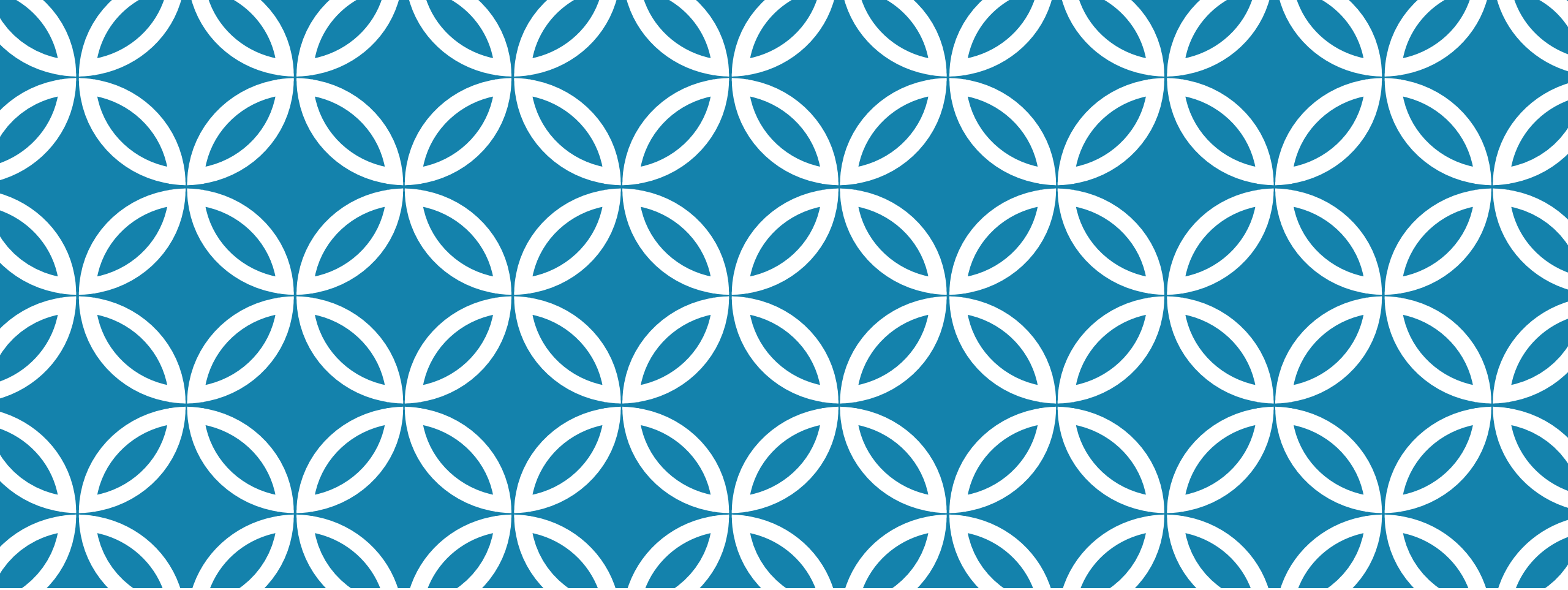
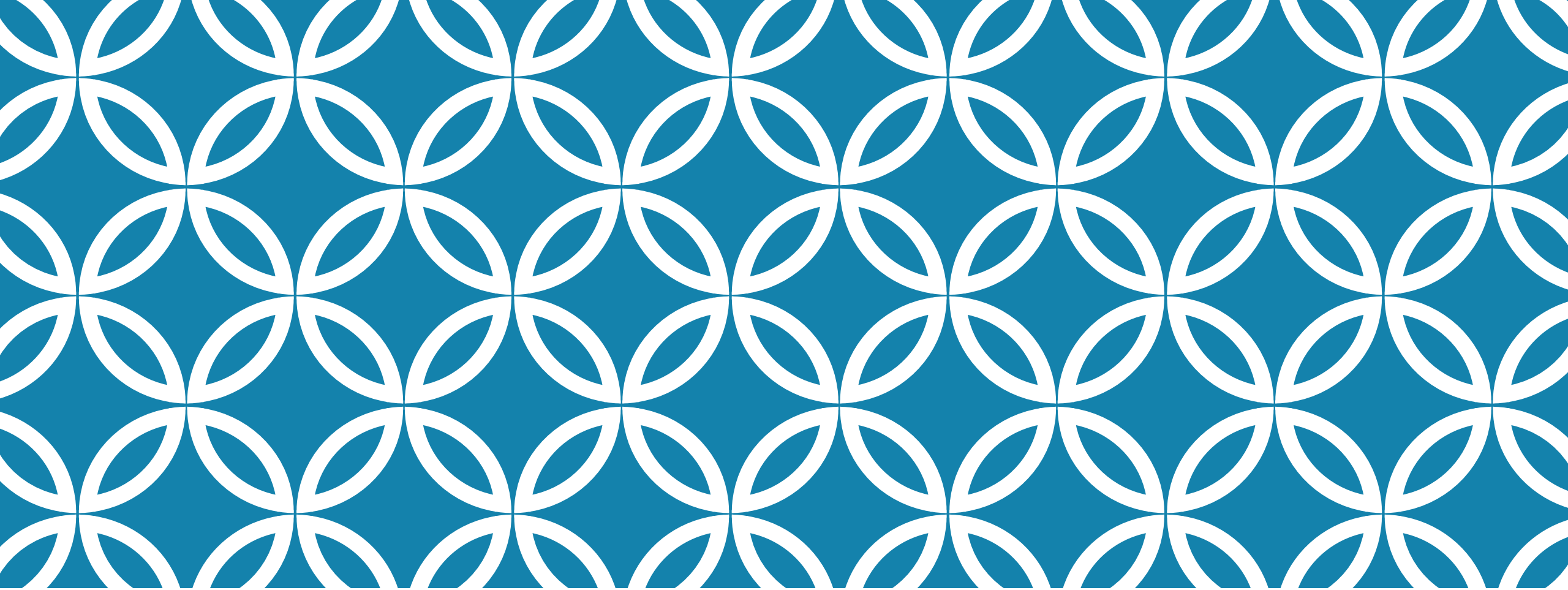


CHAPTER – 1
BASIC CONCEPTS AND PRELIMINARIES

OUTLINE OF CHAPTER

- The Quality Revolution
- Software Quality
- Role of Testing
- Verification and Validation
- Failure, Error, Fault and Defect
- The Notion of Software Reliability
- The Objectives of Testing
- Testing Activities
- Testing Level
- White-box and Black-box Testing
- Test Planning and Design
- Monitoring and Measuring Test Execution
- Test Tools and Automation
- Test Team Organization and Management



HOW TO MEASURE QUALITY?

FOCUS OF QUALITY PROCESS

- Paying much attention to customer’s requirements
- Making efforts to continuously improve quality
- Integrating measurement processes with product design and development
- Pushing the quality concept down to the lowest level of the organization
- Developing a system-level perspective with an emphasis on methodology and process
- Eliminating waste through continuous improvement
- Adhering to deadlines

THE QUALITY REVOLUTION

- Started in Japan by Deming, Juran, and Ishikawa during 1940s
- In 1950s, Deming introduced statistical quality control to Japanese engineers
- Statistical quality control (SQC) is a discipline based on measurement and statistics
  - SQC methods use seven basic quality management tool
    - Pareto analysis, Trend Chart, Flow chart, Histogram, Scatter diagram, Control chart, Cause and effect diagram
- "Lean principle" was developed by Taiichi Ohno of Toyota
  - "A systematic approach to identifying and eliminating waste through continuous improvement, flowing the product at the pull of the customer in pursuit of perfection."

THE QUALITY REVOLUTION

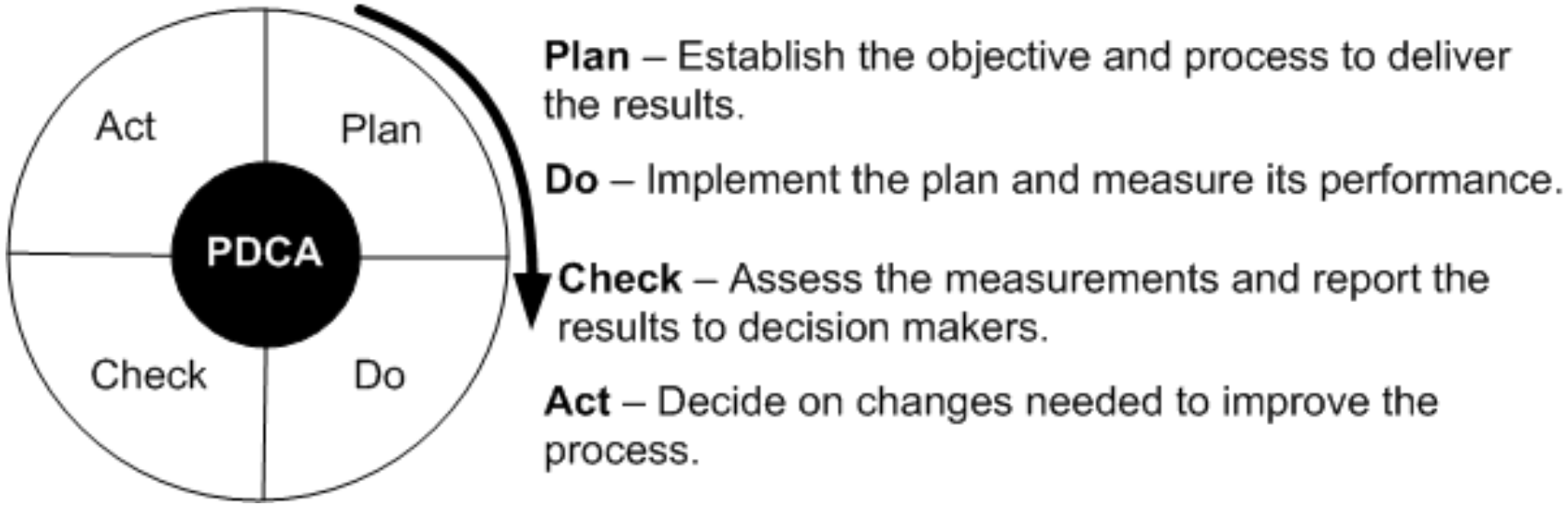


Figure 1: The Shewhart cycle

- Deming introduced Shewhart's PDCA cycle to Japanese researchers
- It illustrates the activity sequence:
  - Setting goals
  - Assigning them to measurable milestones
  - Assessing the progress against the milestones
  - Take action to improve the process in the next cycle

THE QUALITY REVOLUTION

- In 1954, Juran spurred the move from SQC to TQC (Total Quality Control)
- Key Elements of TQC:
  - Quality comes first, not short-term profits
  - The customer comes first, not the producer
  - Decisions are based on facts and data
  - Management is participatory and respectful of all employees
  - Management is driven by cross-functional committees
- An innovative methodology developed by Ishikawa called cause-and-effect diagram

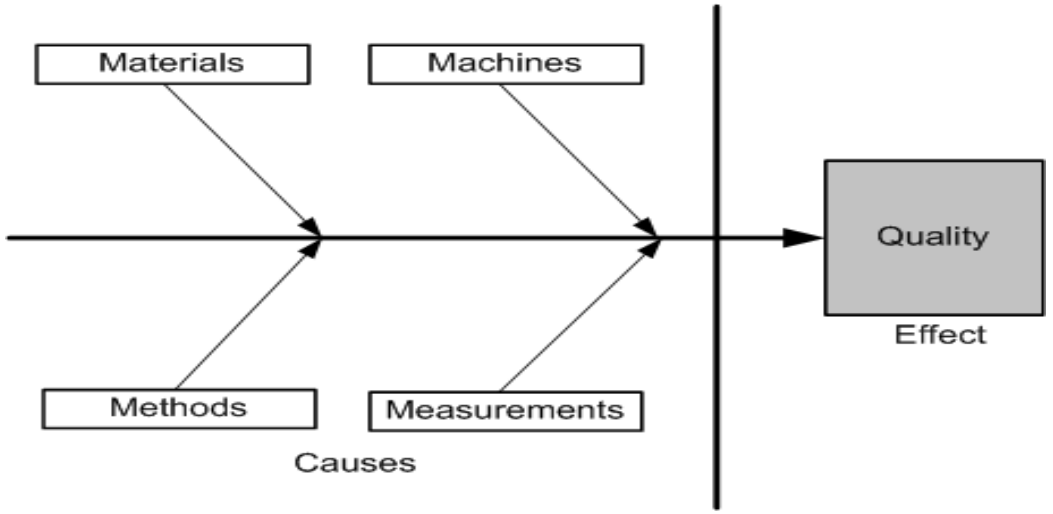


Figure 2: Ishikawa diagram

ISHIKAWA DIAGRAM

- Kaoru Ishikawa found from statistical data that dispersion in product quality came from four common causes, namely *materials*, *machines*, *methods*, and *measurements*, known as the 4 Ms.
- Materials often differ when sources of supply or size requirements vary.
- Machines, or equipment, also function differently depending on variations in their parts, and they operate optimally for only part of the time.
- Methods, or processes, cause even greater variations due to lack of training and poor handwritten instructions.
- Finally, measurements also vary due to outdated equipment and improper calibration.
- Variations in the 4 Ms parameters have an effect on the quality of a product.

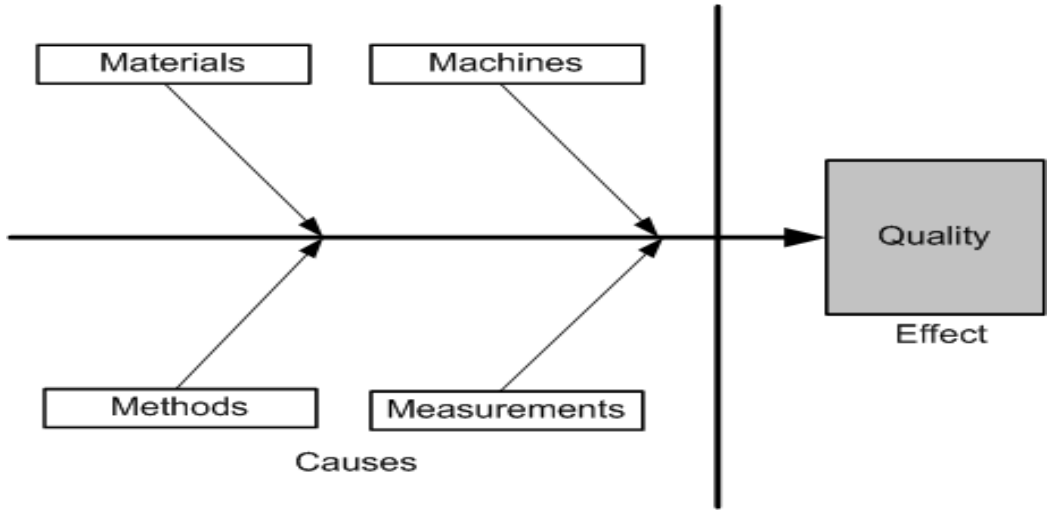


Figure 2: Ishikawa diagram

SOFTWARE QUALITY

- Five Views of Software Quality:
  - Transcendental view
    - It envisages quality as something that can be recognized but is difficult to define.
    - *I know it when I see it.*
  - User's view
    - Does the product satisfy user needs and expectations?
  - Manufacturing view
    - Quality is understood as conformance to the specification.
    - The quality level of a product is determined by the extent to which the product meets its specifications.
  - Product view
    - Quality is viewed as tied to the inherent characteristics of the product. A product's inherent characteristics, that is, internal qualities, determine its external qualities.
  - Value-based view
    - Depends on the amount a customer is willing to pay for it.

SOFTWARE QUALITY

- Software quality in terms of quality factors and criteria
  - A quality factor represents a behavioural characteristic of a system
    - Examples: correctness, reliability, efficiency, and testability
  - A quality criterion is an attribute of a quality factor that is related to software development
    - Example: modularity is an attribute of software architecture
- Quality Models
  - Examples: ISO 9126, CMM, TPI, and TMM

ROLE OF TESTING

- Software quality assessment divide into two categories:
  - Static analysis
    - It examines the code and reasons overall behavior that might arise during run time
    - Identify issues within the logic and techniques
      - Examples: Code review, inspection, and algorithm analysis
  - Dynamic analysis
    - Actual program execution to expose possible program failure
    - Involves running code and examining the outcome, which also entails testing possible execution paths of the code.
    - Static and Dynamic Analysis are complementary in nature
- Focus is to combine the strengths of both approaches

VERIFICATION AND VALIDATION

- Verification
  - Evaluation of software system that helps in determining whether the product of a given development phase satisfies the requirements established before the start of that phase
    - Building the product correctly
    - Are we building the product right?
    - It is static testing
    - It includes checking documents, design, codes and programs.
- Validation
  - Evaluation of software system that helps in determining whether the product meets its intended use
    - Building the correct product
    - Are we building the right product?
    - It is dynamic testing
    - It includes testing and validating the actual product i.e. it includes execution of the code.

FAILURE, ERROR, FAULT AND DEFECT

- Failure
  - A *failure* is said to occur whenever the external behavior of a system does not conform to that prescribed in the system specification
- Error
  - An *error* is a state of the system.
  - An *error* state could lead to a *failure* in the absence of any corrective action by the system
- Fault
  - A *fault* is the adjudged cause of an *error*
- Defect
  - It is synonymous of *fault*
  - It a.k.a. *bug*
- The process of failure manifestation can therefore be succinctly represented as a behavior chain as follows: fault → error → failure.

THE NOTION OF SOFTWARE RELIABILITY

- It is defined as the probability of failure-free operation of a software system for a specified time in a specified environment
- The level of reliability of a system depends on those inputs that cause failures to be observed by the end users.
- It can be estimated via *random testing*
- Test data must be drawn from the input distribution to closely resemble the future usage of the system
- Future usage pattern of a system is described in a form called *operational profile*
-

THE OBJECTIVES OF TESTING

- It does work
  - To find what is working [can be a single unit, integration of units or entire system]
- It does not work
  - To find what is not working
- Reduce the risk of failures
  - Complex system fail from time to time and gives rise to a notion of *failure rate*
- Reduce the cost of testing
  - the cost of designing, maintaining, and executing test cases,
  - the cost of analyzing the result of executing each test case,
  - the cost of documenting the test cases, and
  - the cost of actually executing the system and documenting it.

WHAT IS A TEST CASE?

- Test Case is a simple pair of
  - **<input, expected outcome>**
- State-less systems: A compiler is a stateless system
  - Test cases are very simple
  - A compiler is a stateless system because to compile a program it does not need to know about the programs it compiled previously
  - Outcome depends solely on the current input
- State-oriented: ATM is a state-oriented system
  - Test cases are not that simple. A test case may consist of a sequences of <**input, expected outcome**>
    - The outcome depends both on the current state of the system and the current input
    - ATM example:
      - **< check balance, $500.00 >,**
      - **< withdraw, "amount?" >,**
      - **< $200.00, "$200.00" >,**
      - **< check balance, $300.00 >**

EXPECTED OUTCOME

- An outcome of program execution may include
  - Value produced by the program
  - State Change
  - A sequence of values which must be interpreted together for the outcome to be valid

- A *test oracle* is a mechanism that verifies the correctness of program outputs
  - Generate expected results for the test inputs
  - Compare the expected results with the actual results of execution of the IUT

THE CONCEPT OF COMPLETE TESTING

- Complete or exhaustive testing means
  - “There are no undisclosed faults at the end of the test phase”
- Complete testing is nearly impossible for most of the system
  - The domain of possible inputs of a program is too large
    - Valid inputs
    - Invalid inputs
  - The design issues may be too complex to completely test
  - It may not be possible to create all possible execution environments of the system

THE CENTRAL ISSUE IN TESTING

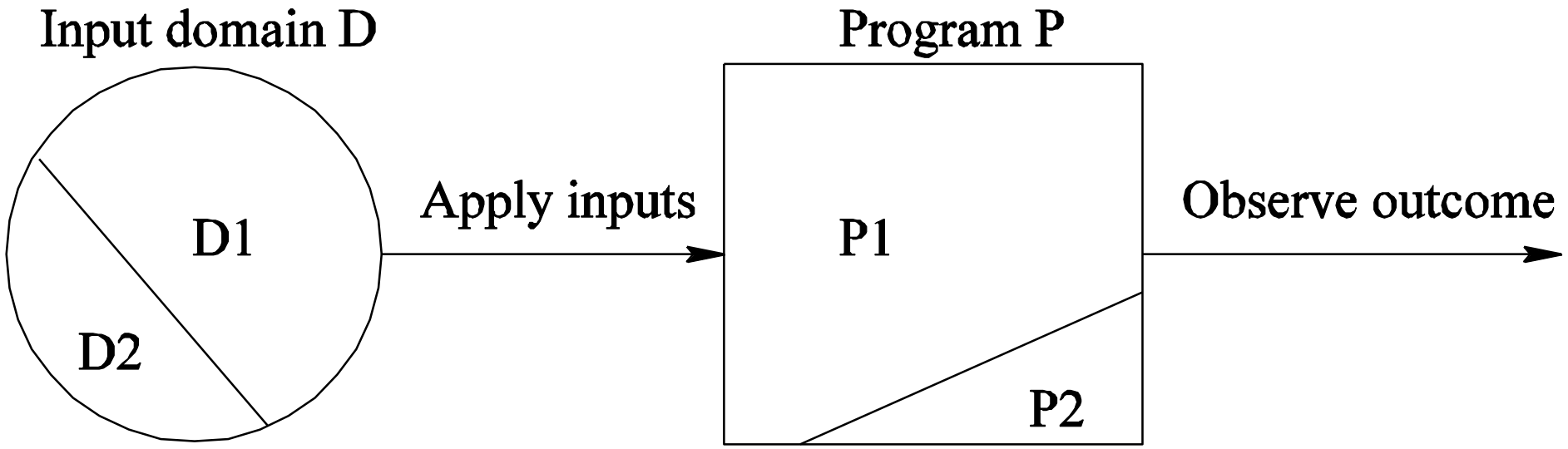


Figure 3: A subset of the input domain exercising a subset of the program behavior

- Divide the input domain D into D1 and D2
- Select a subset D1 of D to test program P
- It is possible that D1 exercise only a part P1 of P

TESTING ACTIVITIES

- Identify the objective to be tested
- Select inputs
- Compute the expected outcome
- Set up the execution environment of the program
- Execute the program
- Analyze the test results

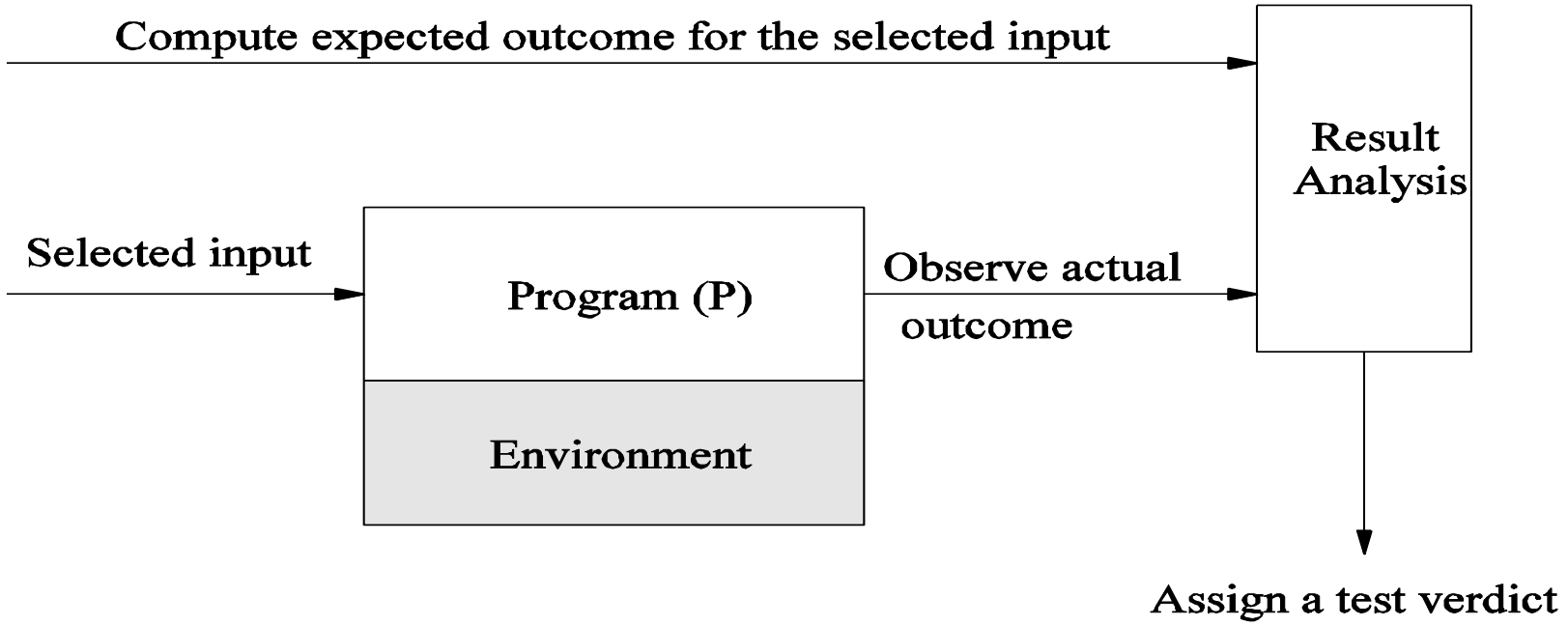


Figure 4: Different activities in process testing

TESTING LEVEL

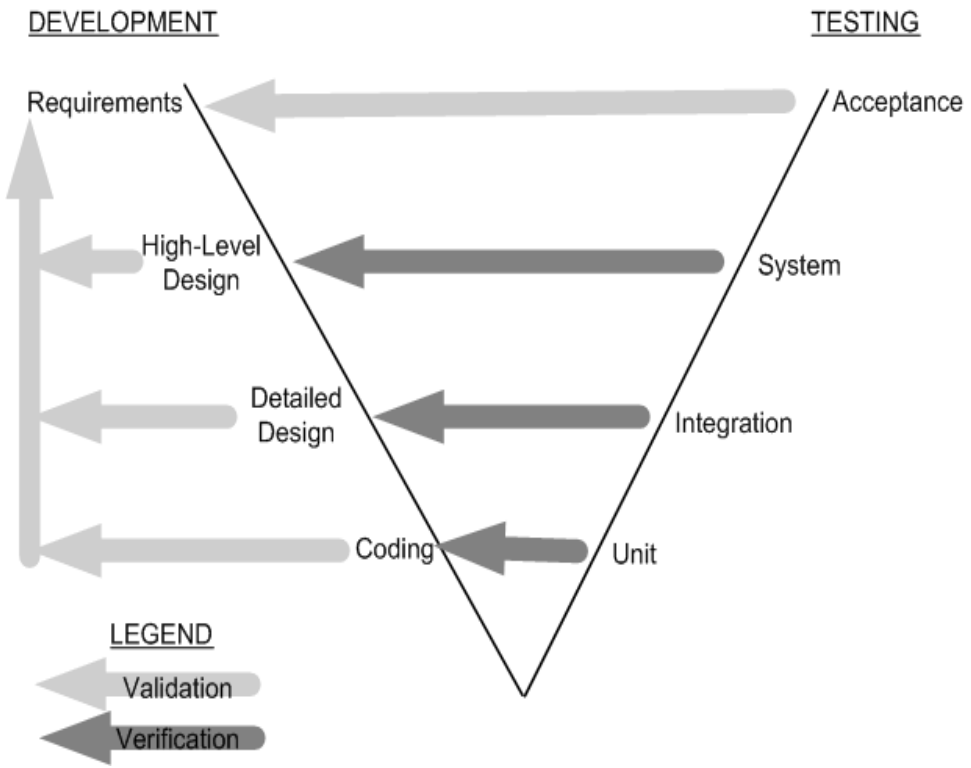


Figure 5: Development and testing phases in the V model

Unit testing

- Individual program units, such as procedure, methods in isolation

Integration testing

- Modules are assembled to construct larger subsystem and tested

System testing

- Includes wide spectrum of testing such as functionality, and load

Acceptance testing

- Customer's expectations from the system
- Two types of acceptance testing
  - User Acceptance Test (UAT)
  - Business Acceptance Test (BAT)
- UAT: System satisfies the contractual acceptance criteria
- BAT: System will eventually pass the user acceptance test

TESTING LEVEL

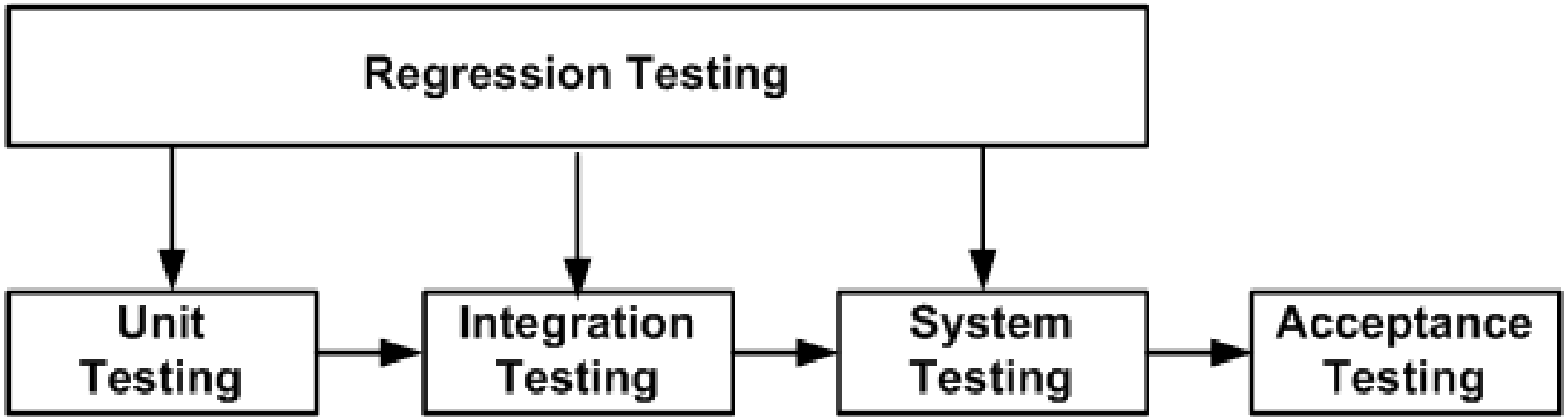


Figure 6: Regression testing at different software testing levels

- New test cases are not designed
- Test are selected, prioritized and executed
- To ensure that nothing is broken in the new version of the software

SOURCE OF INFORMATION FOR TEST SELECTION

- Requirement and Functional Specifications
- Source Code
- Input and output Domain
- Operational Profile
- Fault Model
  - Error Guessing
  - Fault Seeding
  - Mutation Analysis

WHITE-BOX AND BLACK-BOX TESTING

White-box testing a.k.a. **structural testing**

Examines source code with focus on:

- Control flow
- Data flow

Control flow refers to flow of control from one instruction to another

Data flow refers to propagation of values from one variable or constant to another variable

It is applied to individual units of a program

Software developers perform structural testing on the individual program units they write

Black-box testing a.k.a. **functional testing**

Examines the program that is accessible from outside

Applies the input to a program and observes the externally visible outcome

It is applied to both an entire program as well as to individual program units

It is performed at the external interface level of a system

It is conducted by a separate software quality assurance group

TEST PLANNING AND DESIGN

- The purpose is to get ready and organized for test execution
- A test plan provides a:
  - Framework
    - A set of ideas, facts or circumstances within which the tests will be conducted
  - Scope
    - The domain or extent of the test activities
  - Details of resources needed
  - Effort required
  - Schedule of activities
  - Budget
- Test objectives are identified from different sources
- Each test case is designed as a combination of modular test components called test steps
- Test steps are combined together to create more complex tests

MONITORING AND MEASURING TEST EXECUTION

- Metrics for monitoring test execution
- Metrics for monitoring defects
- Test case effectiveness metrics
  - Measure the “defect revealing ability” of the test suite
  - Use the metric to improve the test design process
- Test-effort effectiveness metrics
  - Number of defects found by the customers that were not found by the test engineers

TEST TOOLS AND AUTOMATION

Increased productivity of the testers

Better coverage of regression testing

Reduced durations of the testing phases

Reduced cost of software maintenance

Increased effectiveness of test cases

The test cases to be automated are well defined

Test tools and infrastructure are in place

The test automation professionals have prior successful experience in automation

Adequate budget have been allocation for the procurement of software tools

TEST TEAM ORGANIZATION AND MANAGEMENT

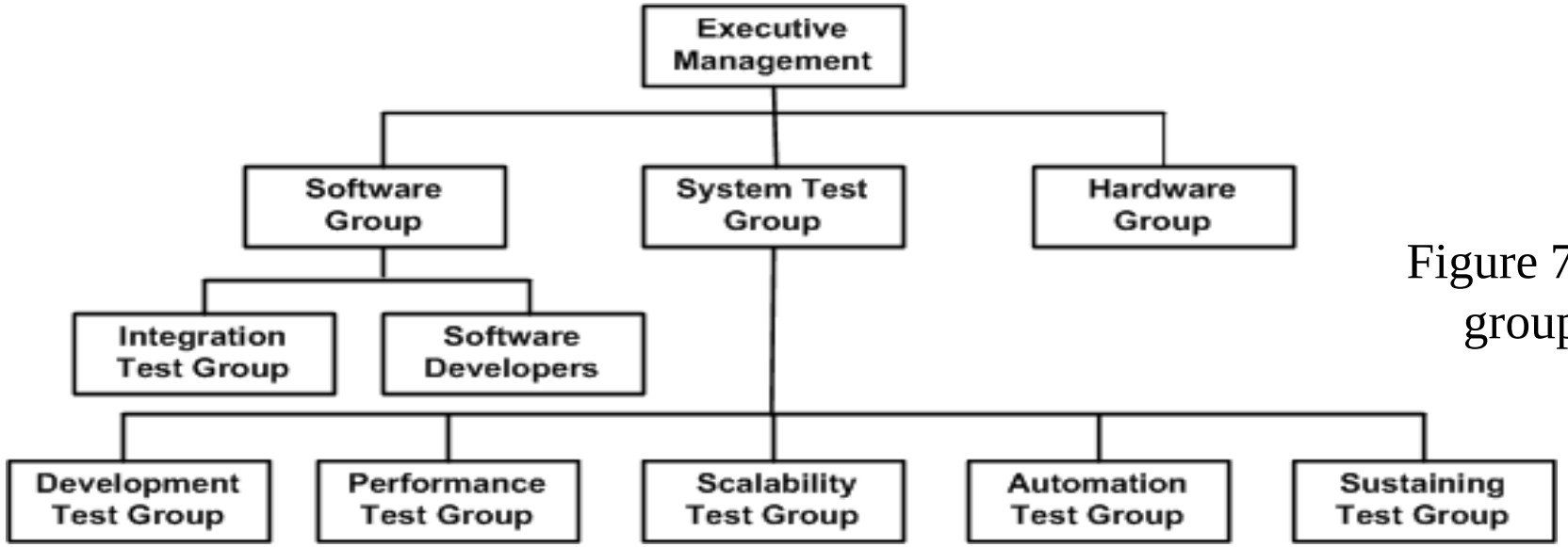
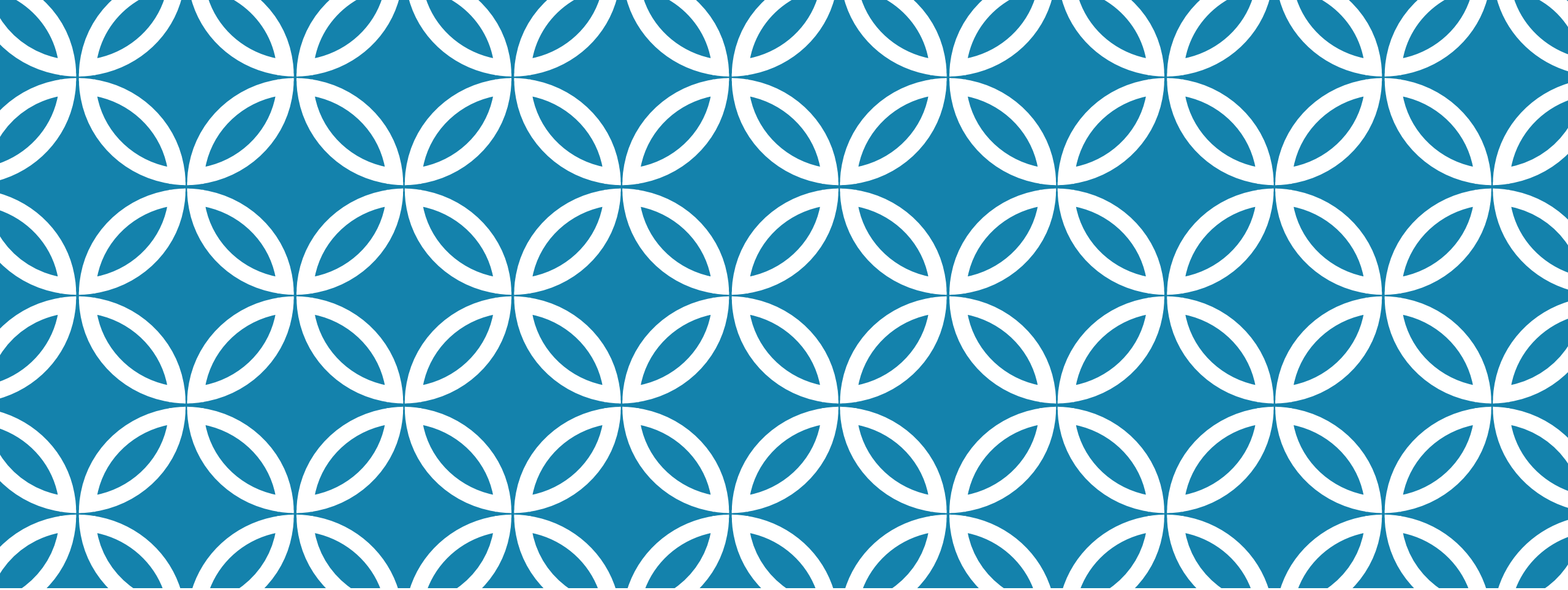
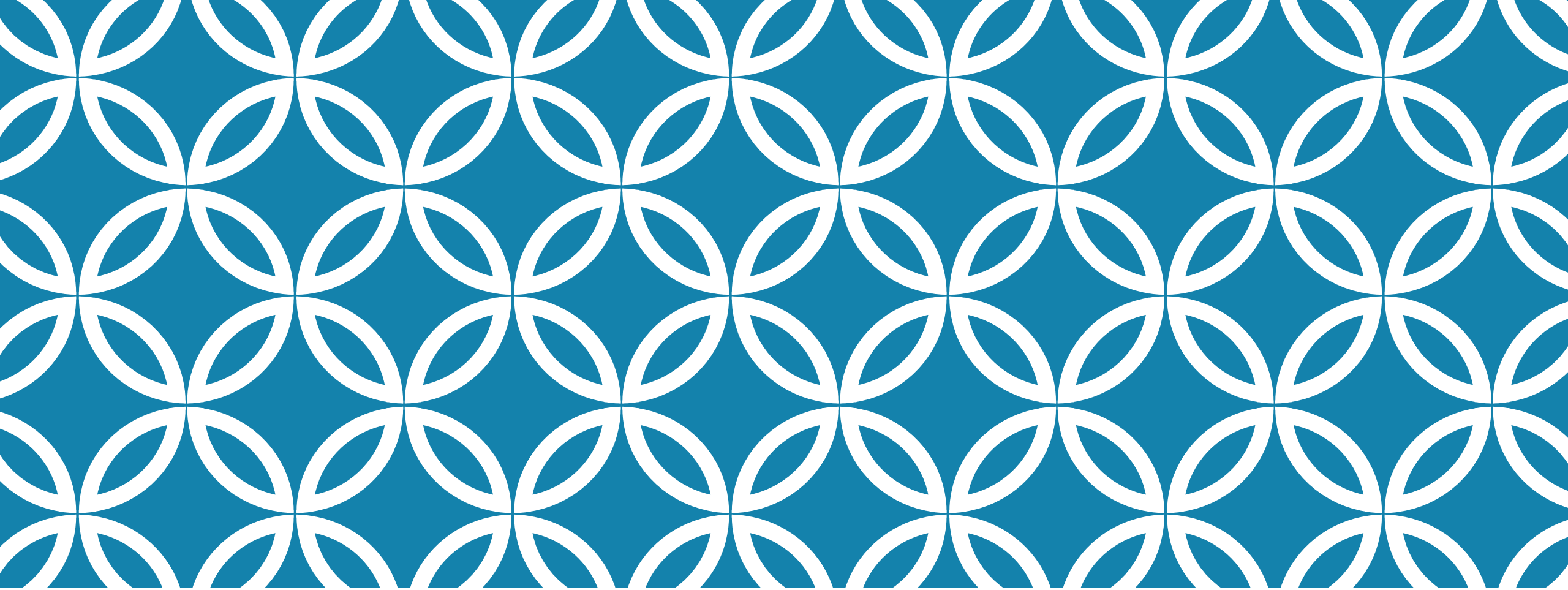


Figure 7: Structure of test groups

- Hiring and retaining test engineers is a challenging task
- Interview is the primary mechanism for evaluating applicants
- Interviewing is a skills that improves with practice
- To retain test engineers management must recognize the importance of testing efforts at par with development effort



THANK YOU!!



CHAPTER – 3
UNIT TESTING

OUTLINE OF THE CHAPTER

- Concept of Unit Testing
- Static Unit Testing
- Defect Prevention
- Dynamic Unit Testing
- Mutation Testing
- Debugging
- Unit Testing in eXtreme Programming
- Tools For Unit Testing

CONCEPT OF UNIT TESTING

- Static Unit Testing
  - Code is examined over all possible behaviors that might arise during run time
  - Code of each unit is validated against requirements of the unit by reviewing the code
- Dynamic Unit Testing
  - A program unit is actually executed and its outcomes are observed
  - One observe some representative program behavior, and reach conclusion about the quality of the system
- Static unit testing is not an alternative to dynamic unit testing
- Static and Dynamic analysis are complementary in nature
- In practice, partial dynamic unit testing is performed concurrently with static unit testing
- It is recommended that static unit testing be performed prior to the dynamic unit testing

STATIC UNIT TESTING

- In static unit testing code is reviewed by applying techniques:
  - **Inspection:** It is a step by step peer group review of a work product, with each step checked against pre-determined criteria
  - **Walkthrough:** It is review where the author leads the team through a manual or simulated executed of the product using pre-defined scenarios
- The idea here is to examine source code in detail in a systematic manner
- The objective of code review is to *review* the code, and *not* to evaluate the author of the code
- Code review must be planned and managed in a professional manner
- The key to the success of code is to divide and conquer
  - An examiner inspect small parts of the unit in isolation
    - nothing is overlooked
    - the correctness of all examined parts of the module implies the correctness of the whole module

STATIC UNIT TESTING (CODE REVIEW)

Step 1: Readiness Criteria

- Completeness
- Minimal functionality
- Readability
- Complexity
- Requirements and design documents

- Roles
  - Moderator
  - Author
  - Presenter
  - Record keeper
  - Reviewers
  - Observer

Step 2: Preparation

- List of questions
- Potential Change Request (CR)
- Suggested improvement opportunities

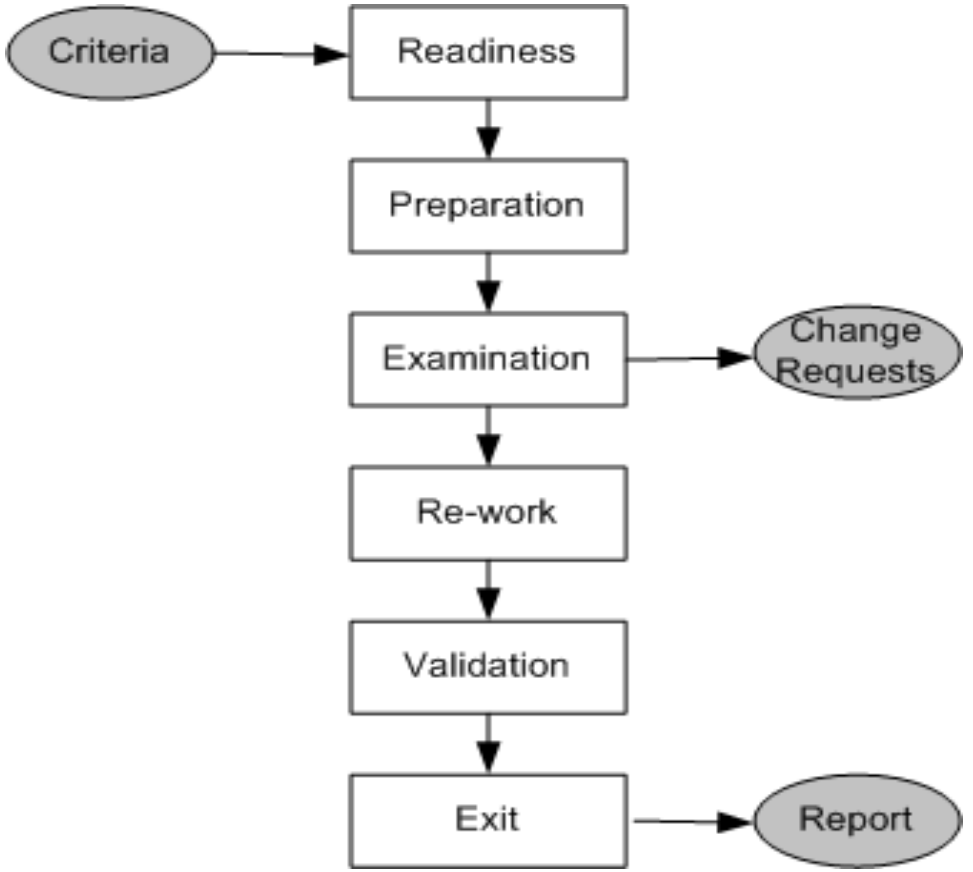


Figure 1: Steps in the code review process

STATIC UNIT TESTING (CODE REVIEW)

- **Step 3: Examination**
  - The author makes a presentation
  - The presenter reads the code
  - The record keeper documents the CR
  - Moderator ensures the review is on track
- **Step 4: Re-work**
  - Make the list of all the CRs
  - Make a list of improvements
  - Record the minutes meeting
  - Author works on the CRs to fix the issue
- **Step 5: Validation**
  - CRs are independently validated
- **Exit**
  - A summary report of the meeting minutes is distributed

A **Change Request (CR)** includes the following details:

- Give a brief description of the issue
- Assign a priority level (major or minor) to a **CR**
- Assign a person to follow it up
- Set a deadline for addressing a **CR**

STATIC UNIT TESTING (CODE REVIEW)

- The following metrics can be collected from a code review:
  - The number of lines of code (LOC) reviewed per hour
  - The number of CRs generated per thousand lines of code (KLOC)
  - The number of CRs generated per hour
  - The total number of hours spend on code review process

STATIC UNIT TESTING (CODE REVIEW)

- The code review methodology can be applicable to review other documents
- Five different types of system documents are generated by engineering department
  - Requirement
  - Functional Specification
  - High-level Design
  - Low-level Design
  - Code
- In addition, installation, user, and troubleshooting guides are developed by the technical documentation group

Hierarchy of System Documents
Requirement: High-level marketing or product proposal.
Functional Specification: Software Engineering response to the marketing proposal.
High-Level Design: Overall system architecture.
Low-Level Design: Detailed specification of the modules within the architecture.
Programming: Coding of the modules.

Table 1: System documents

DEFECT PREVENTION

- Build instrumentation code into the code
- Use standard control to detect possible occurrences of error conditions
- Ensure that code exists for all return values
- Ensure that counter data fields and buffer overflow/underflow is appropriately handled
- Provide error messages and help texts
- Validate input data, such as arguments, passed to a function
- Use assertions to detect impossible conditions
- Leave assertions in the code (activate/deactivate)
- Fully document the assertions that appear to be unclear
- After every major computation, reverse-compute the input(s) from the results in the code itself
- Develop a timer routine
- Include a loop counter within each loop

DYNAMIC UNIT TESTING

- The environment of a unit is emulated and tested in isolation
- The caller unit is known as *test driver*
  - A *test driver* is a program that invokes the unit under test (UUT)
  - It provides input data to unit under test and report the test result
- The emulation of the units called by the UUT are called *stubs*
  - It is a dummy program
- The *test driver* and the *stubs* are together called *scaffolding*
- The low-level design document provides guidance for selection of input test data

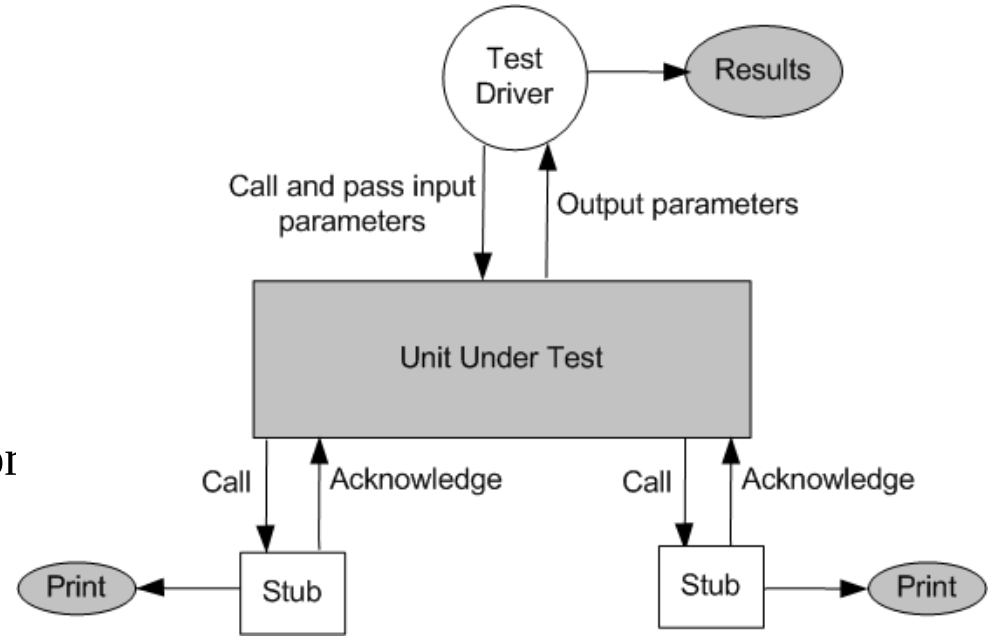


Figure 2: Dynamic unit test environment

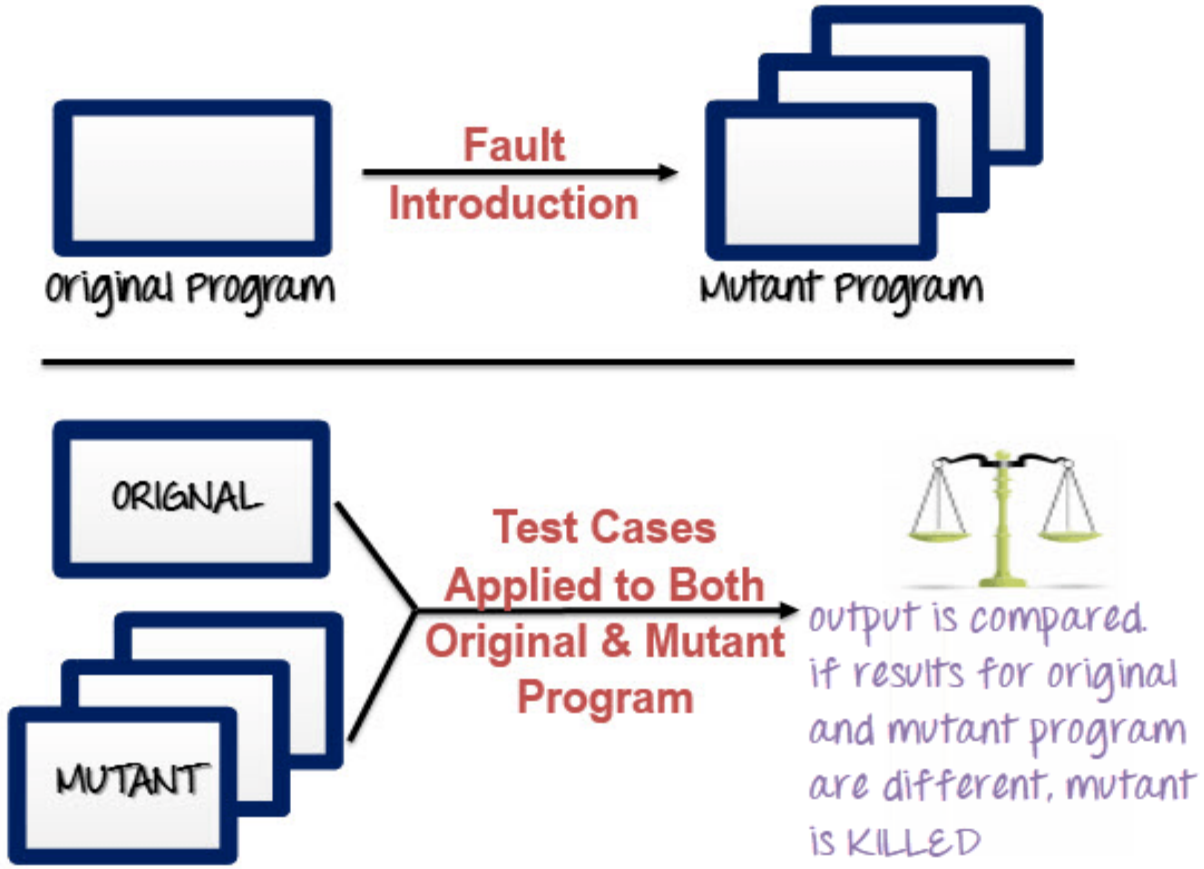
DYNAMIC UNIT TESTING

- Selection of test data is broadly based on the following techniques:
  - Control flow testing [Chapter 4]
    - Draw a control flow graph (CFG) from a program unit
    - Select a few control flow testing criteria
    - Identify a path in the CFG to satisfy the selection criteria
    - Derive the path predicate expression from the selection paths
    - By solving the path predicate expression for a path, one can generate the data
  - Data flow testing [Chapter 5]
    - Draw a data flow graph (DFG) from a program unit and then follow the procedure described in control flow testing.
  - Domain testing [Chapter 6]
    - Domain errors are defined and then test data are selected to catch those faults
  - Functional program testing [Chapter 9]
    - Input/output domains are defined to compute the input values that will cause the unit to produce expected output values

MUTATION TESTING

- Mutation testing, also known as code mutation testing, is a form of white box testing in which testers change specific components of an application's source code to ensure a software test suite will be able to detect the changes.
- Changes introduced to the software are intended to cause errors in the program.
- Modify a program by introducing a single small change to the code
- A modified program is called *mutant*
- A mutant is said to be *killed* when the execution of test case cause it to fail. The mutant is considered to be *dead*
- A mutant is an *equivalent* to the given program if it always produce the same output as the original program
- A mutant is called *killable* or *stubborn*, if the existing set of test cases is insufficient to kill it
- A mutation *score* for a set of test cases is the percentage of non-equivalent mutants *killed* by the test suite
- The test suite is said to be *mutation-adequate* if its mutation score is 100%

MUTATION TESTING



MUTATION TESTING

Consider the following program P

```
1 main(argc,argv)
2 int argc, r, i;
3 char *argv[];
4 { r = 1;
5 for i = 2 to 3 do
6 if (atoi(argv[i]) > atoi(argv[r])) r = i;
7 printf("Value of the rank is %d \n", r);
8 exit(0); }
```

Test Case 1:

input: 1 2 3

output: Value of the rank is 3

Test Case 2:

input: 1 2 1

output: Values of the rank is 2

Test Case 3:

input: 3 1 2

output: Value of the rank is 1

Mutant 1: Change line 5 to for i = 1 to 3 do
Mutant 2: Change line 6 to if (i > atoi(argv[r])) r = i;
Mutant 3: Change line 6 to if (atoi(argv[i]) >= atoi(argv[r])) r = i;
Mutant 4: Change line 6 to if (atoi(argv[r]) > atoi(argv[r])) r = i;
Execute modified programs against the test suite, you will get the results:
Mutants 1 & 3: Programs will pass the test suite, i.e., mutants 1 & 3 are not *killable*
Mutant 2: Program will fail test cases 2
Mutant 1: Program will fail test case 1 and test cases 2
Mutation score is 50%, assuming mutants 1 & 3 non-equivalent

MUTATION TESTING

- The score is found to be low because we assumed mutants 1 & 3 are nonequivalent
- We need to show that mutants 1 and 3 are equivalent mutants or those are killable
- To show that those are killable, we need to add new test cases to kill these two mutants
- First, let us analyze mutant 1 in order to derive a “killer” test. The difference between P and mutant 1 is the starting point
- Mutant 1 starts with i = 1, whereas P starts with i = 2. There is no impact on the result r. Therefore, we conclude that mutant 1 is an equivalent mutant
- Second, if we add a fourth test case as follows:

  Test Case 4:

  input: 2 2 1

- Program P will produce the output “Value of the rank is 1” and mutant 3 will produce the output “Value of the rank is 2”
- Thus, this test data kills mutant 3, which give us a mutation score 100%

MUTATION TESTING

- Execute each test case in T against each mutant P_i. If the output of the mutant P_i differs from the output of the original program P, the mutant P_i is considered incorrect and is said to be killed by the test case.
- If P_i produces exactly the same results as the original program P for the tests in T, then one of the following is true:
  - P and P_i are *equivalent*. What differentiates them from others is that they have the same meaning as the original source code, even though they may have different syntax. That is, their behaviors cannot be distinguished by any set of test cases. Note that the general problem of deciding whether or not a mutant is equivalent to the original program is undecidable.
  - P_i is *killable*. That is, the test cases are insufficient to kill the mutant P_i. In this case, new test cases must be created.

MUTATION SCORE

- The mutation score is defined as the percentage of killed mutants with the total number of mutants.
- Mutation score = $100 \times D/(N - E)$, where D is the dead mutants, N the total number of mutants, and E the number of equivalent mutants.
- Test cases are mutation adequate if the score is 100%.
- It is an effective approach for measuring the adequacy of the test cases.
- Main drawback is high cost of generating the mutants and executing each test case against that mutant program.
- Tools come in handy to speed up the process of mutant generation. Some tools that can be used for mutation testing: Stryker, Jumble, PIT, and Insure++.

MUTATION TESTING TYPES

- Value Mutation
  - Here, we introduce a mutation by changing the parameter and/or constant values, usually by +/- 1.
  - If the above code was meant to multiply the even numbers where **i<4**, then value mutation would mean changing the initialization to let **i=1**.

```
let arr = [2,3,4,5]
for(let i=0; i<arr.length; i++)
{
   if(i%2===0)
  {
    console.log(i*2)
   }
}
```

Original Code

```
let arr = [2,3,4,5]
for(let i=1; i<arr.length; i++)
{
   if(i%2===0)
  {
    console.log(i*2)
   }
}
```

Mutant Code

MUTATION TESTING TYPES

- Statement Mutation
  - Here, we delete or duplicate a statement in a code block. We could also rearrange statements in a code block.
  - In an if-else block, for example, we could delete the else part or even the entire if-else block.

```
let arr = [2,3,4,5]
for(let i=0; i<arr.length; i++)
{
   if(i%2===0)
   {
    console.log(i*2)
   }
}
```

Original Code

```
let arr = [2,3,4,5]
for(let i=0; i<arr.length; i++)
{
   if(i%2===0)
   {
    console.log(i*2)
    console.log(i*2)
   }
}
```

Mutant Code

MUTATION TESTING TYPES

- Decision Mutation
  - The target here is the code that makes decisions, for example, value comparisons.
  - Other operators that we can switch include the following:

	Original Operator	Mutant Operator
1	<=	>=
2	>=	==
3	===	==
4	and	or
5	||	&&

NUMERICAL EXAMPLE

A test engineer generates 70 mutants of a program *P* and 150 test cases to test the program *P*. After the first iteration of mutation testing, the tester finds 58 dead mutants and 4 equivalent mutants. Calculate the mutation score for this test suite. Is the test suite adequate for program *P*? Should the test engineer develop additional test cases? Justify your answer.

SOLUTION

Mutation Score = 100 D/(N − E), where D = number of dead mutants = 58, N = total number of mutants = 70, and E = number of equivalent mutants = 4.

Therefore, mutation score = 100 x 58/66 = 87%.

The test suite is not adequate, because the mutation score is not 100%.

There are 8 mutants that are killable. The tester needs to develop additional test cases to kill these eight mutants in order to achieve 100% mutation score.

DEBUGGING

- The process of determining the cause of a failure is known as *debugging*
- It is a time consuming and error-prone process
- Debugging involves a combination of systematic evaluation, intuition and a little bit of luck
- The purpose is to isolate and determine its specific cause, given a symptom of a problem
- There are three approaches to *debugging*
  - Brute force
  - Cause elimination
  - Backtracking

DEBUGGING

- **Brute force method:**
  - Print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error.
- **Backtracking:**
  - Starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered.
- **Cause elimination method:**
  - Once a failure is observed, the symptoms of the failure are noted.
  - Based on the failure symptoms, the causes which could have contributed to the symptom is developed and tests are conducted to eliminate each.

UNIT TESTING IN EXTREME PROGRAMMING

1. Pick a requirement, i.e., a story
2. Write a test case that will verify a small part of the story and assign a fail verdict to it
3. Write the code that implement particular part of the story to pass the test
4. Execute all test
5. Rework on the code, and test the code until all tests pass
6. Repeat step 2 to step 5 until the story is fully implemented

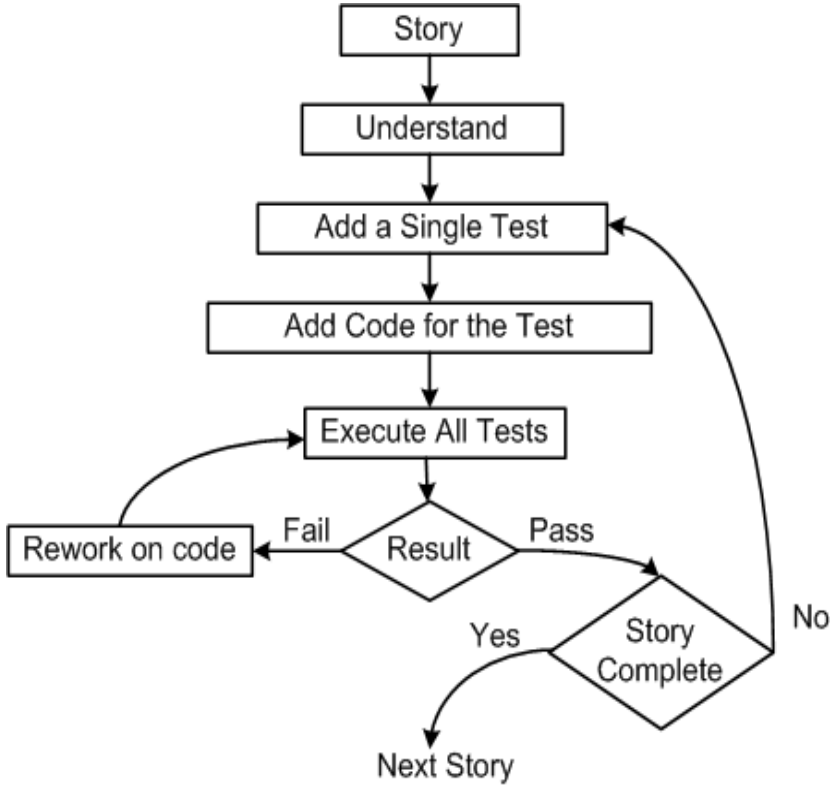


Figure 3: *Test-first* process in XP

UNIT TESTING IN EXTREME PROGRAMMING

- Three laws of Test Driven development (TDD)
  - One may not write production code unless the first failing unit test is written
  - One may not write more of a unit test than is sufficient to fail
  - One may not write more production code than is sufficient to make the failing unit test pass
- **Pair programming:**
  - In XP, code is being developed by two programmers working side by side
  - One person develops the code tactically and the other one inspects it methodically by keeping in mind the story they are implementing

TOOLS FOR UNIT TESTING

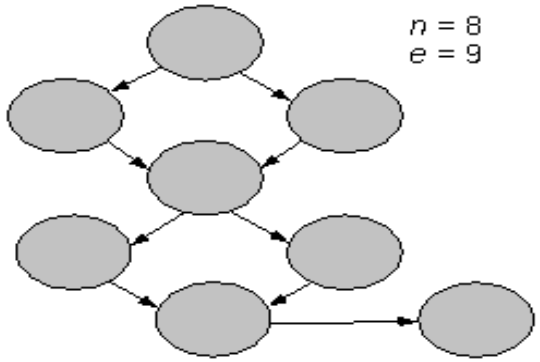
- Code auditor
  - This tool is used to check the quality of the software to ensure that it meets some minimum coding standard
- Bound checker
  - This tool can check for accidental writes into the instruction areas of memory, or to other memory location outside the data storage area of the application
- Documenters
  - These tools read the source code and automatically generate descriptions and caller/callee tree diagram or data model from the source code
- Interactive debuggers
  - These tools assist software developers in implementing different debugging techniques
    - Examples: Breakpoint and Omniscient debuggers
- In-circuit emulators
  - It provides a high-speed Ethernet connection between a host debugger and a target microprocessor, enabling developers to perform source-level debugging

TOOLS FOR UNIT TESTING

- Memory leak detectors
  - These tools test the allocation of memory to an application which request for memory and fail to de-allocate memory
- Static code (path) analyzer
  - These tool identify paths to test based on the structure of code such as McCabe's cyclomatic complexity measure

Cyclomatic complexity

McCabe's complexity measure is based on the cyclomatic complexity of a program graph for a module. The metric can be computed by using the formula: $v = e - n + 2$,
where:
v = cyclomatic complexity of the graph,
e = number of edges (program flow between nodes),
n = number of nodes (sequential group of program statements).
If a strongly connected graph is constructed (one in which there is an edge between the exit node and the entry node) the calculation is $v = e - n + 1$.
Example: A program graph, illustrated below is used to depict control flow. Each circled node represents a sequence of program statements, and the flow of control is represented by directed edges. For this graph the cyclomatic complexity is $v = 9 - 8 + 2 = 3$.



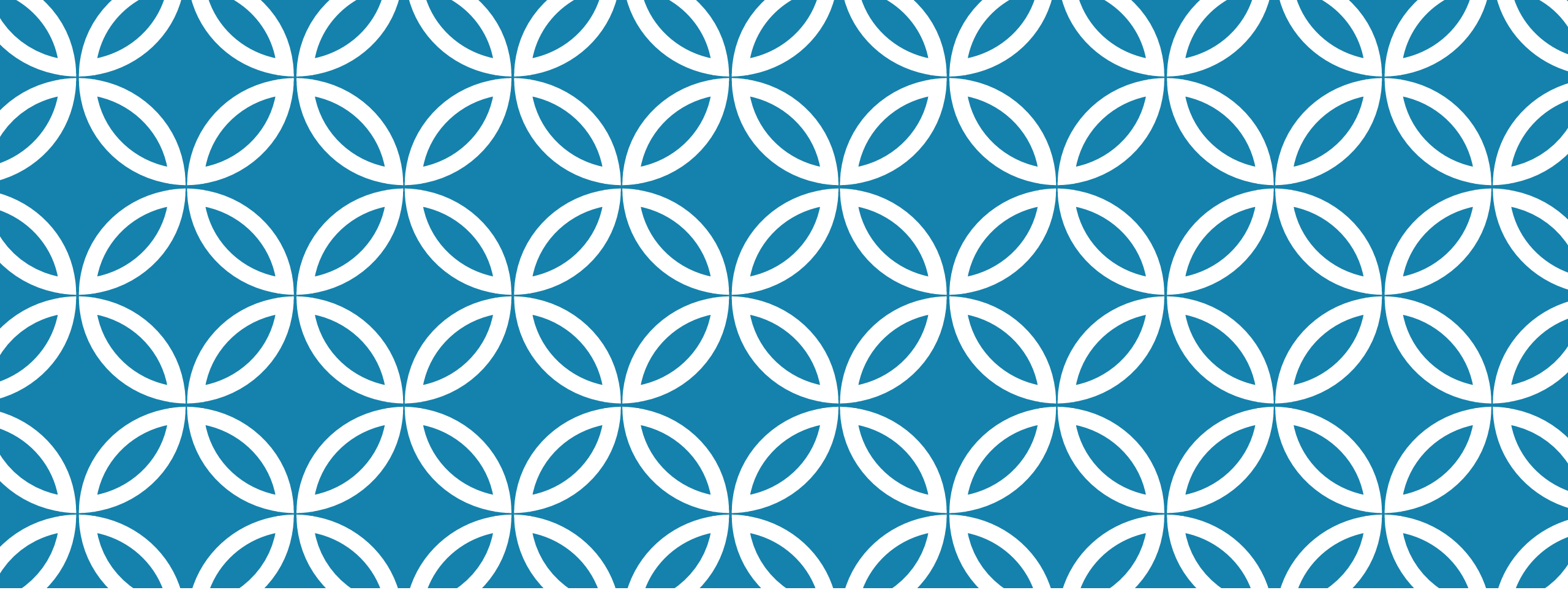
McCabe complexity measure

TOOLS FOR UNIT TESTING

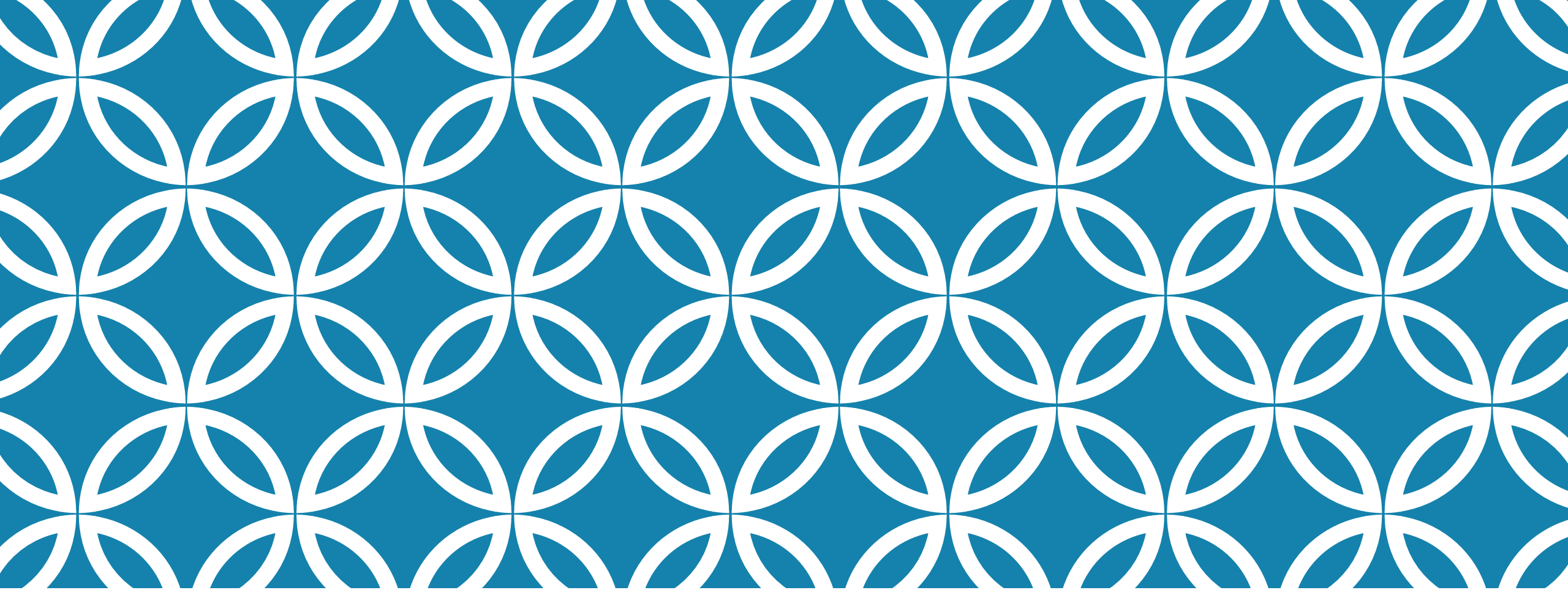
- Software inspection support
  - Tools can help schedule group inspection
- Test coverage analyzer
  - These tools measure internal test coverage, often expressed in terms of control structure of the test object, and report the coverage metric
- Test data generator
  - These tools assist programmers in selecting test data that cause program to behave in a desired manner
- Test harness
  - This class of tools support the execution of dynamic unit tests
- Performance monitors
  - The timing characteristics of the software components be monitored and evaluate by these tools
- Network analyzers
  - These tools have the ability to analyze the traffic and identify problem areas

TOOLS FOR UNIT TESTING

- Simulators and emulators
  - These tools are used to replace the real software and hardware that are not currently available. Both the kinds of tools are used for training, safety, and economy purpose
- Traffic generators
  - These produces streams of transactions or data packets.
- Version control
  - A version control system provides functionalities to store a sequence of revisions of the software and associated information files under development



THANK YOU!!



CHAPTER – 4
CONTROL FLOW TESTING

OUTLINE OF THE CHAPTER

- Basic Idea
- Outline of Control Flow Testing
- Control Flow Graph
- Paths in a Control Flow Graph
- Path Selection Criteria
- Generating Test Input
- Containing Infeasible Paths
- Summary

BASIC IDEA

- Two kinds of basic program statements:
  - Assignment statements (Ex. x = 2*y;)
  - Conditional statements (Ex. if(), for(), while(), …)
- Control flow
  - Successive execution of program statements is viewed as flow of control.
  - Conditional statements alter the default flow.
- Program path
  - A program path is a sequence of statements from entry to exit.
  - There can be a large number of paths in a program.
  - There is an (input, expected output) pair for each path.
  - Executing a path requires invoking the program unit with the right test input.
  - Paths are chosen by using the concepts of path selection criteria.
- Tools: Automatically generate test inputs from program paths.

OUTLINE OF CONTROL FLOW TESTING

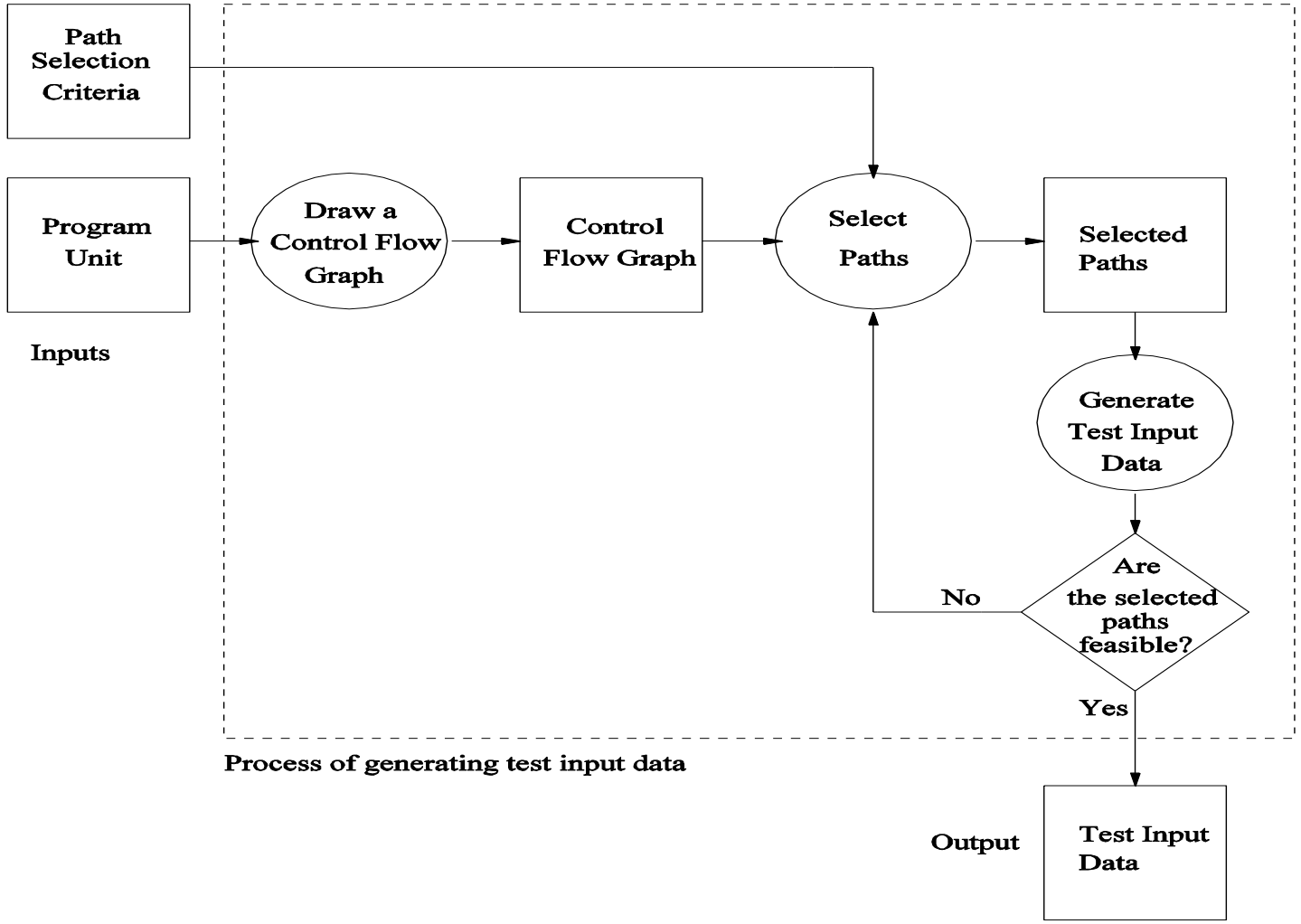


Figure 1: The process of generating test input data for control flow testing.

OUTLINE OF CONTROL FLOW TESTING

- Inputs to the test generation process
  - Source code
  - Path selection criteria: statement, branch, …
- Generation of control flow graph (CFG)
  - A CFG is a graphical representation of a program unit.
  - Compilers are modified to produce CFGs. (You can draw one by hand.)
- Selection of paths
  - Enough entry/exit paths are selected to satisfy path selection criteria.
- Generation of test input data
  - Two kinds of paths
    - Executable path: There exists input so that the path is executed.
    - Infeasible path: There is no input to execute the path.
  - Solve the path conditions to produce test input for each path.

CONTROL FLOW GRAPH

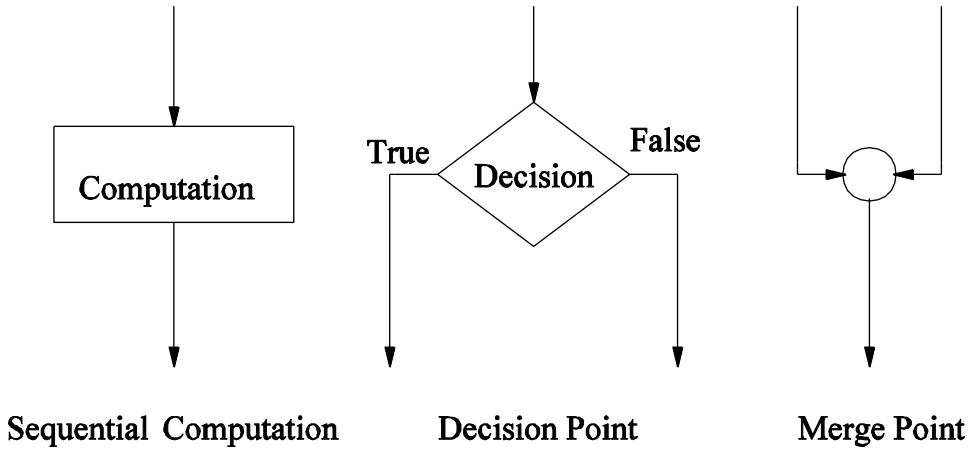


Figure 2: Symbols in a control flow graph

CONTROL FLOW GRAPH

```
FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */

int openfiles(){
   /*
      This function tries to open files "file1", "file2", and
      "file3" for read access, and returns the number of files
      successfully opened. The file pointers of the opened files
      are put in the global variables.
   */
     int i = 0;
     if(
         ((( fptr1 = fopen("file1", "r")) != NULL) && (i++)
                                                   && (0)) ||
         ((( fptr2 = fopen("file2", "r")) != NULL) && (i++)
                                                   && (0)) ||
         ((( fptr3 = fopen("file3", "r")) != NULL) && (i++))
     );
     return(i);
}
```

CONTROL FLOW GRAPH

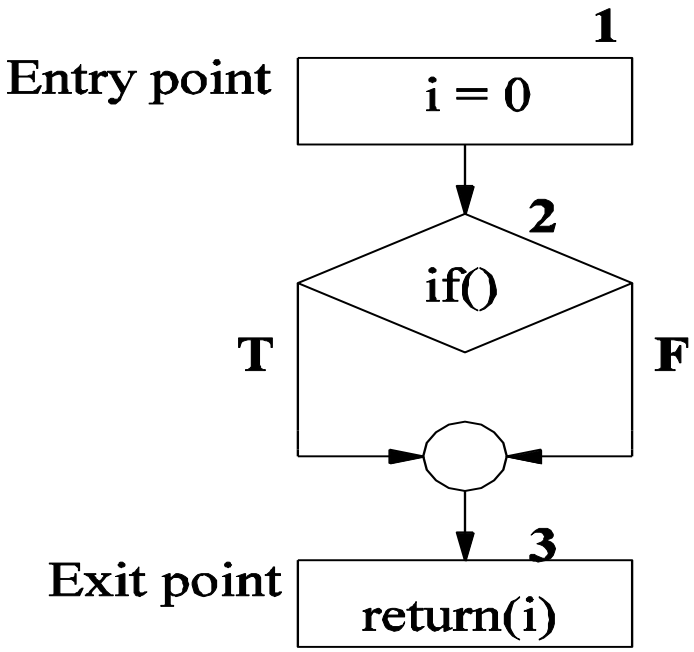


Figure 3: A high-level CFG representation of openfiles().

CONTROL FLOW GRAPH

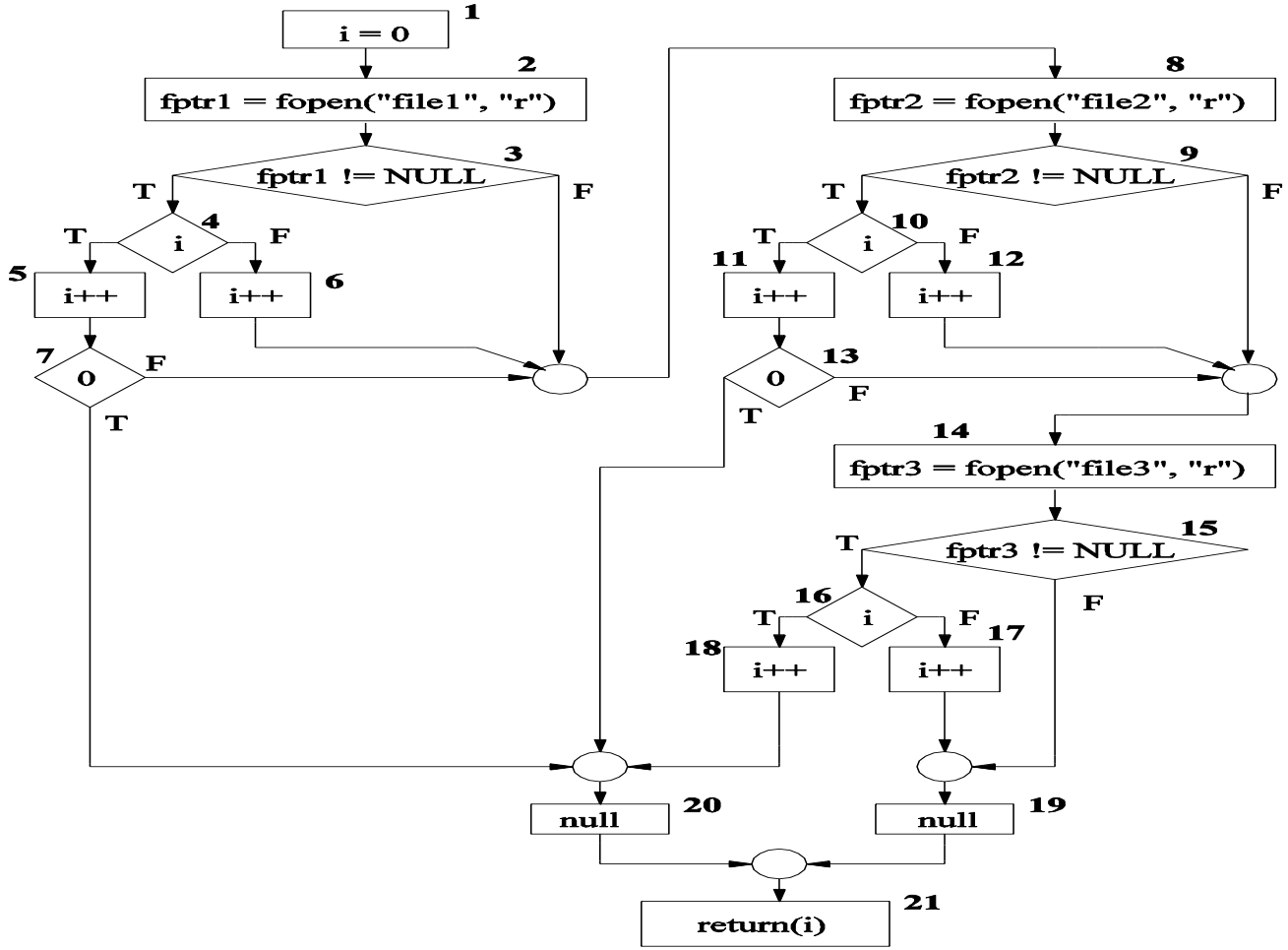


Figure 4: A detailed CFG representation of openfiles().

CONTROL FLOW GRAPH

```
public static double ReturnAverage(int value[],
                            int AS, int MIN, int MAX){
  /*
   Function: ReturnAverage  Computes the  average
   of all  those  numbers in the  input array  in
   the  positive  range  [MIN, MAX]. The  maximum
   size  of the array is AS. But, the  array size
   could be smaller than AS in which case the end
   of input is represented by -999.
  */
     int i, ti, tv, sum;
     double av;
     i = 0; ti = 0; tv = 0; sum = 0;
     while (ti < AS && value[i] != -999) {
         ti++;
         if (value[i] >= MIN && value[i] <= MAX) {
            tv++;
            sum = sum + value[i];
         }
         i++;
     }
     if (tv > 0)
        av = (double)sum/tv;
     else
        av = (double) -999;
     return (av);
}
```

CONTROL FLOW GRAPH

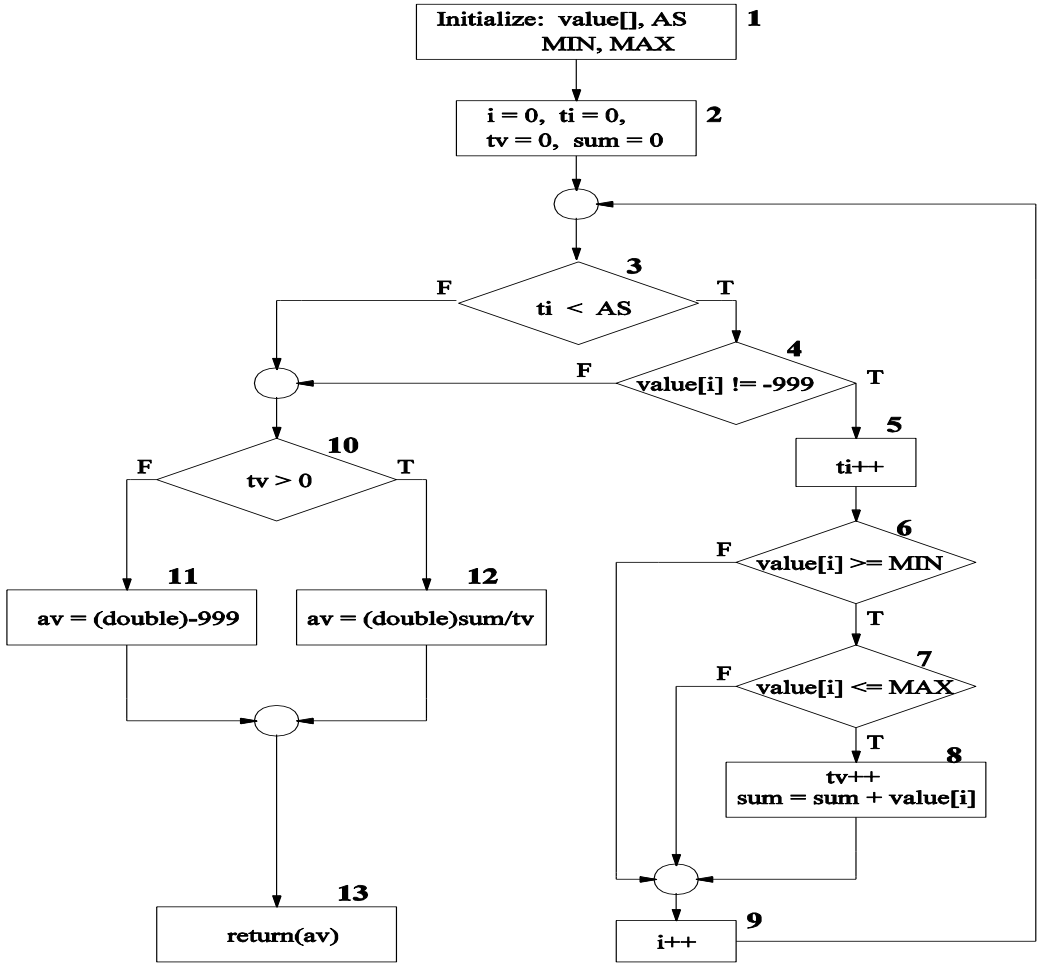


Figure 5: A CFG representation of ReturnAverage().

PATHS IN A CONTROL FLOW GRAPH

- A path is represented as a sequence of computation and decision nodes from the entry node to the exit node.
- A few paths in Figure 5
  - Path 1: 1-2-3(F)-10(T)-12-13
  - Path 2: 1-2-3(F)-10(F)-11-13
  - Path 3: 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
  - Path 4: 1-2-3(T)-4(T)-5-6-7(T)-8-9-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

PATH SELECTION CRITERIA

- Program paths are selectively executed.
- Question: What paths do I select for testing?
- The concept of *path selection criteria* is used to answer the question.
- Advantages of selecting paths based on defined criteria:
  - Ensure that all program constructs are executed at least once.
  - Repeated selection of the same path is avoided.
  - One can easily identify what features have been tested and what not.
- Path selection criteria
  - Select all paths.
  - Select paths to achieve complete statement coverage.
  - Select paths to achieve complete branch coverage.
  - Select paths to achieve predicate coverage.

PATH SELECTION CRITERIA

All-path coverage criterion: Select all the paths in the program unit under consideration.

- The openfiles() unit has 25+ paths.
- Selecting all the inputs will exercise all the program paths.

Existence of "file1"	Existence of "file2"	Existence of "file3"
No	No	No
No	No	Yes
No	Yes	No
No	Yes	Yes
Yes	No	No
Yes	No	Yes
Yes	Yes	No
Yes	Yes	Yes

Table 1: Input Domain of openfiles()

PATH SELECTION CRITERIA

Input	Path
<No, No, No>	1-2-3(F)-8-9(F)-14-15(F)-19-21
<Yes, No, No>	1-2-3(T)-4(F)-6-8-9(F)-14-15(F)-19-21
<Yes, Yes, Yes>	1-2-3(T)-4(F)-6-8-9(T)-10(T)-11-13(F)-14- 15(T) - 16(T)-18-20-21

Table 2: Inputs and paths in openfiles()

PATH SELECTION CRITERIA

- **Statement coverage criterion**
  - Statement here means node.
  - Statement coverage means executing individual program statements and observing the output.
  - 100% statement coverage means all the statements have been executed at least once.
    - Cover all assignment statements.
    - Cover all conditional statements.
  - Less than 100% statement coverage is unacceptable.

SCPath1	1-2-3(F)-10(F)-11-13
SCPath2	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

Table 3: Paths for statement coverage of the CFG of Figure 5.

PATH SELECTION CRITERIA

- **Branch coverage criterion**
  - A branch is an outgoing edge from a node in a CFG.
    - A condition node has two outgoing branches – corresponding to the True and False values of the condition.
  - Covering a branch means executing a path that contains the branch.
  - Complete branch coverage means selecting a number of paths such that every branch is included in at least one path.

PATH SELECTION CRITERIA

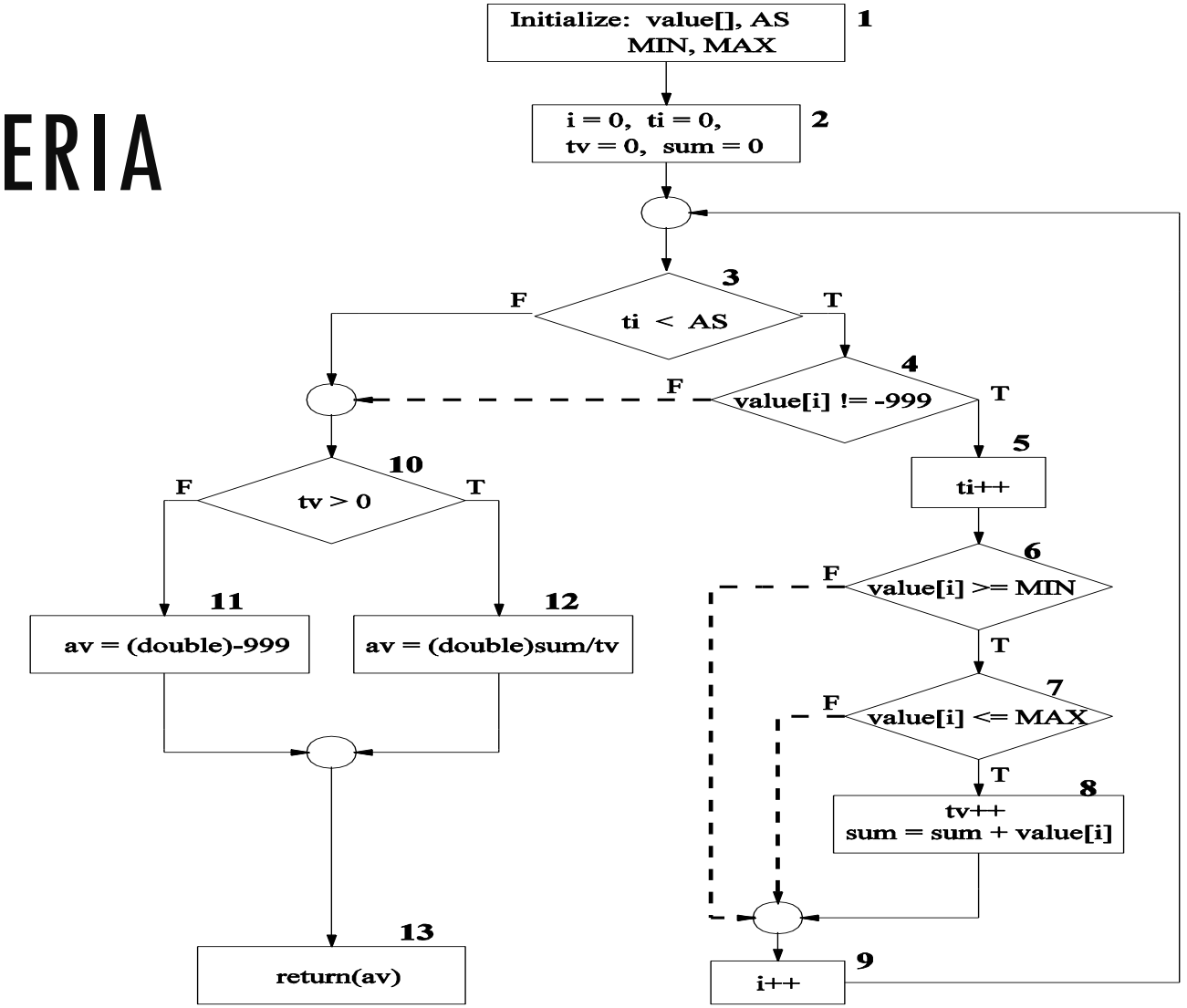


Figure 6: The dotted arrows represent the branches not covered by the statement covering in Table 3.

PATH SELECTION CRITERIA

BCPath 1	1-2-3(F)-10(F)-11-13
BCPath 2	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
BCPath 3	1-2-3(T)-4(F)-10(F)-11-13
BCPath 4	1-2-3(T)-4(T)-5-6(F)-9-3(F)-10(F)-11-13
BCPath 5	1-2-3(T)-4(T)-5-6(T)-7(F)-9-3(F)-10(F)-11-13

Table 4: Paths for branch coverage of the flow graph of Figure 5

PATH SELECTION CRITERIA

- **Predicate coverage/condition coverage criterion**
  - If all possible combinations of truth values of the conditions affecting a path have been explored under some tests, then we say that predicate coverage has been achieved.

Cases	OB1	OB2	OB3	OB
1	T	F	F	T
2	F	F	F	F

Table 5: Two test cases providing complete statement coverage and branch coverage

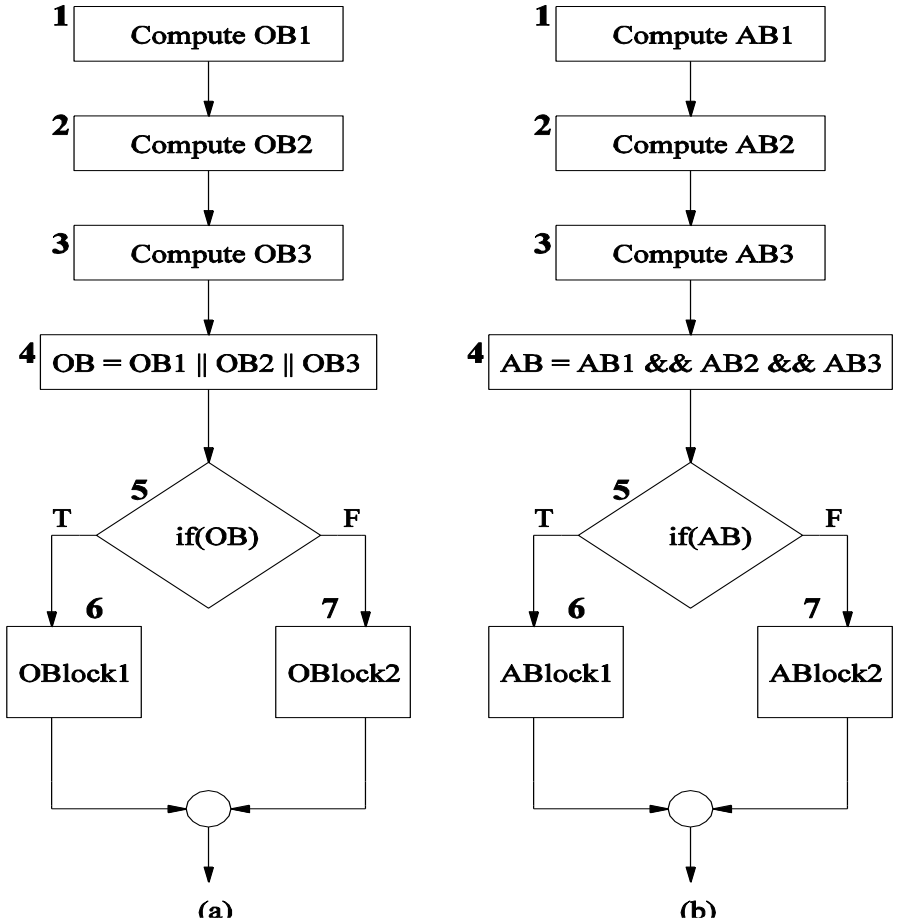


Figure 7: Partial control flow graph with (a) OR operation and (b) AND operation.

PATH SELECTION CRITERIA

- The False branch of node **5** is executed under exactly one condition, namely, when OB1 = False, OB2 = False, and OB3 = False, whereas the true branch executes under *seven* conditions.
- If all possible combinations of truth values of the conditions affecting a selected path have been explored under some tests, then we say that *predicate coverage* has been achieved.
- Therefore, the path taking the true branch of node **5** in Figure 7 must be executed for all seven possible combinations of truth values of OB1, OB2, and OB3, which results in OB = True.

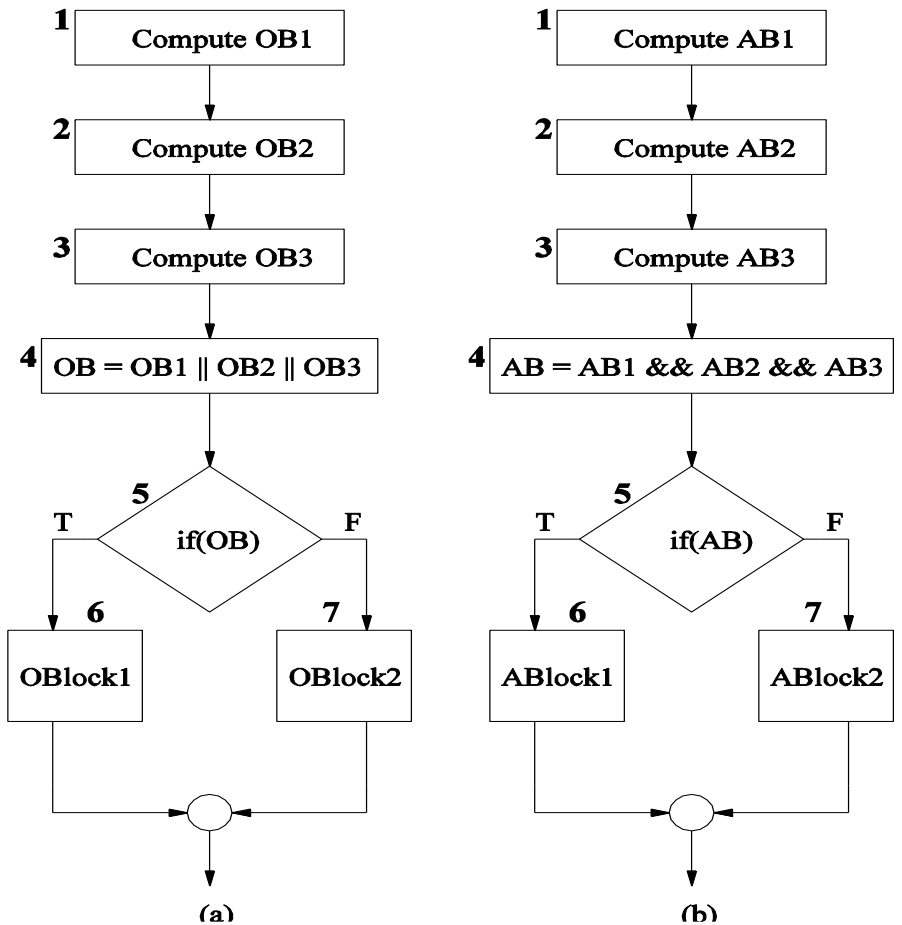


Figure 7: Partial control flow graph with (a) OR operation and (b) AND operation.

GENERATING TEST INPUT

- Having identified a path, a key question is how to make the path execute, if possible.
  - Generate input data that satisfy all the conditions on the path.
- Key concepts in generating test input data
  - Input vector
  - Predicate
  - Path predicate
  - Predicate interpretation
  - Path predicate expression
  - Generating test input from path predicate expression

GENERATING TEST INPUT

- Input vector
  - An input vector is a collection of all data entities read by the routine whose values must be fixed prior to entering the routine.
  - Members of an input vector can be as follows.
    - Input arguments to the routine
    - Global variables and constants
    - Files
    - Contents of registers (in Assembly language programming)
    - Network connections
    - Timers
  - Example: An input vector for openfiles() consists of individual presence or absence of the files “files1,” “file2,” and “file3.”
  - Example: The input vector of ReturnAverege() shown in Figure 5 is <value[], AS, MIN, MAX>.

GENERATING TEST INPUT

- Predicate
  - A predicate is a logical function evaluated at a decision point.
  - Example: ti < AS is a predicate in node 3 of Figure 5.
  - Example: The construct OB is a predicate in node 5 in Figure 7.
- Path predicate
  - A path predicate is the set of predicates associated with a path.
  - An example path from Figure 5.
    - 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13.
  - The path predicate for the path shown above.
    - ti < AS ≡ True
    - value[i] != -999 ≡ True
    - value[i] >= MIN ≡ True
    - value[i] <= MAX ≡ True
    - ti < AS ≡ False
    - tv > 0 ≡ True

GENERATING TEST INPUT

- Predicate interpretation
  - A path predicate may contain local variables.
  - Example: It is composed of elements of the input vector <value[], AS, MIN, MAX>, a vector of local variables <i, ti, tv>, and the constant −999.
  - The local variables are not visible outside a function but are used to hold intermediate results, pointer to array elements, and control loop iterations.
  - Local variables play no role in selecting inputs that force a path to execute.
  - Local variables can be eliminated by a process called **symbolic execution**.

SYMBOLIC SUBSTITUTION

```
public static int SymSub(int x1, int x2){
int y;
y = x2 + 7;
if (x1 + y >= 0)
     return (x2 + y);
else return (x2 - y);
}
```

- The input vector for the method in example is given by<x1, x2>.
- The method defines a local variable y and uses the constants 7 and 0.
- The predicate x1 + y >= 0 can be rewritten as x1 + x2 + 7 >= 0 by symbolically substituting y with x2 +7.
- The rewritten predicate x1 + x2 + 7 >= 0 has been expressed solely in terms of the input vector<x1,x2>and the constant vector<0,7>.
- Predicate interpretation is defined as the process of
  - symbolically substituting operations along a path in order to express the predicate solely in terms of the input vector and a constant vector.
- A predicate may have different interpretations depending on how control reaches the predicate.

GENERATING TEST INPUT

- Path predicate expression
  - An interpreted path predicate is called a path predicate expression.
  - A path predicate expression has the following attributes.
    - It is void of local variables.
    - It is a set of constraints in terms of the input vector, and, maybe, constants.
    - Path forcing inputs can be generated by solving the constraints.
    - If a path predicate expression has no solution, the path is infeasible.
  - Path predicate expression for the path → 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13 [Figure 5].
    - 0 < AS ≡ True …… (1)
    - value[0] != -999 ≡ True …… (2)
    - value[0] >= MIN ≡ True …… (3)
    - value[0] <= MAX ≡ True …… (4)
    - 1 < AS ≡ False …… (5)
    - 1 > 0 ≡ True …… (6)

GENERATING TEST INPUT

- Path → 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
- The rows here have been obtained from Side 24 [from path predicate] by combining each interpreted predicate in column 3 with its intended evaluation in column 1.

Interpretation of Path Predicate of Path in Figure 5

Node	Node Description	Interpreted Description
1	Input vector: < value[], AS, MIN, MAX >	
2	i = 0, ti = 0, tv = 0, sum = 0	
3(T)	ti < AS	0 < AS
4(T)	value[i]! = − 999	value[0]! = − 999
5	ti++	ti = 0 + 1 = 1
6(T)	value[i] > = MIN	value[0] > = MIN
7(T)	value[i] < = MAX	value[0] < = MAX
8	tv++	tv = 0 + 1 = 1
	sum = sum + value[i]	sum = 0 + value[0] = value[0]
9	i++	i = 0 + 1 = 1
3(F)	ti < AS	1 < AS
10(T)	tv > 0	1 > 0
12	av = (double) sum/tv	av = (double) value[0]/1
13	return(av)	return(value[0])

Note: The bold entries in column 1 denote interpreted predicates.

GENERATING TEST INPUT

Path predicate expression

- An example of infeasible path
- Another example of path from Figure 5.
  - 1-2-3(T)-4(F)-10(T)-12-13
- Path predicate expression for the path shown above.

$0 < AS \equiv True \quad \ldots\ldots (1)$

$value[0] != -999 \equiv True \quad \ldots\ldots (2)$

$0 > 0 \equiv True \quad \ldots\ldots (3)$

Node	Node Description	Interpreted Description
1	Input vector: < value[], AS, MIN, MAX >	
2	i = 0, ti = 0, tv = 0, sum = 0	
3(T)	ti < AS	0 < AS
4(F)	value[i]! = − 999	value[0]! = − 999
10(T)	tv > 0	0 > 0
12	av = (double)sum/tv	av = (double)value[0]/0
13	return(av)	return((double) value[0]/0)

Note: The bold entries in column 1 denote interpreted predicates.

GENERATING TEST INPUT

Generating input data from a path predicate expression

- Consider the path predicate expression (reproduced below.)

0 < AS	≡ True	…… (1)
value[0] != -999	≡ True	…… (2)
value[0] >= MIN	≡ True	…… (3)
value[0] <= MAX	≡ True	…… (4)
1 < AS	≡ False	…… (5)
1 > 0	≡ True	…… (6)

- One can solve the above equations to obtain the following test input data

AS	= 1
MIN	= 25
MAX	= 35
Value[0]	= 30

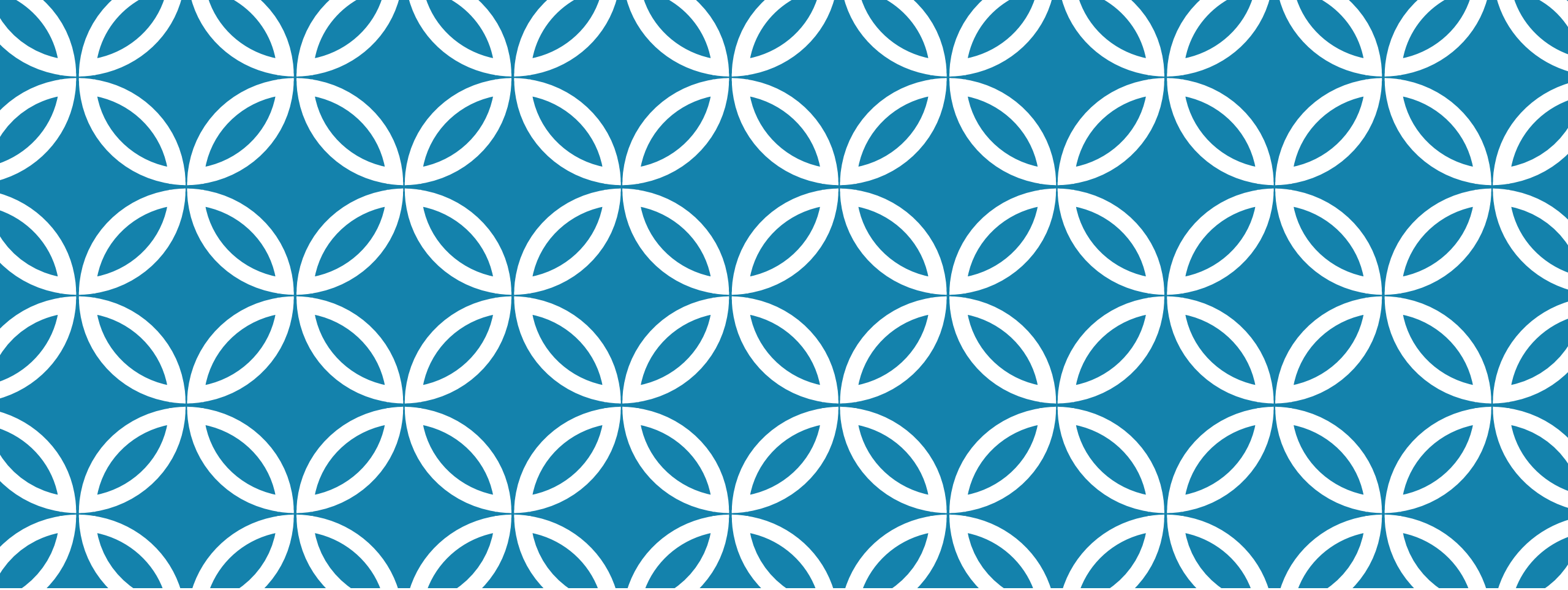
- Note: The above set is not unique.

CONTAINING INFEASIBLE PATHS

- A program unit may contain a large number of paths.
  - Path selection becomes a problem. Some selected paths may be infeasible.
  - Apply a path selection strategy:
    - Select as many short paths as possible.
    - Choose longer paths.
  - There are efforts to write code with fewer/no infeasible paths.

SUMMARY

- Control flow is a fundamental concept in program execution.
- A program path is an instance of execution of a program unit.
- Select a set of paths by considering path **selection criteria**.
  - Statement coverage
  - Branch coverage
  - Predicate coverage
  - All paths
- From source code, derive a CFG (compilers are modified for this.)
- Select paths from a CFG based on path selection criteria.
- Extract path predicates from each path.
- Solve the path predicate expression to generate test input data.
- There are two kinds of paths.
  - feasible
  - infeasible



THANK YOU!!