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

- Reviews/ walkthroughs
- Control flow analysis
- Data flow analysis
- Compiler metrics/ analysis

Structure-based or white-box techniques

- The following techniques will be explained in detail:
  - **Statement testing and coverage**
  - **Branch testing and coverage**
  - **Decision 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

- **Path coverage**

  - the percentage of execution paths, that have been exercised by the test cases

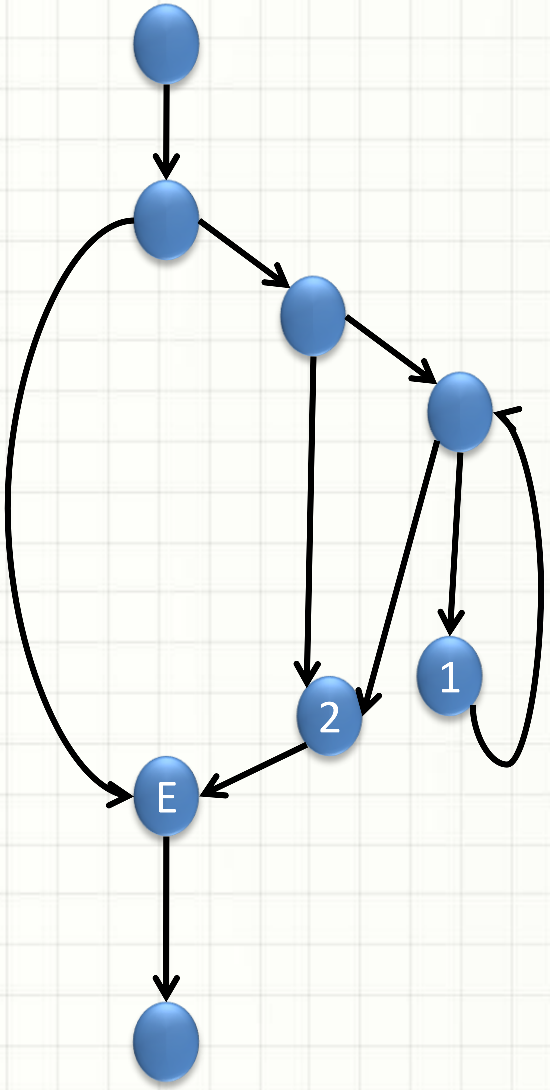
- **Condition coverage**

  - 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

Statement Coverage:

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



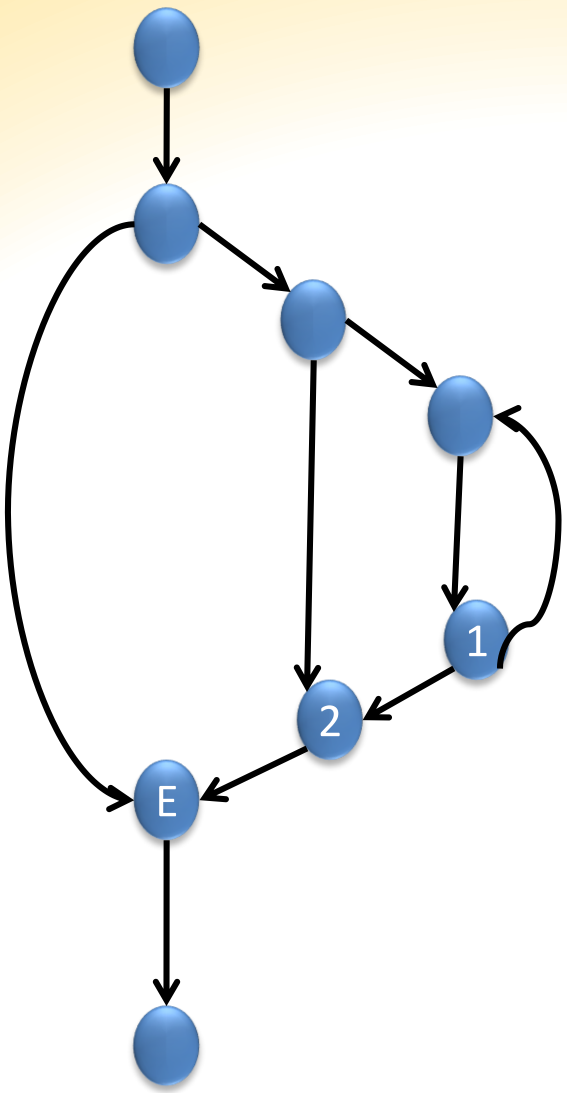
For this Statement 1 test case is needed

Statement coverage

- Example:

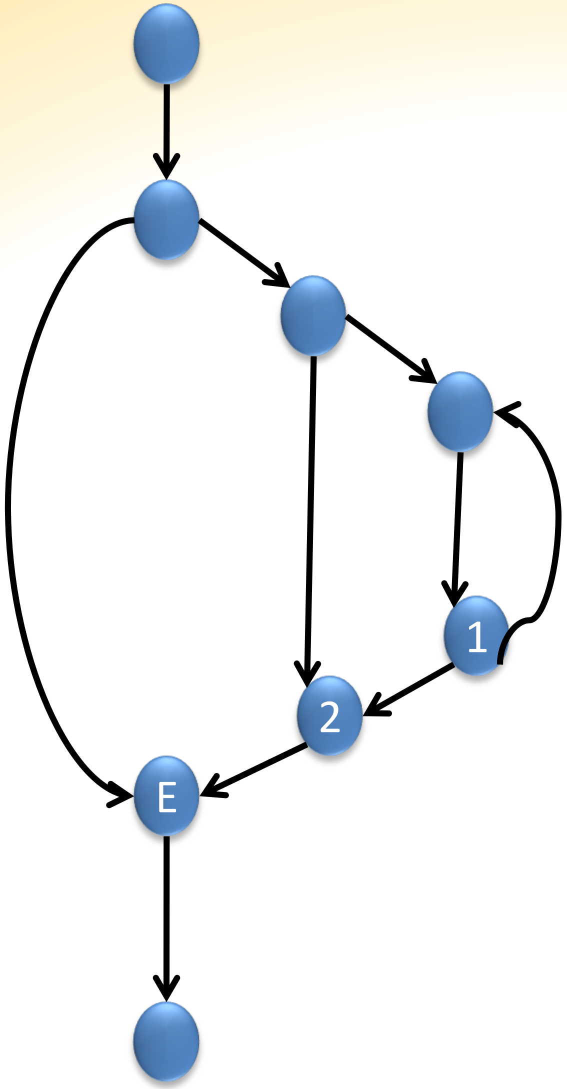
  -We are assessing the following segments of program code, which is represented by the control flow graph (see right side):

```
if( i  > 0 )
  {
      j=f (i);
      if(j>10)
        {
            while(k>10)
           {
           }
         }
  }
```



Statement Coverage- Example 1/2

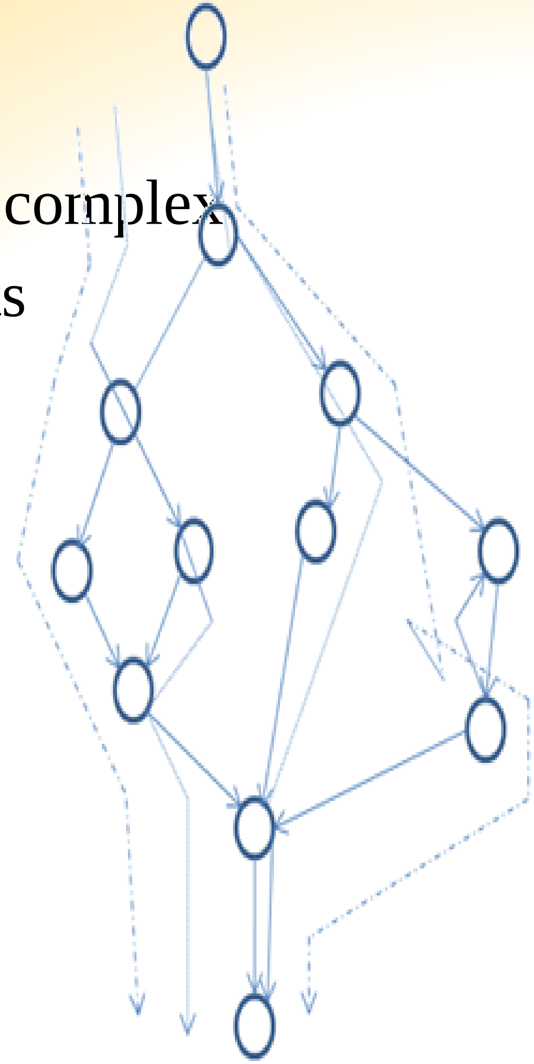
- Consider the program represented by the control flow graph on the right
  - Contains two if statements and a loop (do while) inside the second if-statement
- There are three different "routes" through the program statement
  - The first if- statement allows two directions
  - The right hand direction 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



- A single test case will be enough to reach 100% statement coverage

Statement coverage- Example 2

- Example IV/02-2
- 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



- For a 100% statement coverage, four test cases are needed

Statement coverage

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

$$\text{Statement coverage(C0)} = \frac{\text{Number of executed Statement}}{\text{Total number of all Statement}} * 100\%$$

<u>Decision coverage</u>

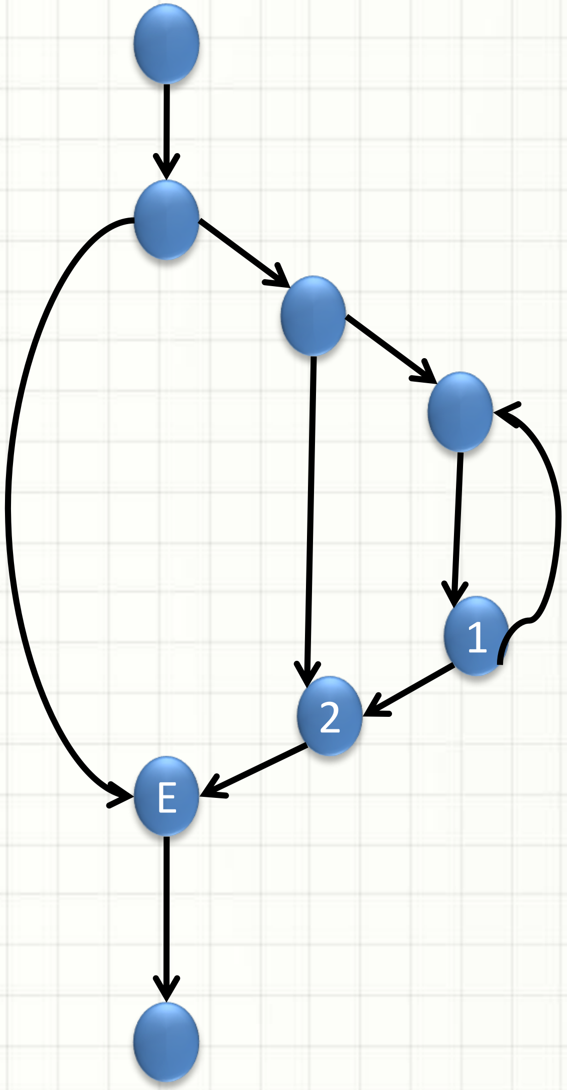
- Instead of statements, decision coverage focuses on the control flow with a program segment (not the needs, but the edges of a control flow graph)
  - All **edges** of control flow graph have to be coverage **at least once**
  - Which test cases are necessary to cover each edge of the control flow 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

$$\text{Decision coverage(C1)} = \frac{\text{Number of executed decisions}}{\text{Total number of all decisions}} * 100\%$$

Branch Coverage:

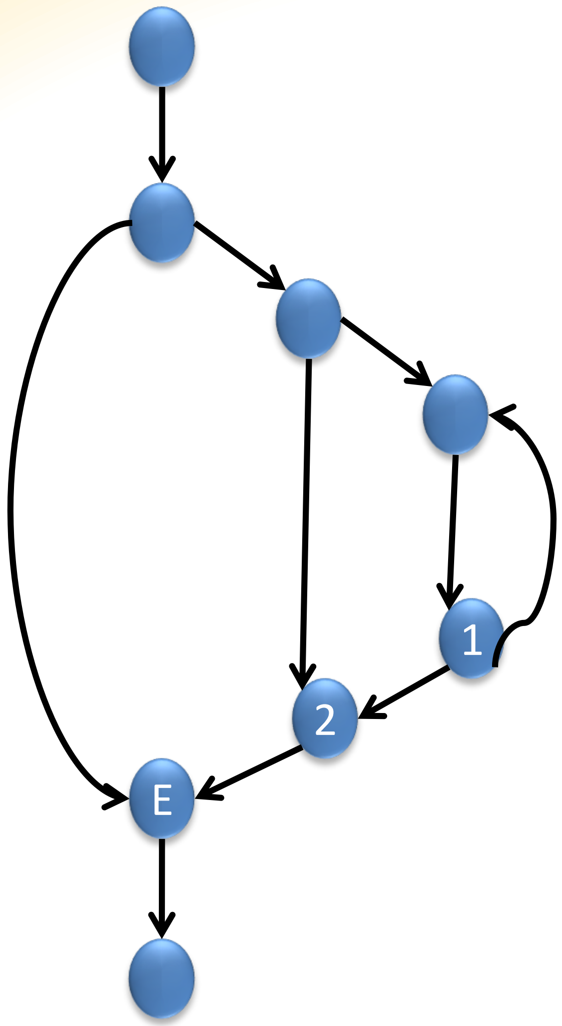
```
If (i>0)
{
    If(j>10)
    {
        While (k>10)
        {
        Do 1st task…
        }
    }
    Do 2nd task…
}
Do End task
```



Considering BC there are 3 test case is needed

Decision coverage- Example 1

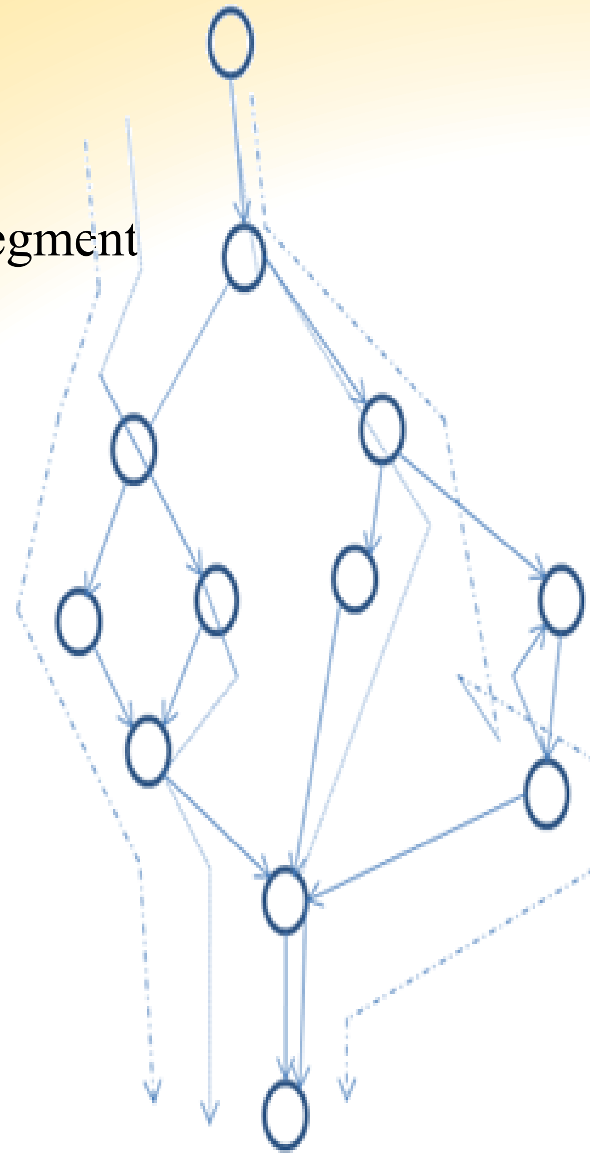
- The control flow graph on the right represents the program segment to be inspected
- Three different "routes" lead through the graph of this program segment
  - The first if statements leads onto two different directions
  - One path of the first if-statement is divided again in two different paths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



- Three test cases are needed to achieve a decision coverage of 100%

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
  - **For a 100% decision coverage four test cases are needed**
  - 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

- Drawbacks
  - **Missing** statement cannot be detected
  - Not sufficient to test complex conditions
  - Not sufficient to test **loops** extensively
  - No consideration of dependencies between loops

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 as well as false

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

- 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**

Multiple condition coverage

- All combinations that can be created Using permutation of the atomic sub conditions be part of the test

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
- 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

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**
  - Some combination may not be possible execute
    - e.g. for x>5 AND x<10 both sub conditions cannot be false at the same time
- 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 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

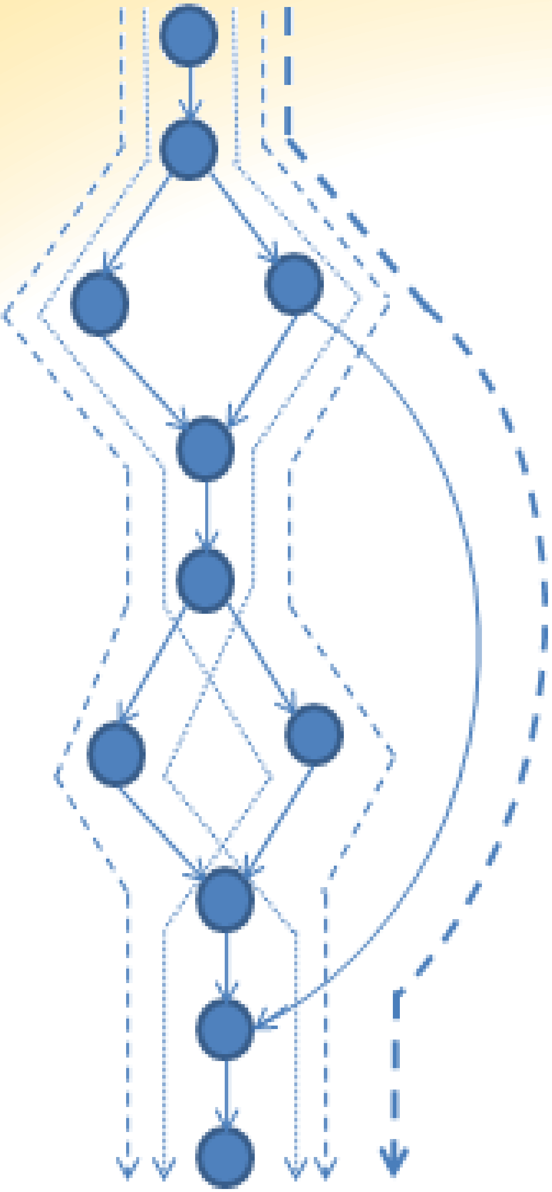
Path coverage

- 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

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

Path coverage- Example 1

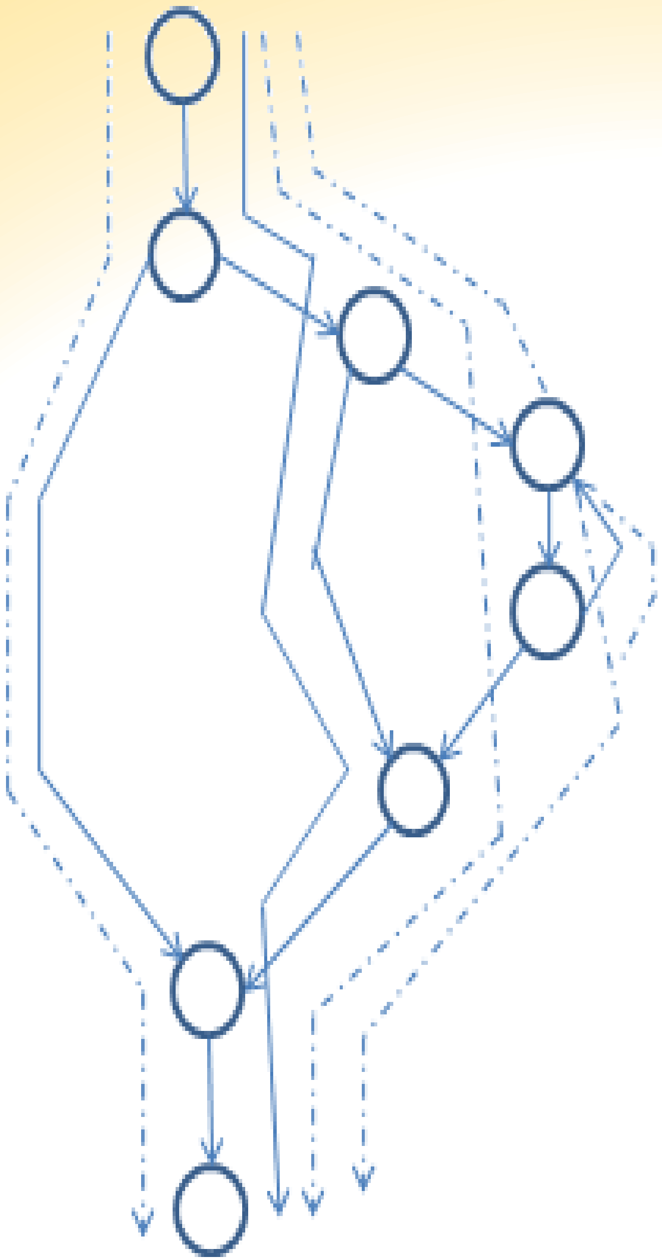
- **Example IV/02-5:**
- The control flow graph on the right represents the program segment to be inspected II contains three statements
- Three different paths leading through the graph of this program segment achieve full decision coverage
- However, five different possible paths may be executed
    - **Five test cases are required to achieve 100% path coverage**
    - **Only two are needed for 100% C0-, three are needed for 100% C1- coverage**



Path coverage- Example 2

Example IV/02-6:

- 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**



Path coverage- general conclusions

- 100% path coverage can be achieved for very simple programs possible number of loop executions constitutions a new test case
  - A single loop can be load to test case explosion because every possible number of loop executions constitutions a new test case
  - Theoretically an identifinite 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**

Experienced-based techniques

Definition of experience- based techniques

Practice of creating test cases without a clear Methodical approach, based on the intuition (সুচতুর অনুমান, স্বত:লব্ধ জ্ঞান) and experience of the tester

- Test cases are based on intuition and and experience
  - Where have errors accumulated in the past?
  - Where does software often fail

Fundamentals

- Experience based testing is also called **intuitive testing** and includes: error guessing (weak point oriented testing) and exploratory testing (iterative testing based on gained knowledge about the system)
- Mostly applied in order to **complement** other, more **formally created test cases**
  - **Does not** meet the criteria for **systematical** testing
  - Often produce **additional test cases** that might not be created with other practices, for example
    - Testing a leap year after 2060 (known problems of the past)
    - Empty sets within input values (a similar application has had errors on this)

<u>**Test case design**</u>

The tester must dispose of applicable experience or knowledge

- **Intuition-** Where can **errors** be hiding**?**
  - Intuition characterizes a good tester
- **Experience-** What **error** were **encountered** where in the past?
  - Knowledge based on experience
  - An alternative is to set up a list of recurring errors
- **knowledge/Awareness-** Where are specific errors expected?
  - Specific **details of the project** are incorporated
  - Where will errors be made due to **time pressure** and **complexity**?
  - Are **inexperienced** programmers involved?

Intuitive test case design – possible sources

- Test result and practical experience with similar systems
  - Possibly a predecessor of the software or other system with similar functionality
- User experience
  - Exchange of experience with the system as a user
- Focus of deployment
  - What parts of the system will be used the most?
- Development problems
  - Are there any weak points out of a difficulties in the development process?

Error guessing in practice

- Check error lists
  - List possible errors
  - Weight factors depending on risk and probability of occurrence
- Test case design
  - Creating test cases aimed at producing the errors on the list - Prioritizing test cases by their risk value
- Update the error list along testing
  - Iterative procedure
  - A structured collection of experience is useful when repeating the procedure in future projects

Explorative testing

- Test case design procedure especially suitable when the information basis is weakly structured
- Also useful when time for testing is scarce
- Procedure:
  - Examine the single parts of the test object
  - Execute few test cases, exclusively on the parts to be tested applying error guessing
  - Analyze results, develop a rough model of how the test object
  - Iteration: Design test objects applying the knowledge functions recently acquired
  - Thereby focusing on conspicuous areas and on exploring further characteristics of the test object
  - Capture tools may be useful for logging test activities

Explorative testing principles

- Choose small objects and/or concentrate on particular aspects of a test object –
  - A single iteration should not take more than 2 hours
- The results of one iteration form the information basis of the following iteration
  - Additional test cases are derived from the particular test situation
- Modeling takes place during testing
  - A Model of the test object is created along testing
  - A test goal is the continuous refinement of the model
- Preparing further tests –
  - Herewith, knowledge can be gained to support the appropriate choice of test case design methods

Intuitive test case design versus systematic test case design

- Intuitive test case design is a good complement to systematical approaches
  - It should still be treated as a complementary activity
  - It cannot give proof of completeness-the number of test cases can vary considerably
- Test are executed in the same way as with systematically defined test cases
- The difference is the way in which the test cases were designed /identified
- Through intuitive testing defects can be detected that may not be found through systematical testing methods

Summary

- Experience based techniques complement systematical techniques to

  determine test cases
- They depend strongly on the **individual ability** of the tester
- **Error guessing** and **Explorative testing** are two of the more widely used techniques of experience based testing

IV- Test Design Techniques

06 -Choosing test techniques

Criteria for choosing the appropriate test case design /1

- State of information about the test object
  - Can white-box tests be made at all (source code available)?
  - Is there sufficient specification material to define black-box tests, or are explorative tests needed to start with?
- Predominant test goals
  - Are functional tests explicitly requested?
  - Which non-functional test are needed?
  - Are structural tests needed to attain the test goals?
- Risk aspects
  - Is serious damage expected from hidden defects?
  - How high is the sequence of usage for the test object?
  - Are there contractual or legal standards on test execution and test coverage that have to be met?

IV- Test Design Techniques

06 -Choosing test techniques

Criteria for choosing the appropriate test case design /2

- Project preconditions
  - How much time and who is planed for testing?
  - How high is the risk, that testing might not be completed as planed?
  - Which software development method are used?- What are the weak points of the project process?-
- Characteristics of the test object
  - What possibilities for testing does the test object offer?
  - What is the availability of the test object?
- Contractual and clent requirements –
  - Were there any specific agreements made with the client / originator of the project about the test procedures?
  - What documents ore to be handed over at the time of deployment of the system?

IV- Test Design Techniques

06 -Choosing test techniques

Criteria for choosing the appropriate test case design /3

- Best practice
  - Which approaches have proven to be appropriate on similar structures?
  - What experience was gained with which approaches in the past?
- Test levels
  - At which test levels should tests be done?
- Further criteria should be applied depending on the specific situation!

IV- Test Design Techniques

06 -Choosing test techniques

Different interests cause different test design approaches

- Interest of the project manager
  - To create software of ordered quality
  - meeting time and budget restrictions
- Interests of the client / initiator of the project
  - To receive software of best quality (functionality, reliability usability, efficiency, portability and maintainability)
  - meeting time and budget restrictions
- Interests of the test manager
  - Sufficient and intensive testing / adequate deployment of the required techniques, from the testing point of view
  - To assess the quality level that the project has reached
  - To allocate and use the resources planed for tests in an optimal way

IV- Test Design Techniques

06 -Choosing test techniques

Summary

- Criteria for choosing the appropriate test case design approach:
  - Test basis (Information basis about the test objects)
  - Testing goals (What conclusions are to be reached through testing)
  - Risk aspects
  - Project framework / preconditions
  - Contractual / client requirements