Test Design Technique

Rashed Karim

Test Design Technique

Agenda

- Static test design technique
- Dynamic test design technique

Chapter III – Static Testing Techniques

- III/01 Reviews and the test process
- III/02 Review process
- III/03 Tool based static analysis

Basic Approach

- **Static** testing techniques summarize various methods, that **do not execute** the component or the system that is being tested.
- Static tests include:
  - ✔ **review** (manual activity)
  - ✔ **static analysis** (mostly tool based activity)
- Static techniques component dynamic methods
  - ✔ static tests find defects rather than failures
  - ✔ concepts are inspected as well, not only executable code
  - ✔ defects / deviations are found in an early phases, before they are implemented in the code
  - ✔ Static tests might find defects not found in **dynamic** testing
- **High quality documents** lead to high quality product
  - ✔ Even if the reviewed specifications do not contain any errors, interpreting the specification and creating design could be faulty

Static Testing Techniques

- Tool based static analysis

- Analytical QA
  - dynamic
    - Black box
      - Equivalence partitioning
      - Boundary value analysis
      - State transition testing
      - Decision tables
      - Use case based testing
    - Experience-based techniques
    - White box
      - Statement coverage
      - Branch coverage
      - Condition coverage
      - Path coverage
  - Static
    - Review/walkthroughs
    - Control flow analysis
    - Dataflow analysis
    - Computer metrics/analyzer

Terminology and Definitions

- **Static analysis (Definition):**

  Static analysis is the task of analyzing a test object (e.g. source code, script, requirement) without executing the test object.

- **Possible aspect to be checked with static analysis are**:

  1. programming rules and **standards**
  2. program design (**control flow** analysis)
  3. use of data (**data flow** analysis)
  4. **complexity** of the program structure (metrics, e.g. the cycloramic number)

General aspects /1

- **All test objects must have a formal structure**
  1. this is especially important when using testing tools
  2. very often documents are not generated formally
  3. in practice, modeling, **programming** and scripting **languages** comply to the rule, some diagrams as well

- **The tool-based static analysis of a programmed with less effort than an inspection.**
  - therefore, very often a static analysis is performed before a review takes place

General aspects /2

Tools to be used are compilers and analyzing tools (analyzer)

- **Compiler**

1. detect syntax errors inside a program source code
2. create reference data of the program (e.g. cross reference list, call hierarchy, symbol table) of the program
3. check for consistency between types of variables
4. find undeclared variables and unreachable (dead) code

- **Analyzer address additional aspects, such as**

  - conventions and standards

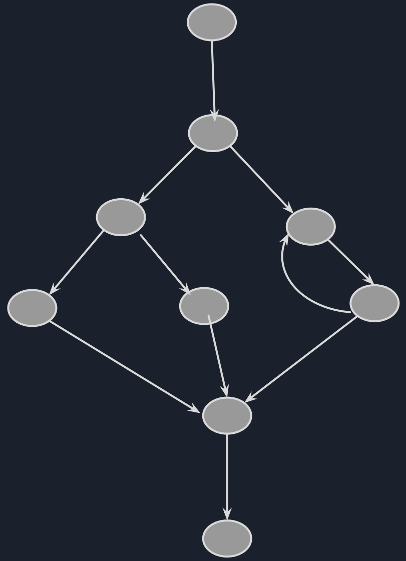
  - metrics of complexity

  - object coupling

Control flow analysis /1

- **Aim**
  - To find defects caused by wrong construction of the program code (dead branches, dead code etc.)

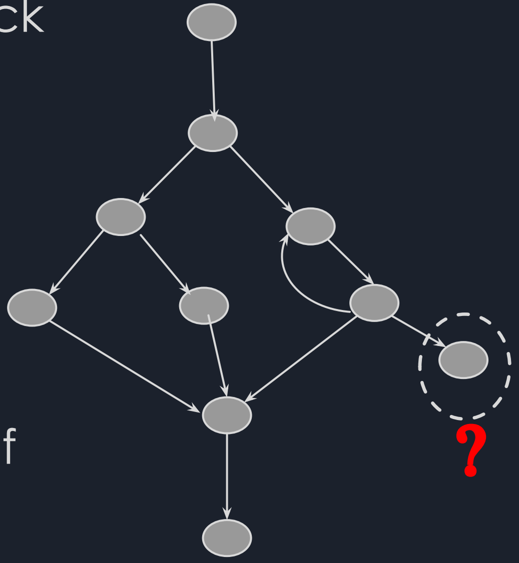
- **Method**
  1. Code structure is represented as a control flow graph
  2. Directed graph
     a. Nodes represent statements or sequences of statements
     b. Edges represent control flow transfer, as in decisions and loops
     c. Tools based construction



Control Flow analysis /2

Result

- easy understandable overview of program code
- anomalies* can easily be detected, defects stick out
    - loops exited by jumps
    - dead branches
    - multiple returns
- a control flow graph is just a simplified version of flow chart



* Anomalies: an irregularity of inconsistency

Metrics and their computation

- Using Metrics, certain aspect of program quality can be measured
  - The metric has only relevance for the measured aspect
- The static complexity of the program can be measured
  Currently, there are about 100 different metrics available

- Different metrics address different aspects of program complexity
  - program size (e.g. Lines of Code - LOC)
  - program control structure (e.g. cyclomatic complexity)
  - Data control structures (e.g. Halstead complexity)

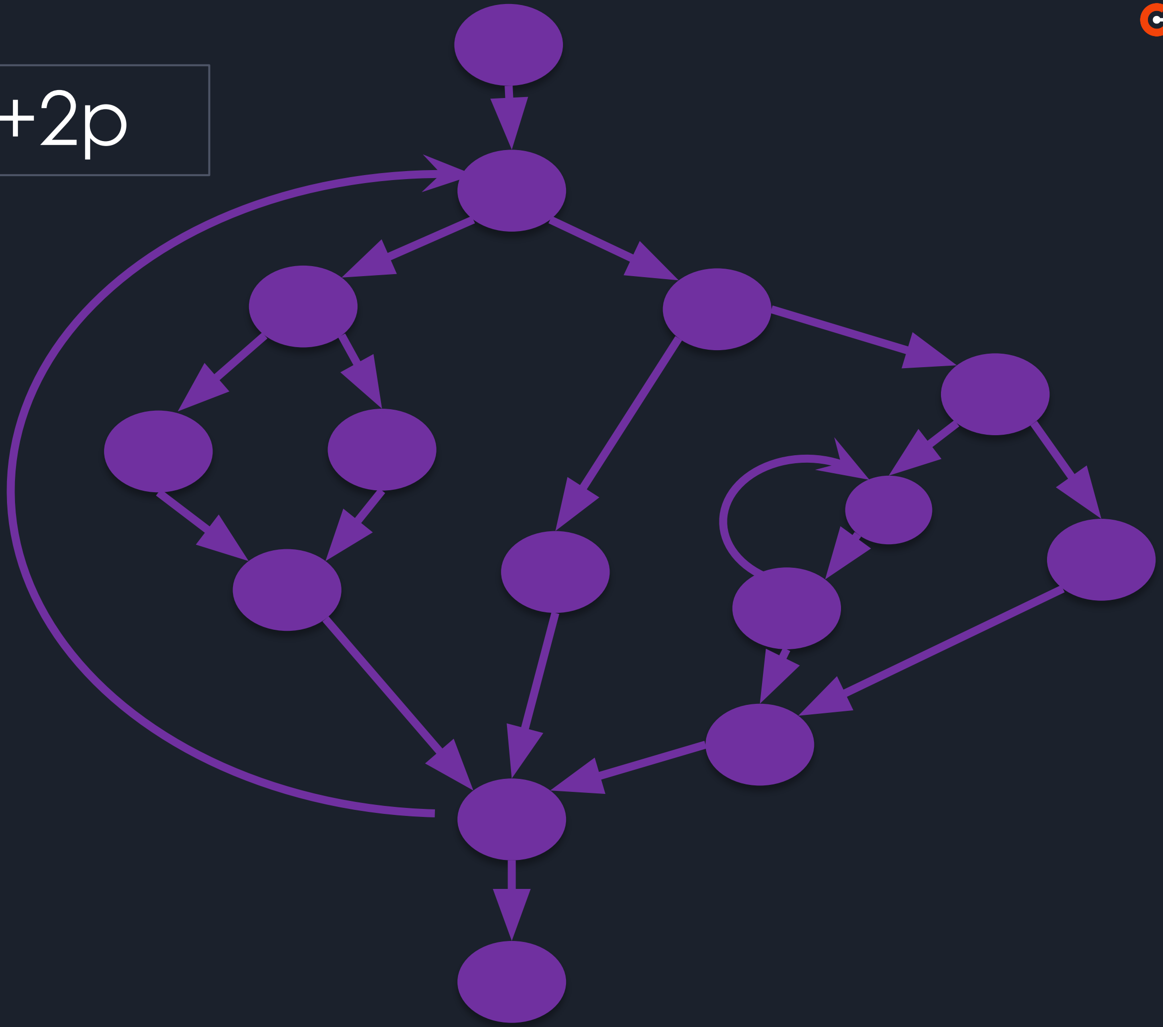
- It is difficult to compare different metrics, even if they address the same attribute of the program !

Metrics and their implementation /1

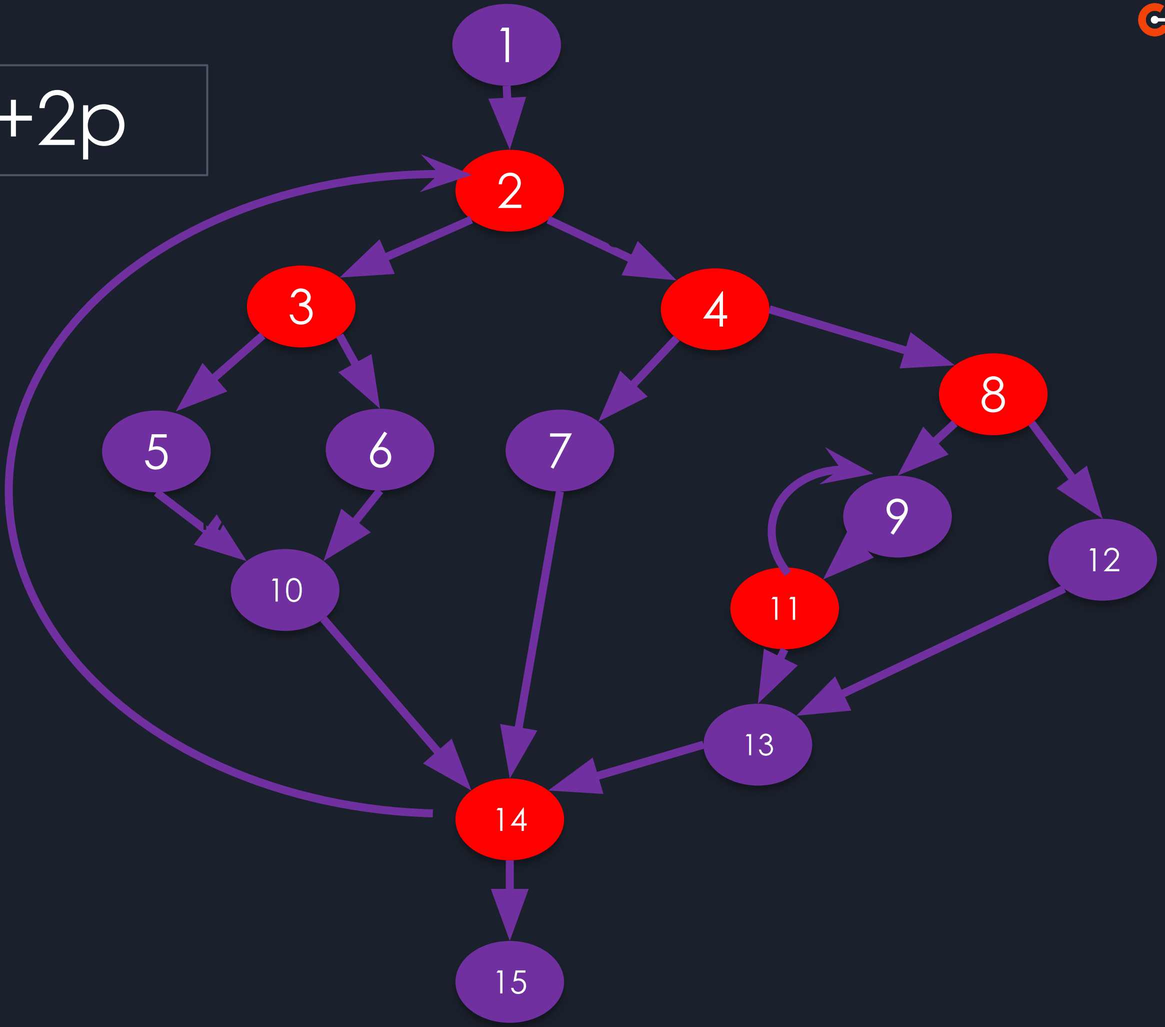
Cyclomatic number v(G)

- Metric to measure the static complexity of a program based on its control flow graph
- Measure linear independent program paths, as an indication for testability and maintainability
- The cyclomatic number is made up of
  - Number of edges e
  - Number of nodes n
  - Number of inspected independent program parts p (mostly 1)
- Values up to 10 are acceptable. Beyond this, code should be reworked/improved (best practice, McCabe)

$$v(G)=e-n+2p$$



$$v(G)=e-n+2p$$



FULL STACK

Metrics and their implementation /2

Cyclomatic Complexity

1 program part : $p = 1$

15 nodes : $n = 15$

20 edges : $e = 20$

The number to a cyclomatic complexity

$v(G) = e-n+2p$

$v(G)=7$

FULL STACK

Metrics and their implementation /3

Cyclomatic Complexity (by Thomas J. McCabe) - implication

- The cyclomatic complexity can be used as a target value for code reviews
- The cyclomatic complexity can also be calculated as the number of independent decisions plus one. If both ways of calculation give different results, it may be due to
  - ✔ A superfluous branch
  - ✔ A missing branch
- The cyclomatic complexity also gives as indication for the number of necessary test cases (to achieve decision coverage)

Summary

Static Analysis

- Using tools for static analysis (complier, analyzers), program code can be inspected **without** being executed.
- Using **tools**, the **static analysis** of a program can be performed with **less effort** than an **inspection**.

Analysis results

- The **control flow diagram** shows the program flow and allows for the detection of "dead branches" and unreachable code
- Data anomalies are found using **data flow analysis**
- **Metrics** can be used to access the structural complexity, leading to an estimation of testing efforts to expect

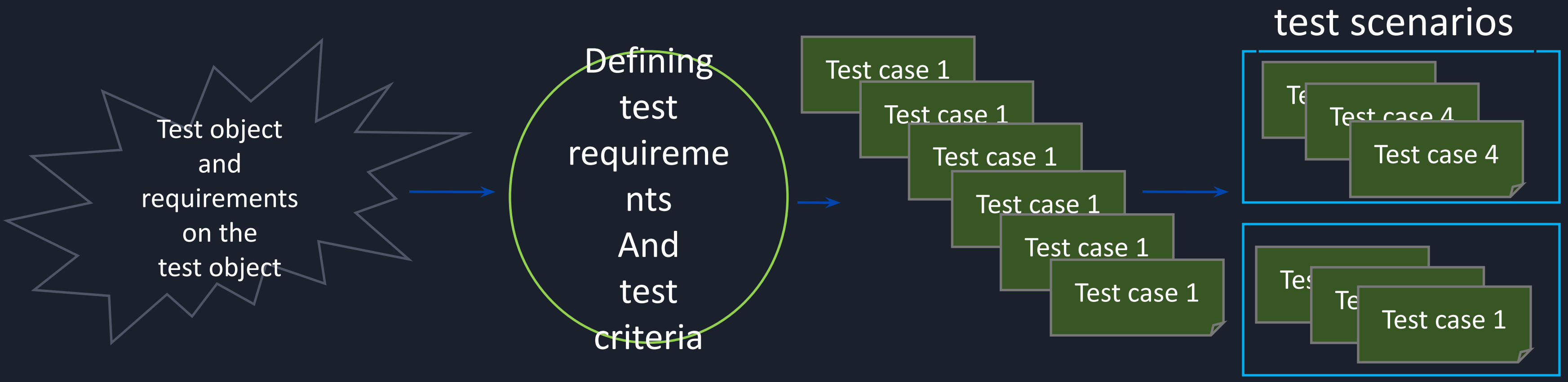
Test Design Technique

Designing Test cases

Deriving test cases from requirements

Deriving test cases must be a **controlled** process.

- Test cases can be **created** in a **formal** way or in an **informal** way, depending on the project delimitations and on the maturity of the processes in use.
- Test case should be **traceable.**



Deriving Test Case from Requirement:

-Test Object:

The subject to be examined: a document or a piece of software in the software development process

-Test Condition

An item or an event: a function, a transaction, or an element in the system

-Test Criteria

The test object has to confirm the test

Test case description according to IEEE 829:

- **Distinct identification: Id or key** in order to link, for example, an error report to the test case where it appeared
- **Pre-Conditions:** situation previous to test execution or characteristics of the test object before conducting the test case
- **Input Values:** description of the input data on the test object
- **Expected Result:** output data that the test object is expected to produce
- **Post-Conditions:** Characteristics of the test object after test execution, description of its situation after the test
- **Dependencies**: order of execution of test cases, reason for dependencies
- **Requirements**: Characteristics of the test object that the test case will examine

Black-box Technique:

The Tester looks the test object as a Black Box

-internal structure of the test object is irrelevant or unknown

-Testing of input/ output behavior

-Black box testing is also called functional testing or specification oriented testing

White-box Technique:

The Tester knows the internal structure of the program
 and code
-i.e. component hierarchy, control flow, data flow

Test case are selected on the basis of internal program
Code/ program structure

White box testing is also called structure based testing
Or control flow based testing

Categories of test design methods:

Specification based methods:
(Black Box)

Structure based methods:
(White Box)

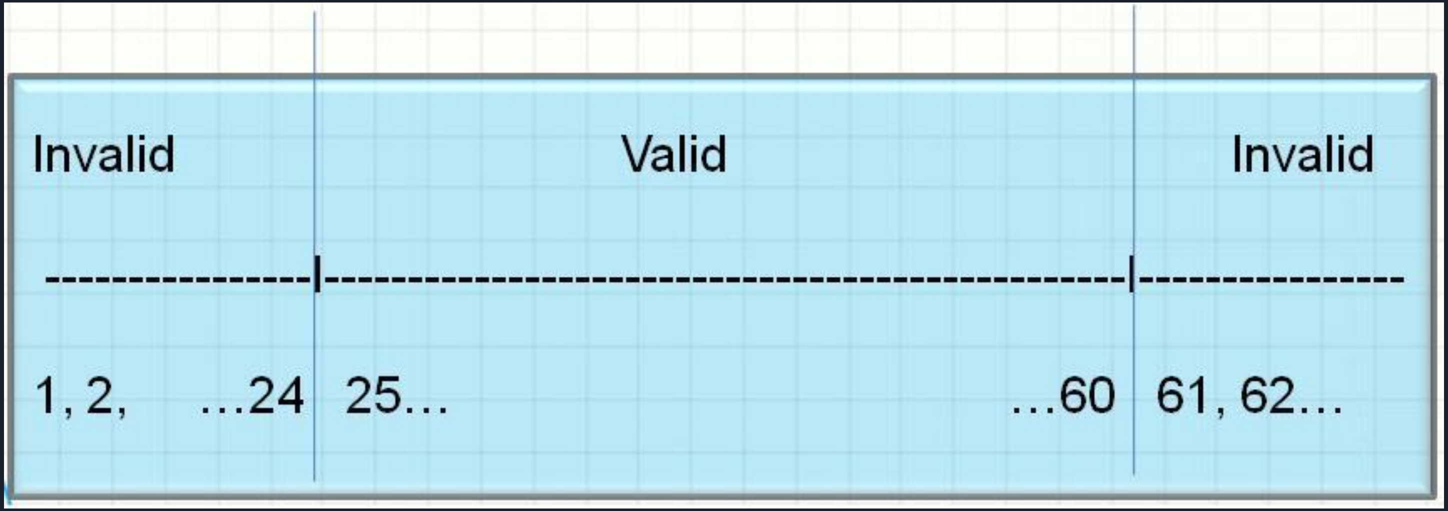
Experienced based methods:
(Black Box)

Equivalence class (EC) partitioning

- The range of defined values is grouped into **equivalence classes**, for which the following **rules** apply:
  - All values, for which a **common behavior** of the program is **expected**, are grouped together in one equivalence class
  - Equivalence class may **not overlap** and may **not contain any gaps**
  - Equivalence class may contain a **range** of values(e.g. 0<x<10) or a **single** value (e.g. x="Yes")
- **Valid EC**
- **Invalid EC**:

FULL STACK

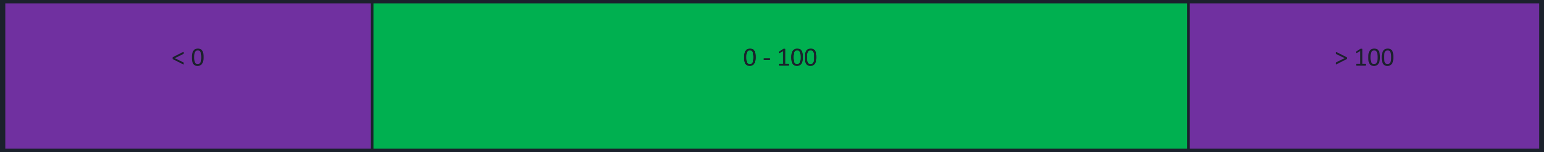
Age limit = between 25 and 60
25<=Age=<60



Equivalence class partitioning – example

- Equivalence classes are chosen for valid and invalid inputs
    - if a value x is defined as **0≦ x ≦ 100**, then when we can initially identify three equivalence classes:
        1. **x<0** (invalid input values)
        2. **0 ≦ x≦100** (valid input)
        3. **x> 100** (invalid input values)

- Further invalid EC can be defined, containing, but not limited to:
    non-numerical inputs,
    numbers to big or to small,
    non-supported format for numbers



Equivalence class partitioning – example

Problem:

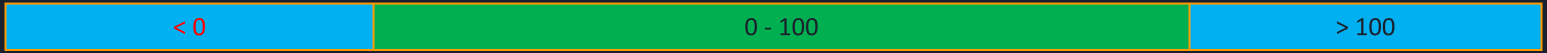
A program expected a **percentage** value according to the following requirements:

- only integer values are allowed
- 0 is the valid lower boundary of the range
- 100 is the valid upper boundary of the range

Solution:

- **Valid** are all numbers from 0 to 100,
- **Invalid** are all negative numbers,
  - all numbers greater than 100,
  - all decimal numbers and
  - all non numerical values (e.g. “rashed”)

- one valid equivalence class: $0 \leq x \leq 100$
- 1st invalid equivalence class: $x < 0$
- 2nd invalid equivalence class: $0 > 100$
- 3rd invalid equivalence class: x = no integer
- 4th invalid equivalence class: x = not numeric (e.g. “abc”)



Equivalence class partitioning – example

Additional Requirement:

The percentage value will now be displayed in a bar chart.

The following additional requirements apply (both values included):

- values between 0 and 15 :	Orange bar,
- values between 16 and 50:	Green bar,
- values between 51 and 85:	Yellow bar,
- values between 86 and 100:	Blue bar,

Additional Solution for valid EC:

- Now there are four instead of one valid equivalence classes:

- 1st valid equivalence class:	**$0 \leq x \leq 15$**
- 2nd valid equivalence class:	**$16 \leq x \leq 50$**
- 3rd valid equivalence class:	**$51 \leq x \leq 85$**
- 4th valid equivalence class:	**$86 \leq x \leq 100$**

< 0	0 - 15	16 - 50	51 - 85	86 – 100	> 100

FULL STACK

EC Partitioning – Picking Representatives

Variable	Status	EC	Representative
Percentage Value	Valid	EC 1: 0<= X <=15	+10
		EC 2: 16<= X <=50	+20
		EC 3: 51<= X <=85	+80
		EC 4: 86<= X <=100	+90
	Invalid	EC 5: X<0	-10
		EC 6: X>100	+200
		EC 7: X not integer	1.5
		EC 8: X non number	fred

Equivalence class partitioning – example 2 /1

Analyzing the specification

A piece of code computes the **price** of a product, based on its **value**, a **discount** in % and **shipping costs** (BDT 19, 29, 49 depending on shipping mode)

Variable	Equivalence class	Status	Representatives
Values of goods	EC_{11}: x >= 0	Valid	1000
	EC_{12}: x < 0	invalid	-1000
	EC_{13}: x non-numerical value	Invalid	Rashed
Discount	EC_{21}: $0\% \leq x \leq 100\%$	valid	10%
	EC_{22}: x < 0%	Invalid	-10%
	EC_{23}: > 100	Invalid	200%
	EC_{24}: x non numeric value	Invalid	Karim
Shipping costs	EC_{31}: x = 19	valid	19
	EC_{32}: x = 29	valid	29
	EC_{33}: x = 49	valid	49
	EC_{34}: $x \neq \{19, 29, 49\}$	Invalid	30
	EC_{35}: x non numeric value	invalid	Student

Equivalence class partitioning – example 2 /2

Test cases for valid EC:

Valid equivalence classes provide the following combinations or test cases: T1, T2 and T3

Variable	Equivalence class	Status	Representatives	T1	T2	T3
Values of goods	EC_{11}: x >= 0	Valid	1000	*	*	*
	EC_{12}: x < 0	invalid	-1000			
	EC_{13}: x non-numerical value	Invalid	Rashed			
Discount	EC_{21}: 0% ≤ x ≤ 100%	valid	10%	*	*	*
	EC_{22}: x < 0%	Invalid	-10%			
	EC_{23}: > 100	Invalid	200%			
	EC_{24}: x non numeric value	Invalid	Karim			
Shipping costs	EC_{31}: x = 19	valid	19	*		
	EC_{32}: x = 29	valid	29		*	
	EC_{33}: x = 49	valid	49			*
	EC_{34}: x ≠ {19, 29, 49}	Invalid	30			
	EC_{35}: x non numeric value	invalid	Student			

Equivalence class partitioning – example 2 /3

Test cases for invalid EC: The following test cases were created using the invalid EC, each in combination with valid ECs of other elements:

Variable	Equivalence class	Status	Representatives	T4	T5	T6	T7	T8	T9	T10
Values of goods	EC_{11}: x >= 0	Valid	1000			*	*	*	*	*
	EC_{12}: x < 0	invalid	-1000	*						
	EC_{13}: x non-numerical value	Invalid	Rashed		*					
Discount	EC_{21}: 0% ≤ x ≤ 100%	valid	10%	*	*				*	*
	EC_{22}: x < 0%	Invalid	-10%			*				
	EC_{23}: > 100	Invalid	200%				*			
	EC_{24}: x non numeric value	Invalid	Karim					*		
Shipping costs	EC_{31}: x = 19	valid	19	*	*	*	*	*		
	EC_{32}: x = 29	valid	29							
	EC_{33}: x = 49	valid	49							
	EC_{34}: x ≠ {19, 29, 49}	Invalid	30						*	
	EC_{35}: x non numeric value	invalid	Student							*

Equivalence class partitioning – example 2 /4

FULL STACK

All test cases:10 test cases are derived: 3 positive (valid values) and 7 negative (invalid values) test

Variable	Equivalence class	Status	Representative	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Values of goods	EC_{11}: x >= 0	Valid	1000	*	*	*			*	*	*	*	*
	EC_{12}: x < 0	invalid	-1000				*						
	EC_{13}: x non-numerical value	Invalid	Rashed					*					
Discount	EC_{21}: 0% ≤ x ≤ 100%	valid	10%	*	*	*	*	*				*	*
	EC_{22}: x < 0%	Invalid	-10%						*				
	EC_{23}: > 100	Invalid	200%							*			
	EC_{24}: x non numeric value	Invalid	Karim								*		
Shipping costs	EC_{31}: x = 19	valid	19	*			*	*	*	*	*		
	EC_{32}: x = 29	valid	29		*								
	EC_{33}: x = 49	valid	49			*							
	EC_{34}: x ≠ {19, 29, 49}	Invalid	30									*	
	EC_{35}: x non numeric value	invalid	Student										*

Equivalence class partitioning – coverage

Equivalence class coverage can be used as exit criteria to end testing activities

$$\text{EC Coverage} = \frac{\text{Number of EC tested}}{\text{Number of EC defined}} * 100\%$$

FULL STACK

Boundary Value Analysis

Boundary Value Analysis /1

- Boundary value analysis extends equivalence class partitioning by introducing a **rule** for the **choice of representatives**
- The **edge values** of the **equivalence class** are to be tested **intensively**
- **Why put more attention to the edges?**
  - Often, the boundaries of values ranges are **not well defined** or lead to different interpretations
  - Checking if the boundaries were **programmed correctly**

Please note:

Experience shows that **error** occur **very frequently** on the **boundaries** of value ranges!

Defining boundary values:

If the **EC** is defined as **single numerical value**,
for example **x= 5**, the **neighboring values** will be used as well
the representatives (of the class and its neighboring values)
are: **4,5 and 6**

Boundary Value Analysis /3

Boundary analysis example:

Value range for a discount in % : $0 \leq x \leq 100$

Definition of EC

EC1: $x < 0$ (Invalid)

EC2: $0 \leq x \leq 100$ (Valid)

EC3: $x > 100$ (Invalid)

Boundary analysis

extends the representatives with **neighboring values**:

EC2: -1; 0; 1; 99; 100; 101

Please note:

Instead of one representative for the valid EC, there are now six representatives (four valid and two invalid)

Boundary values optimization:

Basic scheme: choose **three values** to be tested the exact boundary and the two neighboring values (within and outside the EC)

Alternative point of view: since the boundary value belongs to the EC, only **two values** are needed for testing: **one within and one outside the EC**

Example:

Value range for a discount in %: 0 ≤ x ≤ 100

Valid EC: 0 ≤ x ≤ 100

Boundary analysis

Additional representatives are: **-1; 0; 100; 101**

1 – same behavior as 0

99 - same behavior as 100

A programming error caused by a wrong comparison operator will be found with the two boundary values

FULL STACK

Decision Table Testing

Decision Table Testing

- Equivalence class partitioning and boundary analysis deal with **isolated input** conditions.
- However, an input condition may have an effect only in **combination** with other input conditions.
- All previously described methods do not take into account the effects of **dependencies and combinations.**
- Using the **full set of combinations** of all input equivalence classes usually leads to a very **high number of test cases** (***test case explosion***).
- With the help of **cause-and-effect graphs** and the decision tables derived from them, **the amount of possible combinations** can be **reduced** systematically to a subset.

FULL STACK

Decision Table

Conditions (Causes)	Condition Entry
Action (Effects)	Action Entry

Business Case

Example: Online-Banking

The user has identified himself via account number and PIN. If having sufficient **Coverage** (enough balance), s/he is able to set a transferal. To do this he must input the correct details of the **Recipient** and a valid **TAN** (OTP)

FULL STACK

Decision Table

	T01	T02	T03	T04	T05	T06	T07	T08
Enough coverage	Yes	Yes	Yes	Yes	No	No	No	No
Correct recipient	Yes	Yes	No	No	Yes	Yes	No	No
Valid TAN	Yes	No	Yes	No	Yes	No	Yes	No
Do transferal	X							
Mark TAN as used	X							
Deny transferal			X	X	X	X	X	X
Request again TAN		X						

Decision Table

	T01	T02	T03	T04	T05	T06	T07	T08
Enough coverage	Yes	Yes	Yes	Yes	No	No	No	No
Correct recipient	Yes	Yes	No	No	Yes	Yes	No	No
Valid TAN	Yes	No	Yes	No	Yes	No	Yes	No
Do transferal	X							
Mark TAN as used	X							
Deny transferal			X	X	X	X	X	X
Request again TAN		X						

Decision Table

	T01	T02	T03	T04	T05
Enough coverage	Yes	Yes	Yes	Yes	No
Correct recipient	Yes	Yes	No	No	-
Valid TAN	Yes	No	Yes	No	-
Do transferal	X				
Mark TAN as used	X				
Deny transferal			X	X	X
Request again TAN		X			

Decision Table

	T01	T02	T03	T05
Enough coverage	Yes	Yes	-	No
Correct recipient	Yes	Yes	No	-
Valid TAN	Yes	No	-	-
Do transferal	X			
Mark TAN as used	X			
Deny transferal			X	X
Request again TAN		X		

Decision Table Testing

Example: Online-Banking

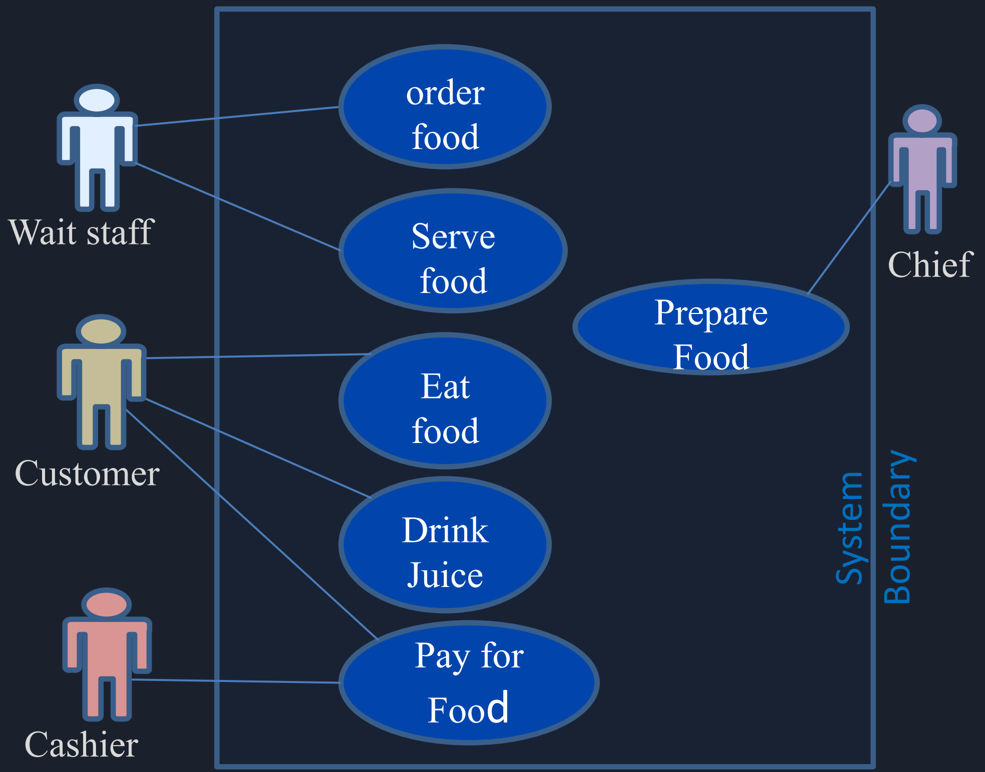
- Each table column represents a test case
- Choose an effect / action
- Each combination of conditions represents a column of the decision table (a test case)
- Identical combinations of causes, leading to different effects, may be merged, to form a single test case

Use Case Based Testing

Use Case based testing

- **Test cases** are derived directly from the **use cases** of the test object
  - The test object is seen as a **system** reacting with **actors**
  - A use case describes the **interaction** of **all involved actors** leading to an **end result** of the system
  - Every use case has **pre-conditions** that have to be met the in order to execute the use case (the test case) successfully
  - Every use case have **post-conditions** describing the system after execution of the use case (the test case)
- Use cases are **elements** of the unified modeling language **UML***
  - **Use case diagram** are one of 13 different types of diagrams used by UML
  - A use case diagram is a diagram describing a **behavior**, it does not describe the sequence events
  - It shows the system reaction from the **viewpoint of user**

Use Case based testing: Use Case Diagram



The diagram on the left describes the functionality of a simple restaurant system. Use case are represented by **ovals** and the actors are represented by **stick** figures. The customer actor can eat food. Pay for food or drink juice
Only the chief can prepare food. Note that both the customer and the cashier are involved in the pay for food use case.
The box defines the boundaries of the restaurant system being modeled, the actors are not

Use Case based testing

- Every use case describes a certain task (user-system-interaction)
- **Use case** descriptions include, but are not limited to:
  - Preconditions
  - Expected results/system behavior
  - Postconditions
- These descriptive elements are also used to define the corresponding **test cases**
- Every **use case** may be used as the basis for a **test case**
- **Every alternative** within the diagram corresponds to a **separate test case**
- Typically, information provided with a use case has **not enough detail** to define the **test case** directly, Additional data is needed (input data, expected results) to make up a test case

FULL STACK

Use Case based testing: Use Case Description

UC-001	Use Case Name	
Dependencies		
Description		
Precondition		
Ordinary Sequence	Step	Action
	1	
	2	
	3	
Postcondition		
Alternative Sequence	Step	Action
	1	
	2	
Comments		

Structure-based or white-box techniques

- **Analytical QA**
  - dynamic
    - Black box
      - Equivalence partitioning
      - Boundary value analysis
      - State transition testing
      - Decision tables
      - Use case based testing
    - Experience-based techniques
    - White box
      - Statement coverage
      - Branch coverage
      - Condition coverage
      - Path coverage
  - Static
    - Review/walkthroughs
    - Control flow analysis
    - Dataflow analysis
    - Computer metrics/analyzer

Structure-based or white-box techniques

- The following techniques will be explained in detail:
  - Statement testing and coverage
  - Branch testing and coverage
  - Condition testing and coverage
  - Path testing coverage

- Remark:

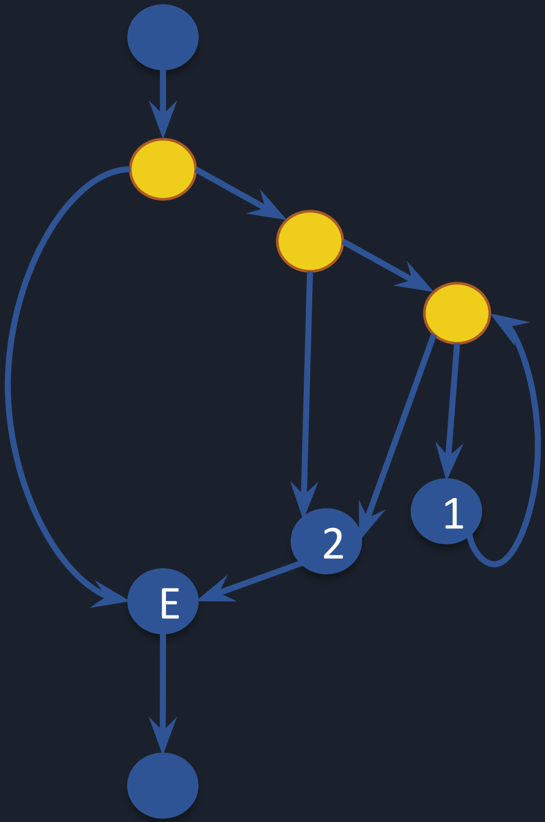
These techniques represent the most important and most widely used dynamic testing techniques. They relate to the static analysis techniques which were described earlier.

The main types of coverage

- **Statement coverage**
  - the percentage of executable statements that have been exercised by the test cases can also be applied to modules, classes, menu points, etc.
- **Decision coverage / branch coverage**
  - the percentage of decision outcomes, that have been exercised by in the test case
- **Condition coverage**
  - the percentage off all single condition outcomes independently affecting a decision outcome, that have been exercised by the test cases
  - Condition coverage comes in various degrees, e.g. single, multiple and minimal multiple condition coverage
- **Path coverage**
  - the percentage of execution paths, that have been exercised by the test cases

Statement Coverage

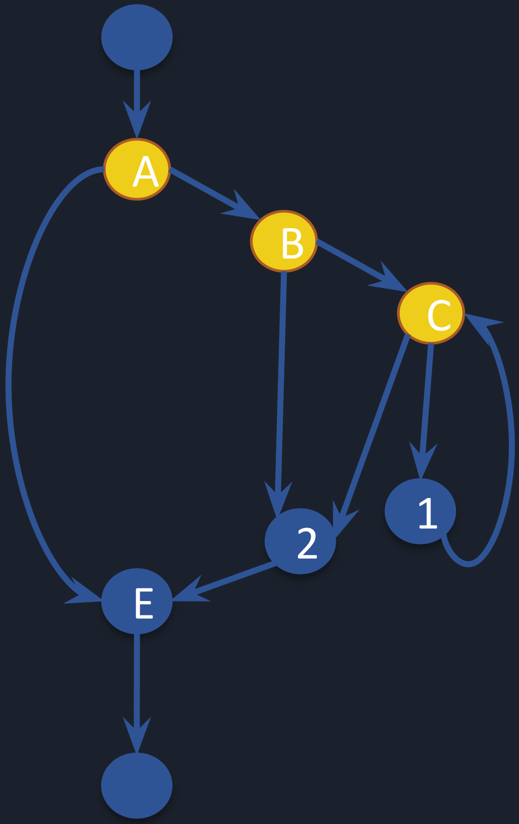
```
If (i>0) {
    If(j>10){
        While (k>10){
            Do 1st task…
        }
    }
    Do 2nd task…
}
Do End task
```



Statement Coverage- Example 1/2

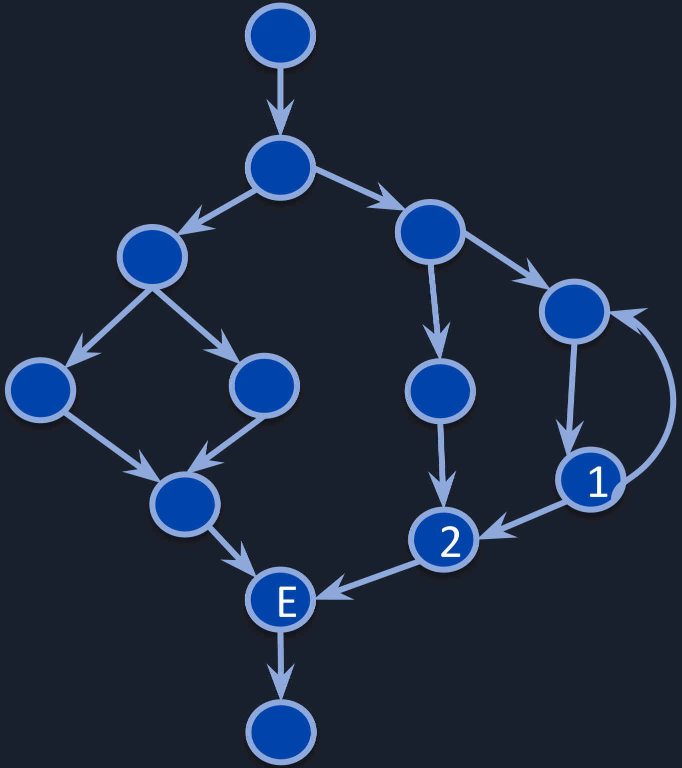
- Consider the program represented by the control flow graph
- Contains two if statements and a loop (do while) inside the second if-statement
- There are three different “routes” through the program statement
  - The first if- statement allows two directions
  - The right hand direction on the first statement is divided again using the second if- statement
  - All statements of this program can be reached using the rout to the right

1->A->B->C->1->C->2->E->z



Statement coverage- Example 2

- In this example the graph is slightly more complex
- The program contains the if statements and loop (inside one if statement)
- Four different “routes” lead trough this program segment
- The first if statements allows two directions
- In both branches of the if statements another if-statement allows the again two different directions



Statement coverage

FULL STACK

- Benefits/drawbacks of this method:
- **Dead code**, that is, code made up of statements that are never executed, will be discovered
- If there is dead code within the program, a 100% coverage cannot be achieved
- **Missing instructions**, that is, code which is necessary in order to fulfill the specification, cannot be detected
- Testing is only done with respect to the executed statements: can all code be reached/executed?
- Missing code cannot be detected using white box test techniques

$$\text{Statement coverage} = \frac{\text{Number of executed Statement}}{\text{Total number of all Statement}} \times 100\%$$

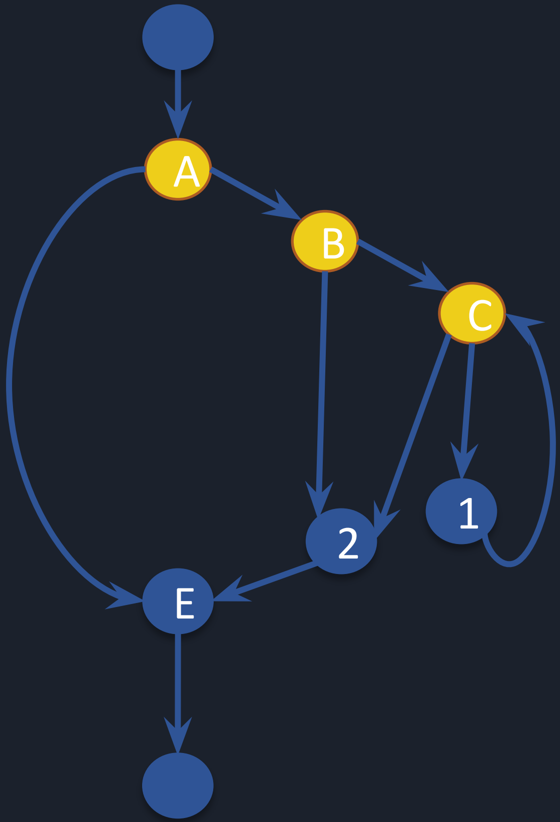
Decision coverage

- Instead of statements, decision coverage focuses on the control flow with a program segment (not the nodes, but the edges of a control flow graph)
- All **edges** of control flow graph have to be coverage **at least once**
- Which test cases are necessary to cover each edge of the control flow graph at least once?
- Aim of this test (test exit criteria) is to achieve the coverage of a selected percentage of all decisions, called the decision coverage

$$\text{Decision coverage} = \frac{\text{Number of executed decisions}}{\text{Total number of all decisions}} \times 100\%$$

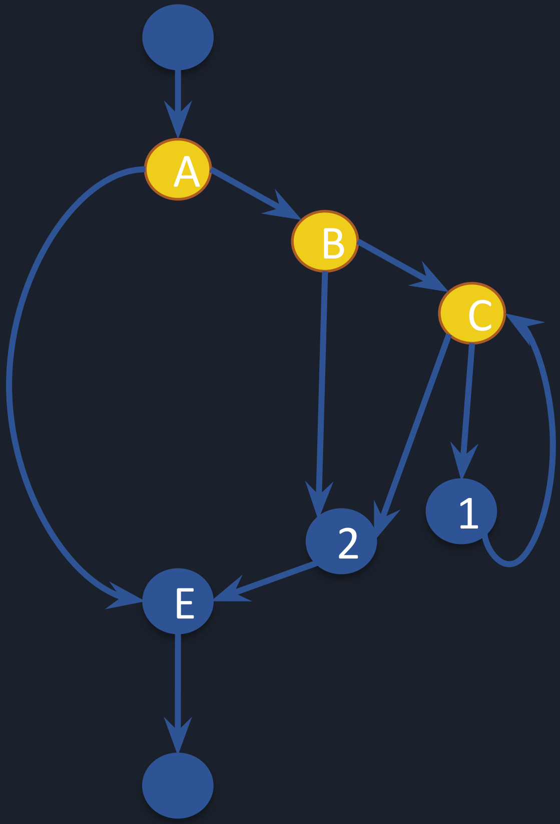
Decision Coverage

```
If (i>0) {
    If(j>10){
            While (k>10){
                Do 1st
            task…
            }
    }
    Do 2nd task…
}
Do End task
```



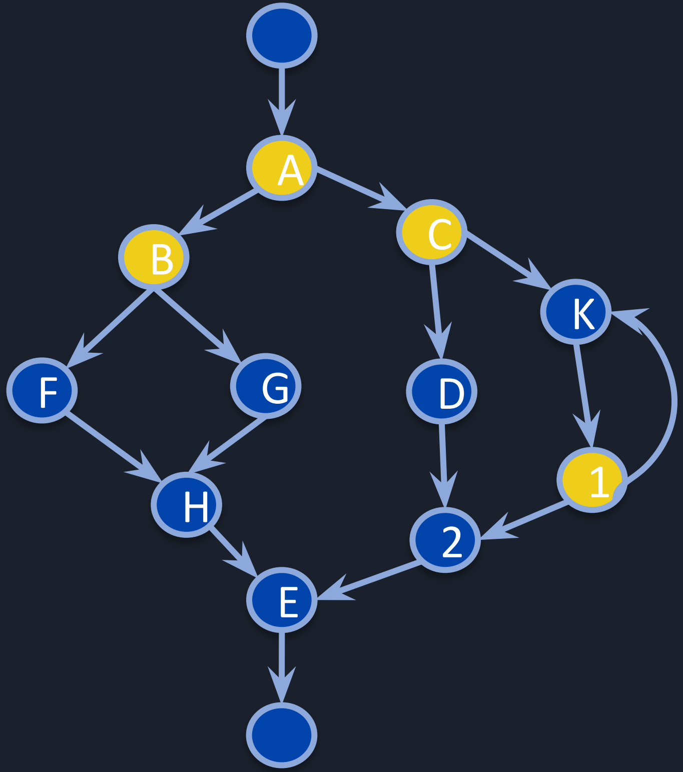
Decision coverage- Example 1

- The control flow graph on the right represents the program segment to be inspected
- Three different "routes" lead through the graph of this program segment
  - The first if statements leads onto two different directions
  - One path of the first if-statement is divided again in two different paths, one of which holds a loop
  - All edges can only be reached via combination of the three possible paths



Decision coverage- Example 2

- In this example the graph is slightly more complex
- Four different "routes" lead through this program segment
  - The first if-statement allows two directions
  - In both branches of the if- statement allows again for two different directions
  - in this example, the loop is not counted as on additional decision
  - In this example, the same set of test cases is also required for 100% statement coverage!



Decision coverage

- Achieving 100% decision coverage requires at least as many test cases as 100% statement coverage – In most cases more
- **a 100% decision coverage always includes a 100% statement coverage!**
- In most cases edges are covered multiple times

Condition Coverage

- The complexity of a condition that is made up of several atomic conditions is taken into account
- An atomic condition can not be divided further into smaller condition statements
- This method aims at finding defects resulting from the implementation of multiple conditions (combined conditions)
- Multiple conditions are made up of atomic conditions, which are combined using logical operators like OR, AND, XOR, etc.
- Atomic conditions do not contain logical operators but only relational operators and the NOT operator (=, >, < etc.)
- There are three types of condition coverage
  - simple condition coverage
  - multiple condition coverage
  - minimal multiple condition coverage

Simple condition coverage

- Every atomic sub-condition of combined condition statement has to take at least once the logical values true / false
- This example is used to explain condition coverage, using multiple condition expression
- With only two test cases, a simple condition coverage can be achieved
- Each sub condition has taken on the value true and the value false
- However, the combined result is **true in both cases**
  - **true OR false = true**
  - **false OR true= true**

Consider the following condition
a>2 OR b<6
Test cases for simple condition Coverage could be for example

a=3 (true)	b=7 (false)	a>2 OR b<6 (true)
a=1 (false)	b=5 (true)	a>2 OR b<6 (true)

FULL STACK

Multiple condition coverage

- All combinations that can be created Using permutation of the atomic sub conditions be part of the test
- This example is used to explain condition coverage using a multiple condition expression
- With four test cases, the multiple condition coverage can be achieved
- All possible combinations of true and false were created
- All possible results of the multiple conditions were achieved
- The number of test cases increase exponentially

  - n= number of atomic conditions

  - 2^n=number of test cases

Consider the following condition

a>2 OR b<6

Test cases for simple condition Coverage could be for example

a=3 (true)	b=7 (false)	a>2 OR b<6 (true)
a=3 (true)	b=5 (true)	a>2 OR b<6 (true)
a=1 (false)	b=5 (true)	a>2 OR b<6 (true)
a=1 (false)	b=7 (false)	a>2 OR b<6 (false)

FULL STACK

Minimal multiple condition coverage

All combinations that can be created using the logical results of the sub conditions must be part of the test, only if the change of the outcome of one sub-condition changes the result of the combined condition

Example IV/02-6
Consider the following condition
a>2 OR b<6
Test cases for simple condition
Coverage could be for example

a=3 (true)	b=7 (false)	a>2 OR b<6 (true)
a=3 (true)	b=5 (true)	a>2 OR b<6 (true)
a=1 (false)	b=5 (true)	a>2 OR b<6 (true)
a=1 (false)	b=7 (false)	a>2 OR b<6 (false)

- This example is used to explain condition coverage using a multiple condition expression
- For three out of four test cases the changes of a sub-condition changes the overall result
- **Only for case no.2** (true OR true=true) the change of a sub condition will not result in a change of the overall condition. This test case can be omitted!

Condition coverage- general conclusion

- The **simple condition coverage** is a week instrument for testing multiple conditions
- The **multiple condition coverage** is a much better method
  - It ensures statement and decision coverage
  - However, it results in a high number of test cases: **2^n**
- The **minimal multiple condition coverage** is even better, because
  - It reduces the number of test cases
  - Statement and decision coverage are covered as well
  - Takes into account the complexity of decision statements
- **All complex decisions must be tested- the minimal multiple condition coverage is a suitable method to achieve this goal**

Path coverage

- Path coverage focuses on the execution of **all possible paths** through a program
- A path is a combination of program segments (in a control flow graph: an alternating sequence of nodes and edges)
- For decision coverage, a single path through a loop is sufficient. For path coverage , there are additional test cases:
    - **One test case not entering the loop**
    - **One additional test case for every number of loop executions**
- This may easily lead to a very **high** number to test cases

Path coverage

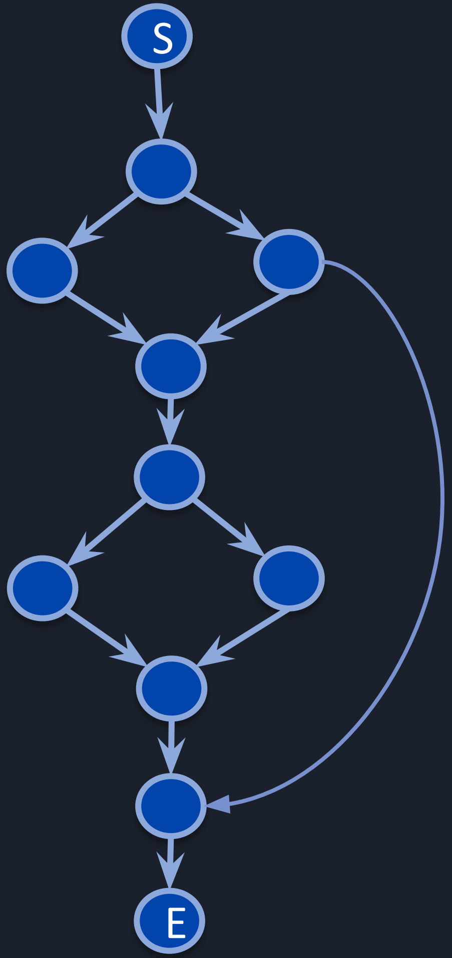
- Focus of the coverage analysis is the control flow graph:
  - Statements are nodes
  - Control flow is represented by the edges
  - Every path is a unique way from the beginning to the end of the control flow graph
- The aim of this test (test exit criteria) is to reach a defined path coverage percentage

$$\text{Path coverage} = \frac{\text{Number of executed Path}}{\text{Total number of all Path}} \times 100\%$$

Path coverage- Example 1

Example:

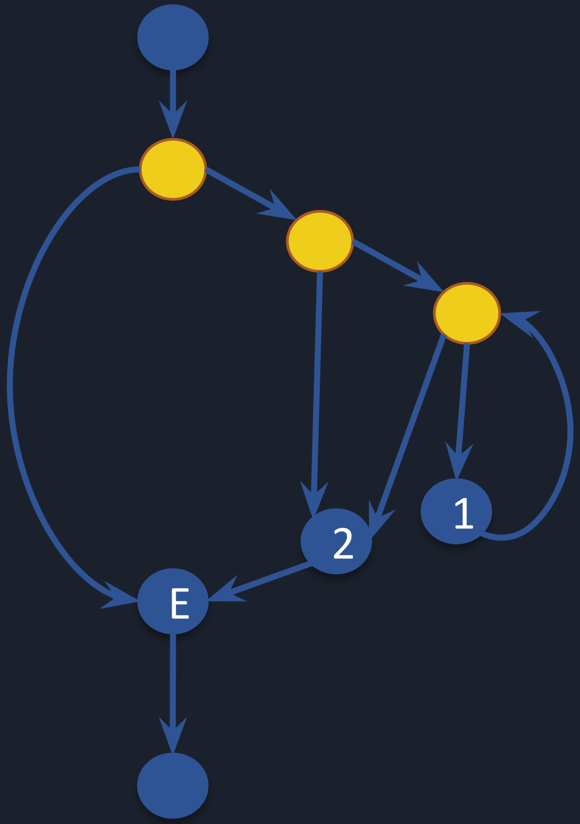
- Three different paths leading through the graph of this program segment achieve full decision coverage
- However, five different possible paths may be executed



Path coverage- Example 2

Example:

- The control flow graph on the right represents the program segment to be inspected. It contains two if-statements and a loop inside the second if-statement
- **Three different paths** leading through the graph of this program segment achieve full decision coverage
- **Four different paths** are possible, if the loop is **executed twice**
- **Every increment of the loop counter adds a new test case**



FULL STACK

Path coverage- general conclusions

- 100% path coverage can be achieved for very simple programs possible number of loop executions constitute a new test case
- A single loop can be loaded to test case explosion because every possible number of loop executions constitute a new test case
- Theoretically an infinite number of paths is possible
- Path coverage is much more comprehensive than statement or decision coverage
- Every possible path through the program is executed
- **100% path coverage includes 100% decision coverage, which again contains 100% statement coverage**

FULL STACK

Thank You