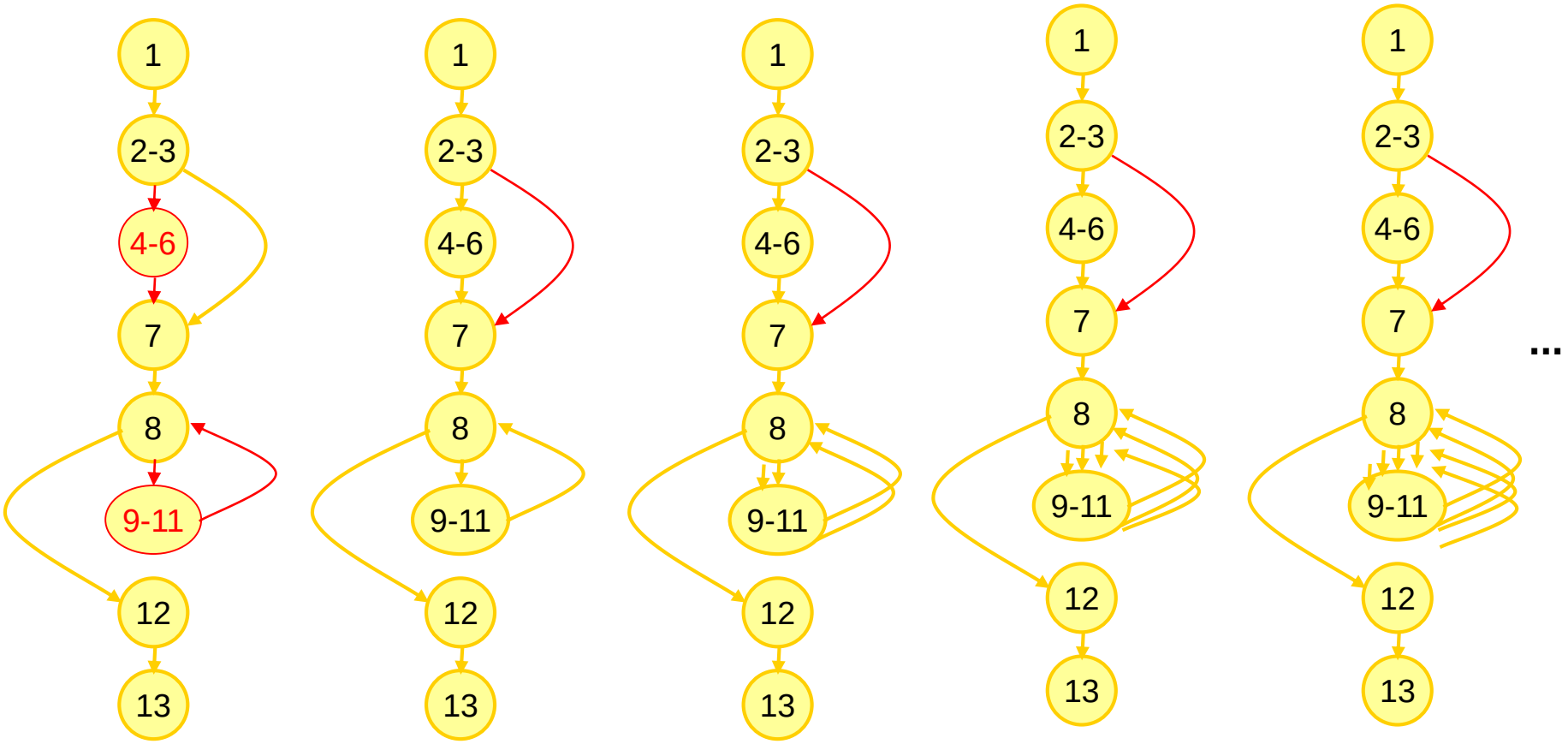
Path coverage

- Requires that every path of control flow graph is tested

$$\text{Path coverage} = \frac{\text{\# of tested paths}}{\text{\# of all paths}}$$

- A control flow graph containing cycles implies an infinite number of paths
  - But: Often upper bounds can be derived from specification or technical constraints
- Not achievable in practise, but important theoretical measurement (comparison with other coverage definitions)

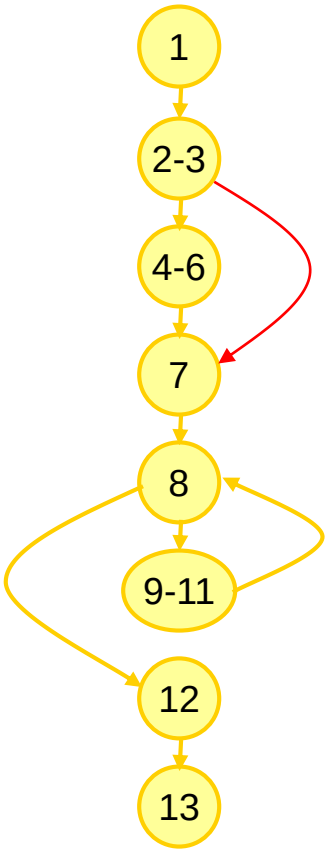
Example: Path coverage for `gcd()`



Deriving test cases from paths

- Up to now: Identification of paths through control flow graph; each path shall be „traversed“ by a test case
- But: Which input is needed to enforce a path?
- Idea:
  - Investigate conditions on path
  - Compute „constraints“ about variables such that conditions are satisfied

Statement coverage: Test case



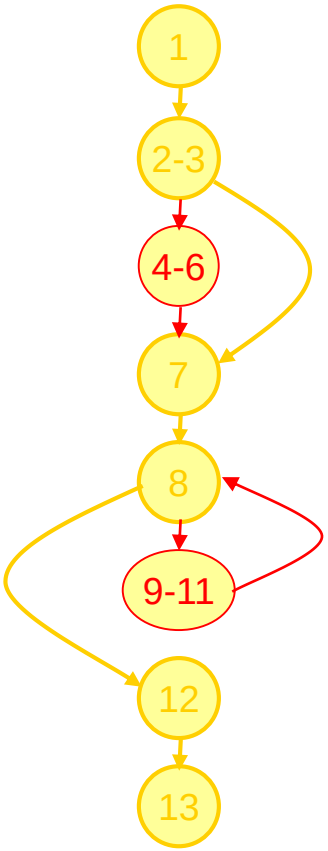
Path: (1, 2-3, 4-6, 7, 8, 9-11, 8, 12, 13)

Logical test case: { n > m ∧ n mod m ≠ 0 ∧ n mod (n mod m) = 0 ; gcd(m, n) }

Concrete test case: { m = 4, n = 6; 2 }

```
1.  public int gcd(int m, int n) {
2.      int aux, rem;
3.      if (n > m) {
4.          aux = m;
5.          m = n;
6.          n = aux;
        }
7.      rem = m mod n;
8.      while (rem != 0) {
9.          m = n;
10.         n = rem;
11.         rem = m mod n;
        }
12.     return n;
13. }
```

Branch coverage: Test case



Path: (1, 2-3, 7, 8, 12, 13)

Logical test case: {n ≤ m ∧ m mod n = 0 ; gcd(m, n) }
Concrete test case: { m = 4, n = 4; 4 }

```
1.  public int gcd(int m, int n) {
2.      int aux, rem;
3.      if (n > m) {
4.          aux = m;
5.          m = n;
6.          n = aux;
        }
7.      rem = m mod n;
8.      while (rem != 0) {
9.          m = n;
10.         n = rem;
11.         rem = m mod n;
        }
12.     return n;
13. }
```

White-box testing and OO

- Statement coverage as well as branch coverage are not well suited for object-oriented programs
  - Typical methods of a class are pretty small
  - Statement and branch coverage easy to achieve, but not very meaningful
  - Often, complexity of OO systems is hidden in the interactions of the objects
  - Also, testing of oo concepts like inheritance, polymorphism, dynamic binding, type casting needs special treatment
- See related literature!

Other white-box testing approaches

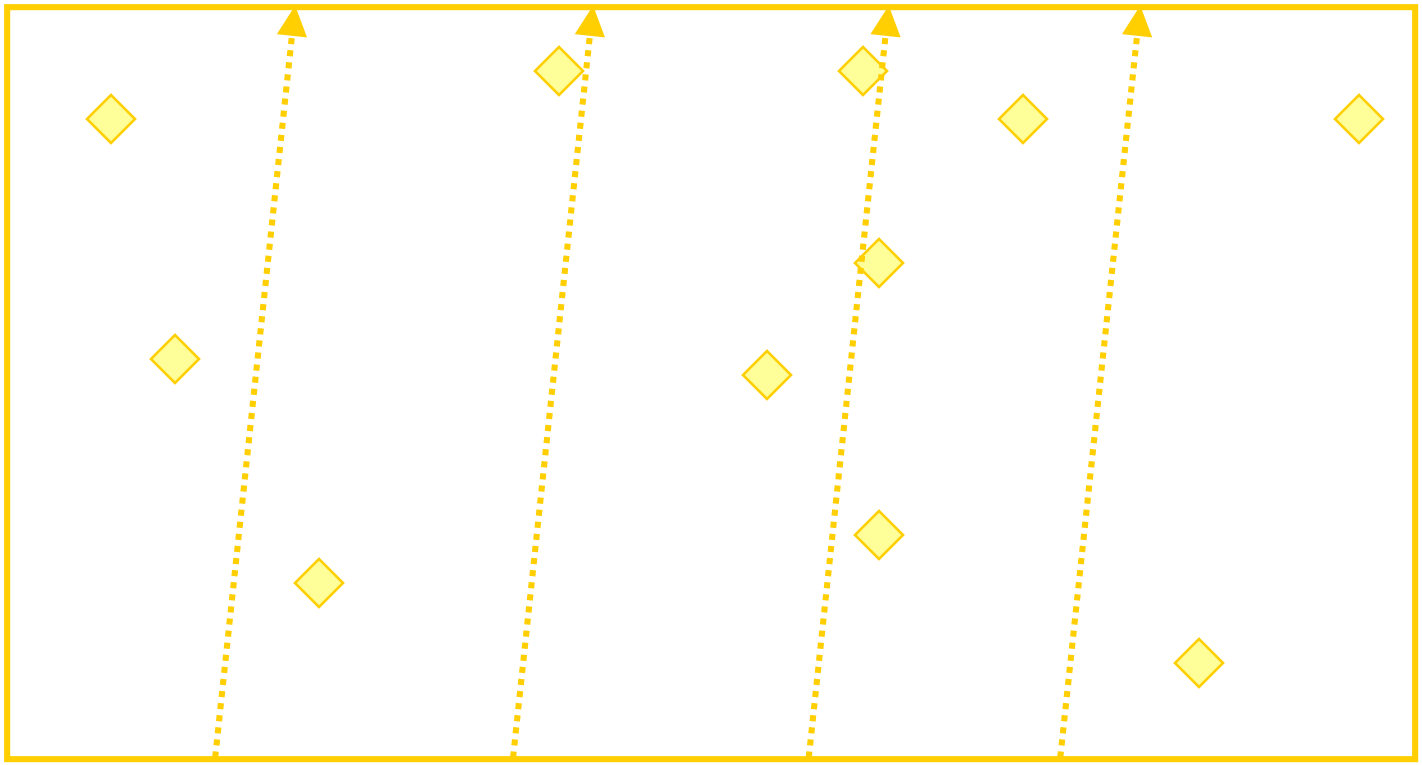
- Data flow based approaches
  - Reasoning about variables
  - Coverage is defined about usage of variables
  - Base: Control flow graph, annotated with information about definition and usage of variables

White-box testing approaches: Comparison

Type of coverage	Effectiveness	Evaluation
Statement coverage (C_0)	Low Detects about 20% of bugs	Necessary, but not sufficient Discovers «dead-code» Combine with other approaches!
Branch coverage (C_1)	Middle, depends heavily on concrete SUT Detects about 30% of all bugs and 80% of all control flow bugs	THE minimal test criterium! Discovers unexecutable branches Especially targets branches
Path coverage (C_∞)	High Detects about 70% of bugs	In most cases not feasible

Dangers of partial test strategies (1)

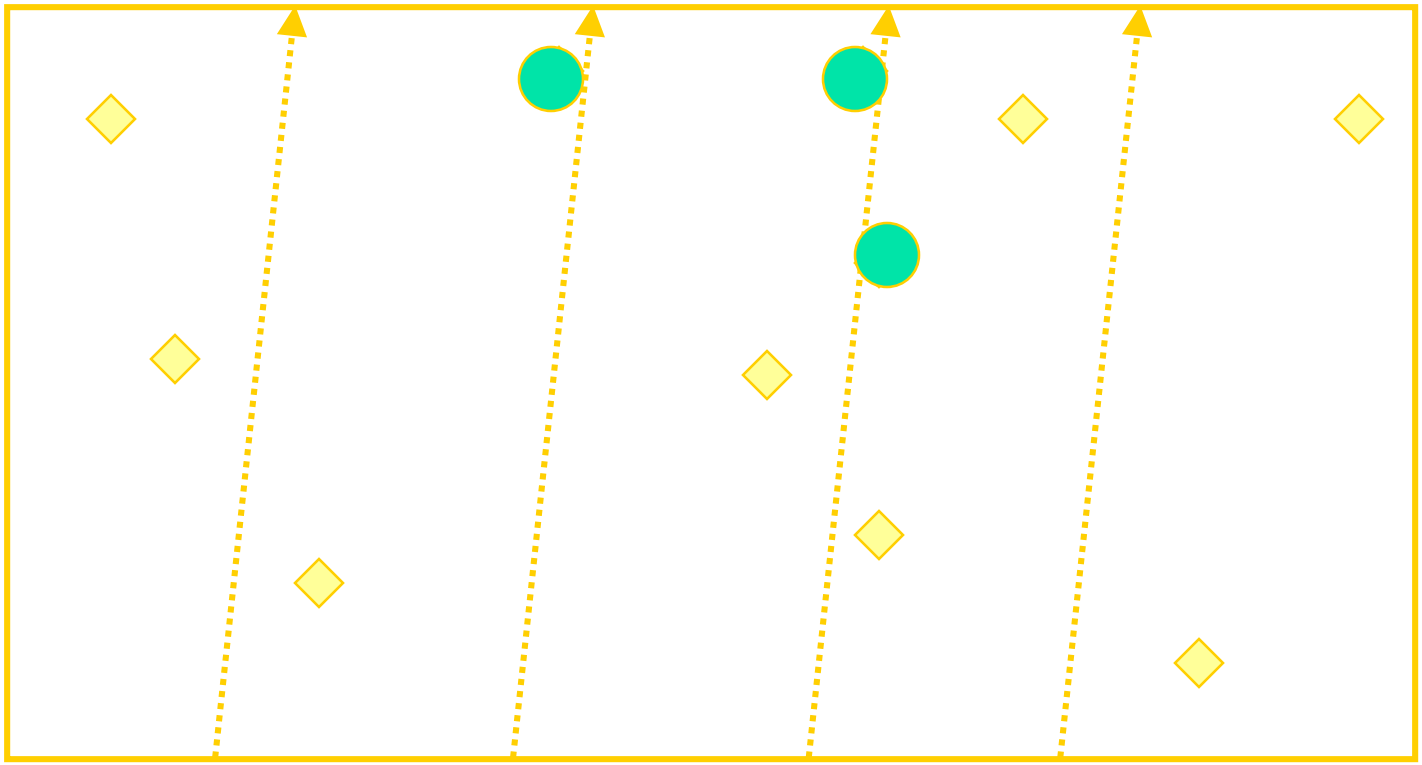
After: Rapid Software Testing,
copyright © 1996-2002 James Bach



Fault

Dangers of partial test strategies (1)

After: Rapid Software Testing,
copyright © 1996-2002 James Bach

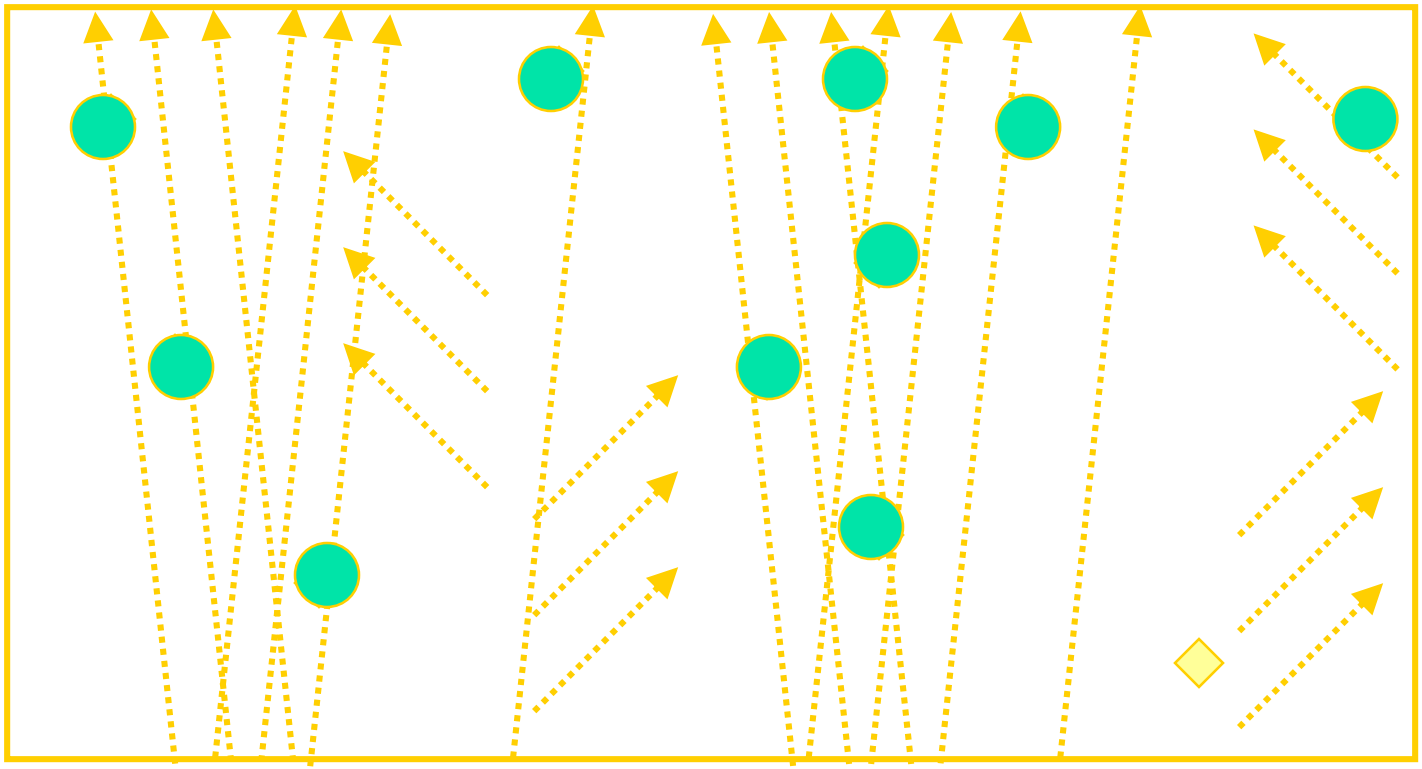


Fault

Discovered (and fixed) fault

Efficiency by variation of testing strategies

After: Rapid Software Testing,
copyright © 1996-2002 James Bach



Fault

Discovered (and fixed) fault

So, what's a good test case?

- Properties of a good testcase
  - Power – When a problem exists, the test will reveal it
  - Valid – When the test reveals a problem, it is a genuine problem
  - Relevant – It reveals things your clients want to know about the product or project
  - Credible - Your client will believe that people will do the things that are done in this test
  - Representative – problem most likely to be encountered by the user
  - Not redundant – This test represents a larger group that address the same risk
  - Executable – Can be executed as specified (no additional knowledge etc. needed)
  - Maintainable, repeatable, easy to evaluate, adequate complexity, low costs, …
- Properties depend on technique used. Examples:
  - Use case based test – focus on validity, relevance, complexity, but not maintainability
  - Equivalence classes – not redundant, repeatable, easy to maintain, but not very representative and credible

After: Cem Kaner:
What Is a Good Test Case?

Testing: Conclusions (1)

- Overall goal: With as less effort as possible, create different test cases which uncover faults with a reasonable probability
  - Choose test design approach taking the SUT into consideration
  - Make sure that functionality of SUT is tested appropriately
  - When defining test cases, identify expected behavior of SUT as well as expected results
- Start with black-box testing
  - Combine equivalence classes with boundary value analysis!
  - If behavior of SUT depends on its state, use state-based testing!
  - When performing tests, measure statement coverage, branch coverage and path coverage (if possible)
  - If code coverage is not enough, continue with white-box testing

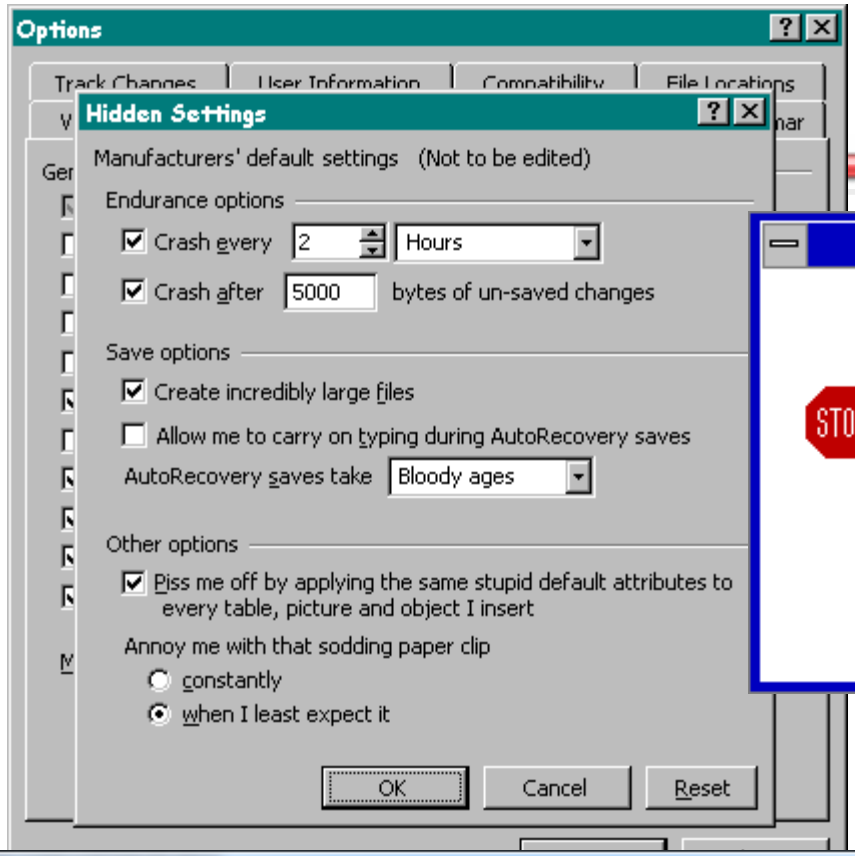
Testing: Conclusions (2)

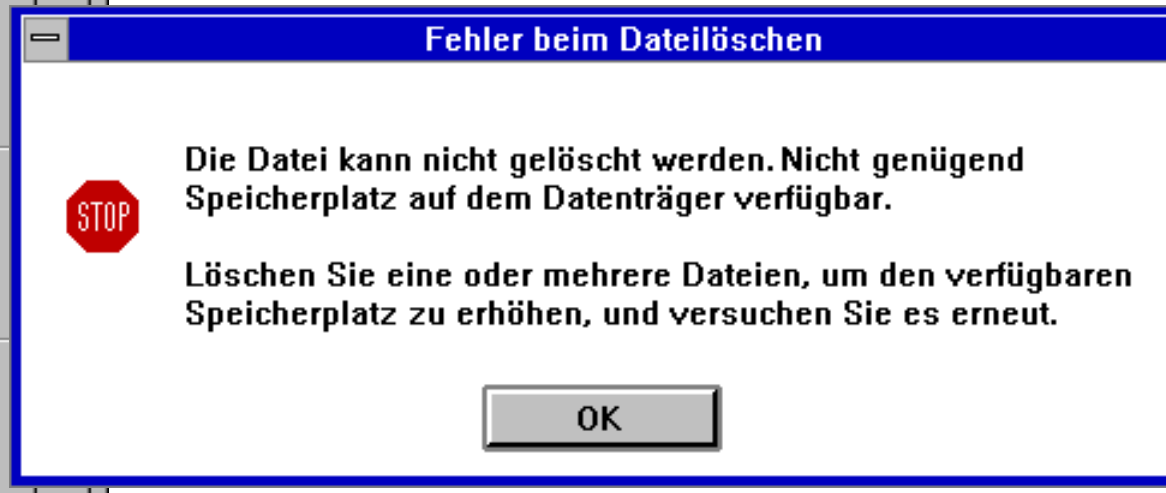
- Continue with white-box testing
  - Depending on SUT and „criticality“, choose appropriate coverage criterium
  - Minimal criterium: Branch coverage
  - When measuring coverage, make sure that loops are also executed more than once
  - Critical systems: Loops have to be checked separately (e.g., prove termination)
  - Path coverage is too expensive in almost all cases

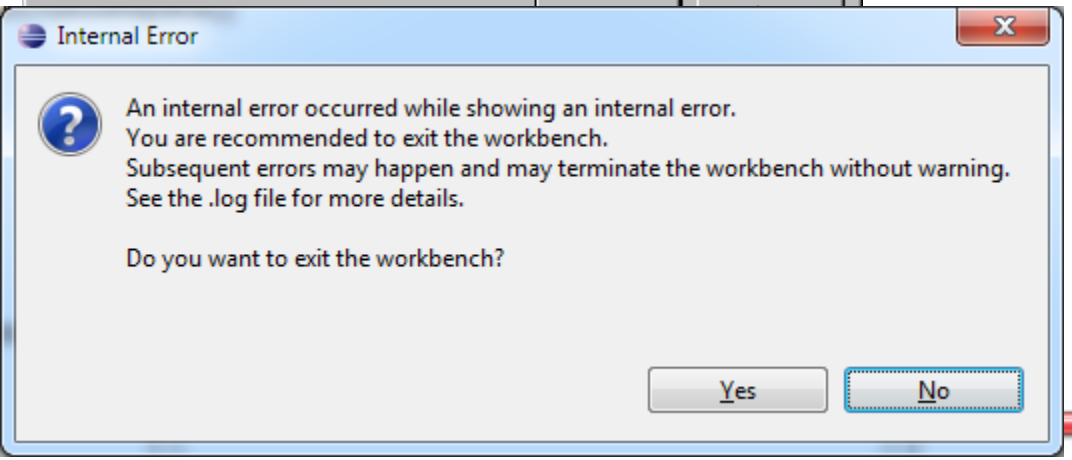
Testing: Conclusions (3)

- V model: white-box tests are suited for lower test levels (e.g., component testing), where black-box approaches can be used on higher test levels (e.g., acceptance testing)
- Testing always combines different approaches! There is no testing approach which takes care of all aspects!
- Proper selection of testing approaches is crucial. Main criterium is the risk of faults which have not been discovered

Last but not least...







- Model-based Testing of Smart-Card Protocols
- Automated Test Planning via Requirement Analysis
- Test Automation for Finance
- Testing can be funny: test to rest

Bundesamt
für Sicherheit in der
Informationstechnik

HJP CONSULTING.

TÜViT®

s-lab
Software Quality Lab

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

SPECIFICATION
- Guidelines, such as BSI TR-03110
- Norms, such as ISO 7816
- Standards, such as PKCS#15

MODEL
- Includes an abstraction of specifications
- Based on well-established Unified Modeling Language (UML)
- Model is machine-readable

TEST GENERATOR
- Derives test cases in an automated and systematic way
- Can be configured by test engineer

TEST SPECIFICATION
- Includes test suites and test cases
- Includes metrics of test depth and test coverage

CONFORMITY TEST / ACCEPTANCE TEST
- Automatic test execution based on customer-specific test
- Based on GlobalTester™ test tool

TEST REPORT
- Provides results of all test cases including test coverage

COSTUMER-SPECIFIC REQUIREMENTS

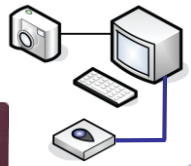
TEST EXPERIENCE
The test engineer:
- Analyzes the specifications
- Implements the model as an abstraction of specifications
- Adds customer-specific requirements
- Adds his domain experience

HJP MBT TOOL

HJP GLOBALTESTER GT

THE NEW APPROACH: MODEL BASED TESTING

Automated Test Planning via Requirement Analysis



Customer

Consultant

Supplier

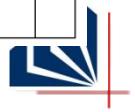
- >5.000 pages
- redundancies
- dependencies
- expensive testing

- Linguistic analysis
- Clustering
- Pattern matching

- free of redundancies
- efficient

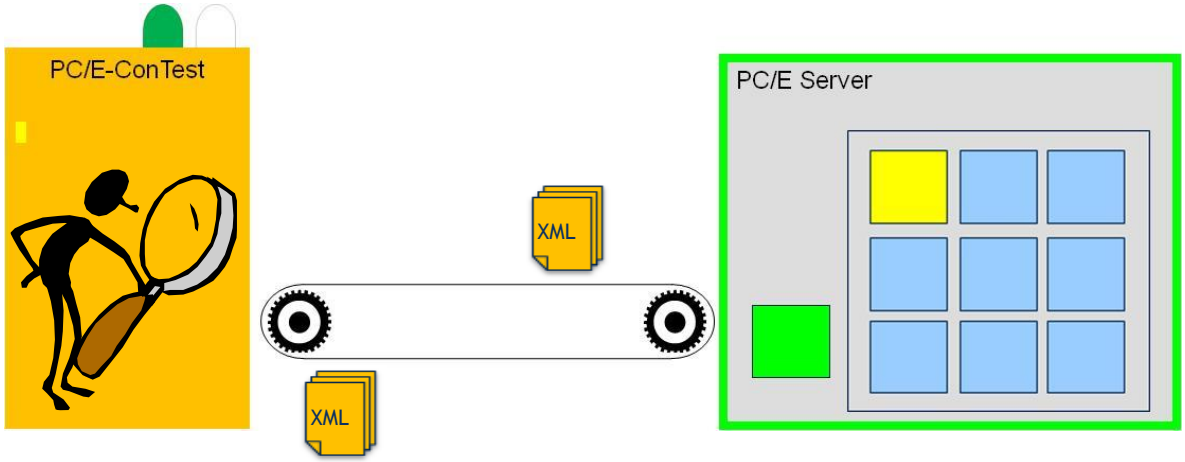
Test plan						
Activity	ID	Actor	Process	Direct Object / Adjective	Accept	
Test step	AC62	recipient	provide	his bill and his ID card	$exec(AC62)$	
Assert	ST65	bill	be	paid	Is bill paid?	
Assert	ST87	ID card	be	valid	Is ID card valid?	
Test step	AC103	officer	authorize	application	$exec(AC10) \wedge valid(ST36)$	
Assert	ST36	application	be	signed	Is application signed?	
Test step	AC265	officer	personalize	e-passport	$exec(AC26) \wedge valid(RL56)$	
Assert	RL56	e-passport	conform	ICAO	Does e-passport conform to ICAO?	
				Total	$\bigwedge_{id \in \{ID\}} valid(id)$	

HJP CONSULTING.



Test Automation for Finance

- Test automation for a J2EE client/server system for banking in
- Two steps process
  - test case definition
  - Execution of test cases and analysis of test results
- Efficient repeatable test process



WINCOR NIXDORF

Advertising: Your career in s-lab!

- PHD Position
- SHK
- BS/MS Thesis

s-lab
Software Quality Lab

Position at industry-driven reserach

- Researcher position with opportunity of PHD
- Partner: arvato IT services GmbH
- Location: Gütersloh
- Topics of interest: Software Testing, Model-driven Development

http://www.arvato.com/de.html

arvato BERTELSMANN

SHK Job Opportunity in On-The-Fly Computing

SFB 901
ON-THE-FLY COMPUTING

You …

- …are experienced in **Java programming**
- …are eager to expand your knowledge in **Eclipse Plugin Development & Model-Driven Engineering**
- …speak **English** and think **analytically**

We …

- …are an **international** team of students & PhD students
- …offer **conceptual** and **technical** tasks like
  - Specification Editors, Matching Algorithms, Model Transformations
- …offer insights into **most recent research**
- …offer work on conference papers, Master theses, …

http://sfb901.upb.de/sfb-901/projects/project-area-b

Contact us: s.arifulina@upb.de m.platenius@upb.de

SHK or BS or MS

- Messdatenerfassung und -visualisierung in der Verfahrenstechnik
- SHK (Aufwand: 19h/Woche) Link
- BS or MA possible Link
- Partner: http://www.loedige.de

Wir entwickeln und produzieren branchenübergreifend hochwertige Komponenten und Teilsysteme für verfahrenstechnische Anwendungen. Die Schwerpunkte liegen dabei in den Bereichen Mischen, Granulieren, Coaten, Trocknen, Reagieren und Emulgieren. Mit unseren individuell konzipierten Produkten und Anwendungslösungen unterstützen wir anspruchsvolle Kunden auf der ganzen Welt.

In Kooperation mit dem s-lab - Software Quality Lab der Universität Paderborn bieten wir kurzfristig folgende Bachelorarbeit und/oder Werkstudententätigkeit im Bereich der Informatik / Softwaretechnik an:

Messdatenerfassung und -visualisierung in der Verfahrenstechnik

(in Absprache ist eine Ausweitung der Aufgabenstellung für eine Masterarbeit möglich)

Unsere Aufgabe für Sie:
Im Rahmen der Bachelorarbeit und/oder Werkstudententätigkeit soll eine Softwarelösung für die Messdatenerfassung konzipiert und prototypisch implementiert werden. Diese soll vorrangig für die Überwachung von Laborgeräten und die Fehlererkennung eingesetzt werden.
Die Kommunikation erfolgt mithilfe von OPC, einer standardisierten Software-Schnittstelle für den Datenaustausch in der Automatisierungstechnik, über TCP/IP.
Über ein Feldbussystem werden I/O-Module oder Controller (SPS) an den OPC-Server angebunden, um Prozessdaten zu übertragen. Ein OPC-Client stellt die Prozessdaten grafisch dar und unterstützt die Analyse, z. B. für die Prozessüberwachung oder Fehlerdiagnose.
Die Entwicklung der Client-Software soll vorzugsweise in C# erfolgen. Neben der Visualisierung der im Datenlogger protokollierten Messdaten sollen auch ein Datenexport, eine Ereigniserkennung (z.B. Schwellwertverletzung) und ein einfaches Lizenzmanagement unterstützt werden.

Ihr individuelles Profil:
- Student der Fachrichtung Elektrotechnik oder Informatik.
- Kenntnisse in mindestens einem der vorgenannten Themengebiete sind vorteilhaft.
- Eigeninitiatives, prozess- und zielorientiertes Arbeiten.

Fühlen Sie sich angesprochen?
Dann freuen wir uns auf Ihre Bewerbungsunterlagen.

Ihre Ansprechpartner:
Falk Westermann
Leiter Steuerung und Elektroengineering
Telefon: +49 5251 309-313
E-Mail: Westermann@loedige.de

s-lab – Software Quality Lab:
Dr. Stefan Sauer
Geschäftsführer
Telefon: +49 5251 60-5390
E-Mail: sauer@s-lab.upb.de

Gebrüder Lödige Maschinenbau GmbH
Elsener Straße 7-9 · 33102 Paderborn
Telefon +49 5251 309-249 · Telefax +49 5251 309-64249
E-Mail gottschalk@loedige.de · Internet http://www.loedige.de
LÖDIGE - ALWAYS THE RIGHT MIX

LÖDIGE PROCESS TECHNOLOGY

SHK job for app development

- Various app development projects
- 9,5 or 19 Std/Woche
- Contact

Dr. rer. nat. Simon Oberthür

- Many cool topics on testing, e.g.
  - Mobile Testing
  - Testing Smart Homes
  - DSL‘s for Testing
  - Keyword-driven Testing
  - Agile Testing
  - Test automation/Model-based testing
- Contact