**Software Testing**

Table of Contents

[Importance 8](#_Toc83393451)

[Myths and Facts 10](#_Toc83393452)

[Goals 11](#_Toc83393453)

[ Immediate Goals 11](#_Toc83393454)

[Terminology 12](#_Toc83393455)

[Causes of Bugs 13](#_Toc83393456)

[Defect/Bug Life Cycle 15](#_Toc83393457)

[Invalid Bugs 16](#_Toc83393458)

[Duplicate Bug 16](#_Toc83393459)

[Not Reproducible Bug 17](#_Toc83393460)

[Can’t Fix Bug 18](#_Toc83393461)

[Deferred/Postponed 18](#_Toc83393462)

[Request for Enhancement 18](#_Toc83393463)

[Bug Severity 19](#_Toc83393464)

[Bug Priority 20](#_Toc83393465)

[Bug Reports 21](#_Toc83393466)

[Parameters 21](#_Toc83393467)

[Characteristics 21](#_Toc83393468)

[Sample 22](#_Toc83393469)

[Bug Management 24](#_Toc83393470)

[Test Cases 25](#_Toc83393471)

[Fields 25](#_Toc83393472)

[Characteristics 26](#_Toc83393473)

[Software Testing Life Cycle 28](#_Toc83393474)

[Requirement Analysis 28](#_Toc83393475)

[Test Planning 29](#_Toc83393476)

[Test Case Development 30](#_Toc83393477)

[Environment Setup 30](#_Toc83393478)

[Test Execution 31](#_Toc83393479)

[Test Cycle Closure 32](#_Toc83393480)

[Testing Categories and Techniques 33](#_Toc83393481)

[Black-Box Testing 34](#_Toc83393482)

[Techniques 35](#_Toc83393483)

[Boundary Value Analysis 35](#_Toc83393484)

[Decision Table-Based Testing 37](#_Toc83393485)

[White-Box Testing 40](#_Toc83393486)

[Basis Path Testing 43](#_Toc83393487)

[Control Flow Graphs 43](#_Toc83393488)

[Cyclomatic Complexity 44](#_Toc83393489)

[Independent Paths 45](#_Toc83393490)

[Validation Activities 47](#_Toc83393491)

[Integration Testing 48](#_Toc83393492)

[Big Bang Approach 48](#_Toc83393493)

[Bottom-Up Testing Strategy 49](#_Toc83393494)

[Top-Down Testing Strategy 50](#_Toc83393495)

[Call Graph Based Integration Testing 51](#_Toc83393496)

[Path-Based Integration Testing 53](#_Toc83393497)

[System Testing 56](#_Toc83393498)

[Steps 56](#_Toc83393499)

[Test Cases 57](#_Toc83393500)

[Acceptance Testing 58](#_Toc83393501)

[Entry Criteria 58](#_Toc83393502)

[Exit Criteria 58](#_Toc83393503)

[UAT Process 59](#_Toc83393504)

[Alpha Testing 60](#_Toc83393505)

[Entry Criteria 60](#_Toc83393506)

[Exit Criteria 61](#_Toc83393507)

[Advantages 61](#_Toc83393508)

[Disadvantages 61](#_Toc83393509)

[Beta Testing 62](#_Toc83393510)

[Entrance Criteria 62](#_Toc83393511)

[Exit Criteria 63](#_Toc83393512)

[Guidelines 63](#_Toc83393513)

[Advantages 63](#_Toc83393514)

[Disadvantages 63](#_Toc83393515)

[Smoke Testing 65](#_Toc83393516)

[Scenarios 65](#_Toc83393517)

[Advantages 66](#_Toc83393518)

[Sanity Testing 67](#_Toc83393519)

[Regression Testing 68](#_Toc83393520)

[Challenges 68](#_Toc83393521)

[Test Case Selection 69](#_Toc83393522)

[Non-Functional Testing 70](#_Toc83393523)

[Performance Testing 71](#_Toc83393524)

[Parts to Test 72](#_Toc83393525)

[Process 72](#_Toc83393526)

[Load Testing 73](#_Toc83393527)

[Stress Testing 73](#_Toc83393528)

[Scalability Testing 73](#_Toc83393529)

[Stability Testing 74](#_Toc83393530)

[Performance Testing Tools 74](#_Toc83393531)

[Compatibility Testing 75](#_Toc83393532)

[Version Checking 75](#_Toc83393533)

[Benefits 76](#_Toc83393534)

[Common Defects 76](#_Toc83393535)

[Compliance Testing 77](#_Toc83393536)

[Steps 77](#_Toc83393537)

[Recovery Testing 78](#_Toc83393538)

[Steps 78](#_Toc83393539)

[Security Testing 79](#_Toc83393540)

[Usability Testing 80](#_Toc83393541)

[Process 80](#_Toc83393542)

**Software Testing** is the process of executing a program with the **intent of finding errors**. The process **detects important bugs** with the objective of improving software quality.

In software testing, we ensure that the software meets **user requirements**, called the **validation** stage, and that the **process to build** the software was followed correctly, called the **verification** stage. Thus, we ensure that we built the right thing and that we built it in the right way.

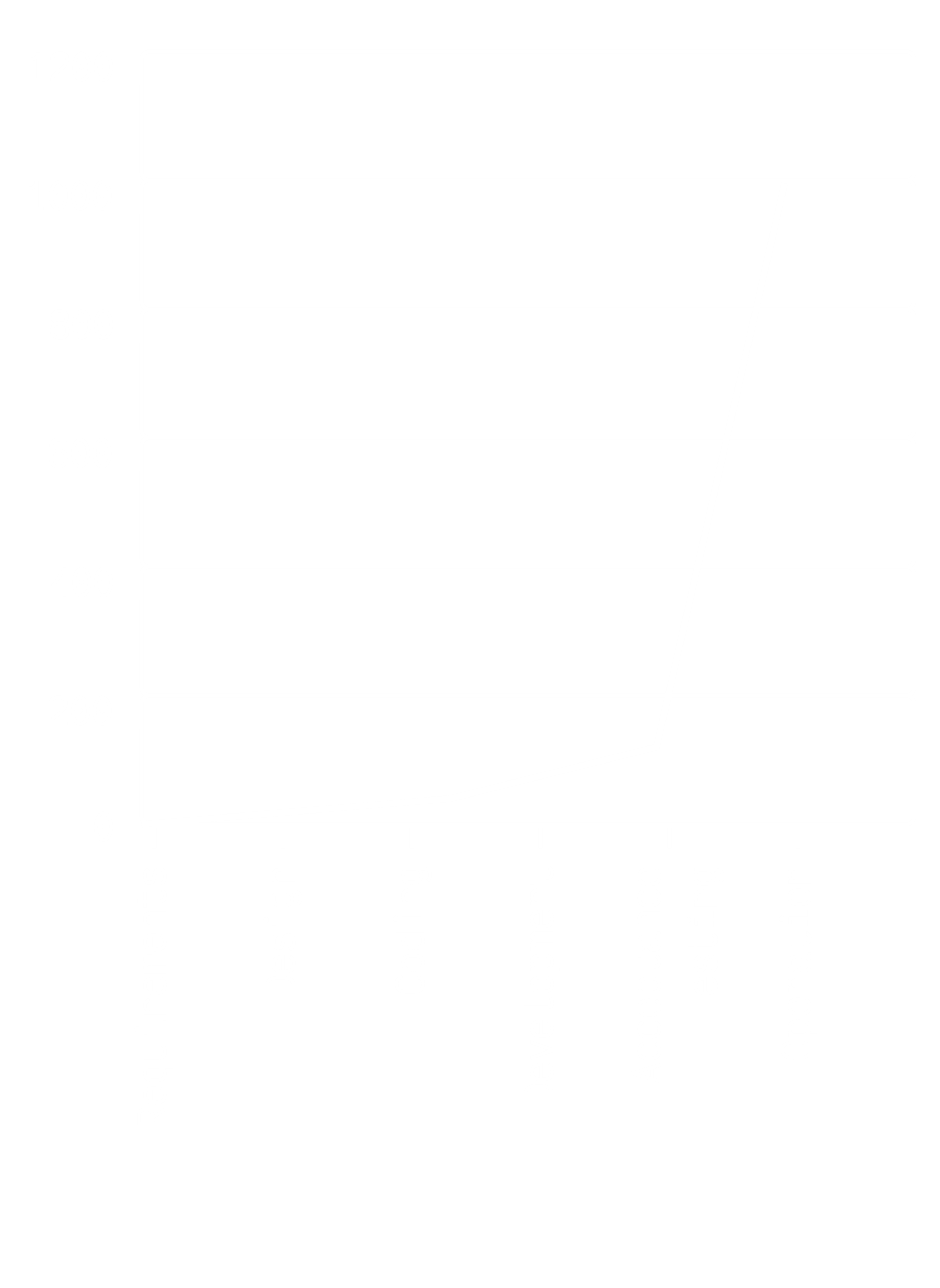
## Importance

The primary purpose is to demonstrate that the software is of **good quality** and is **behaving correctly**. We do this by **detecting and fixing problems**.

Sadly, it is never possible to make a software bug-free. A **few testers** attempt to test all possible **test cases**, while the real application may have **millions of users**. It is entirely possible that a few test cases will be **missed**, which will allow bugs to go **unnoticed**.

The impacts on the **development firm** of bugs in software that end up in the users’ hands include:

* **Financial Loss** – On the one hand, we will lose business, but on the other hand, we could also end up facing penalties due to non-compliance with regulations.
* **Reputational Loss** – Customers will take future business elsewhere if a company delivers poor quality software. Potential customers are also lost.
* **Increased Costs** – Software that does not work may need manual workarounds or could result in staff being unable to work. The later in the development stage we find a bug, the more it will cost us, but if the bug manages to get into the live application, the cost is significantly more.



Even though perfect software cannot be produced, we still need to try in order to **minimize these effects**.

## Myths and Facts

* Testing is a **single phase** in the SDLC – Testing is actually involved in **every stage**, from the moment we get requirements to post-implementation of the software.
* Testing is **easy** – We need to plan and develop all possible test cases **manually**, which requires a **thorough understanding** of the overall design of the project. Testers have a lot of responsibility, which makes their jobs harder than that of the developers.
* Software development is **worth more** than software testing – This cannot be justified. Testing is **just as important** as development.
* **Complete testing** is possible – There are lots of cases that **cannot be tested**. This could be because it takes a huge amount of time to test some cases or due to restrictions.
* The **purpose** of testing is to check the **functionality** of the software – The purpose is to improve quality, and that is not limited to just functionality.
* **Anyone** can be a software tester – A **good tester** needs to be able to think **analytically** and is rare.

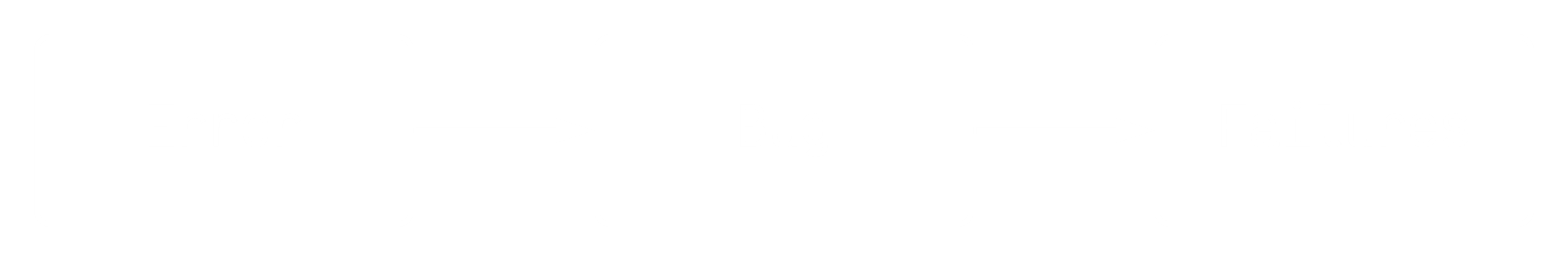
## Goals

### Immediate Goals

* + Bug Discovery
  + Bug Prevention
* Long-Term Goals
  + Reliability
  + Risk Management
  + Quality
  + Customer Satisfaction
* Post-Implementation Goals
  + Reduced maintenance costs
  + Improved testing process

## Terminology

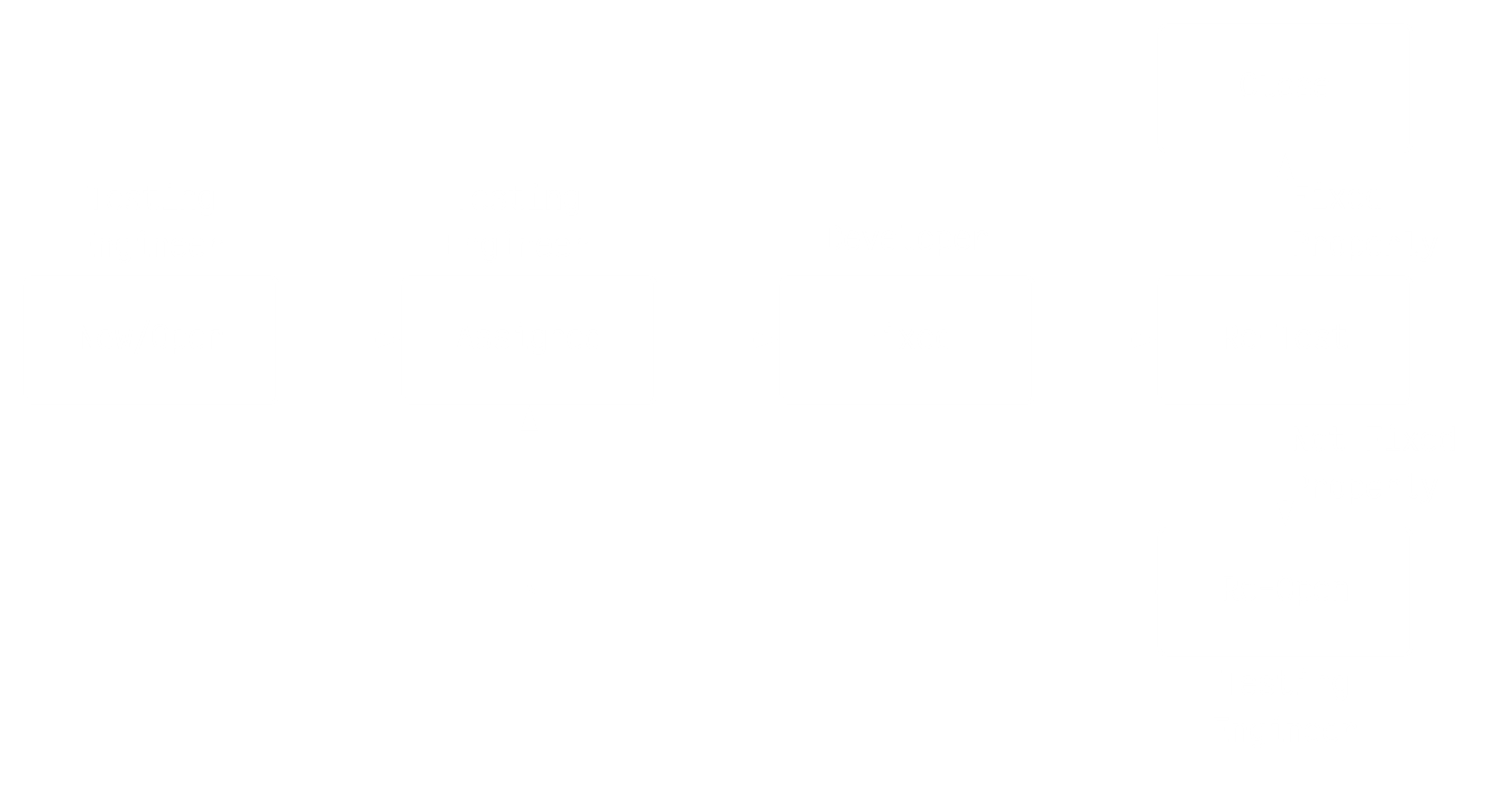
* **Failure** – This is the inability of a system or a component to perform a required function according to its specification.
* **Fault/Defect/Bug** – This is a fault due to an error in code, either due to incorrect design, or logic or implementation, which results in the software not behaving as expected.
* **Error** – Whenever the development team makes a mistake, errors are produced. This could be a typographical error, misunderstanding of a specification, misunderstanding of what a subroutine does and so on.



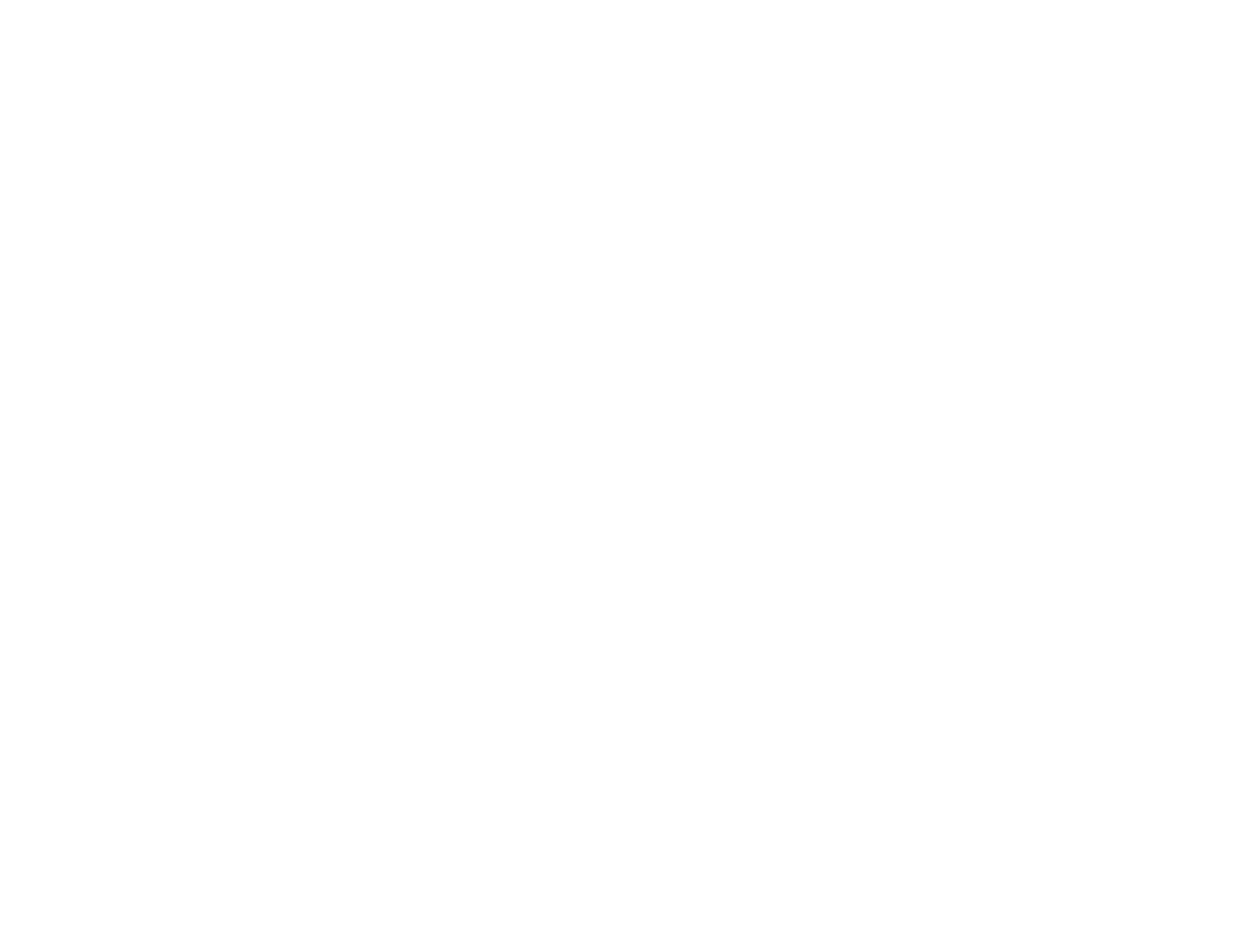
## Causes of Bugs

* **Faulty Definition of Requirements**
  + Erroneous requirement definitions
  + Absence of important requirements
  + Incomplete requirements
  + Unnecessary requirements included
* **Client-Developer Communication Failures**
  + Limited communication between client and developers
  + Misunderstanding of client requirements presented in writing, orally, etc.
  + Misunderstanding of client responses to design problems
* **Coding Errors**
  + Errors interpreting the design document, related to incorrect use of programming language constructs, etc.
* **Deliberate Deviations from Software Requirements**
  + Reuse of existing software components from previous projects without complete analysis
  + Functionality omitted due to budget or time constraints
  + ‘Improvements’ to software that are not in the requirements
* **Logical Design Errors**
  + Errors in interpreting requirements into a design, e.g. errors in definition of boundary conditions, algorithms, reactions to illegal operations, etc.
* **Shortcomings of Testing Process**
  + Incomplete test plan
  + Failure to report all errors/faults resulting from testing
  + Incorrect report of errors/faults
  + Incomplete correction of detected errors

## Defect/Bug Life Cycle

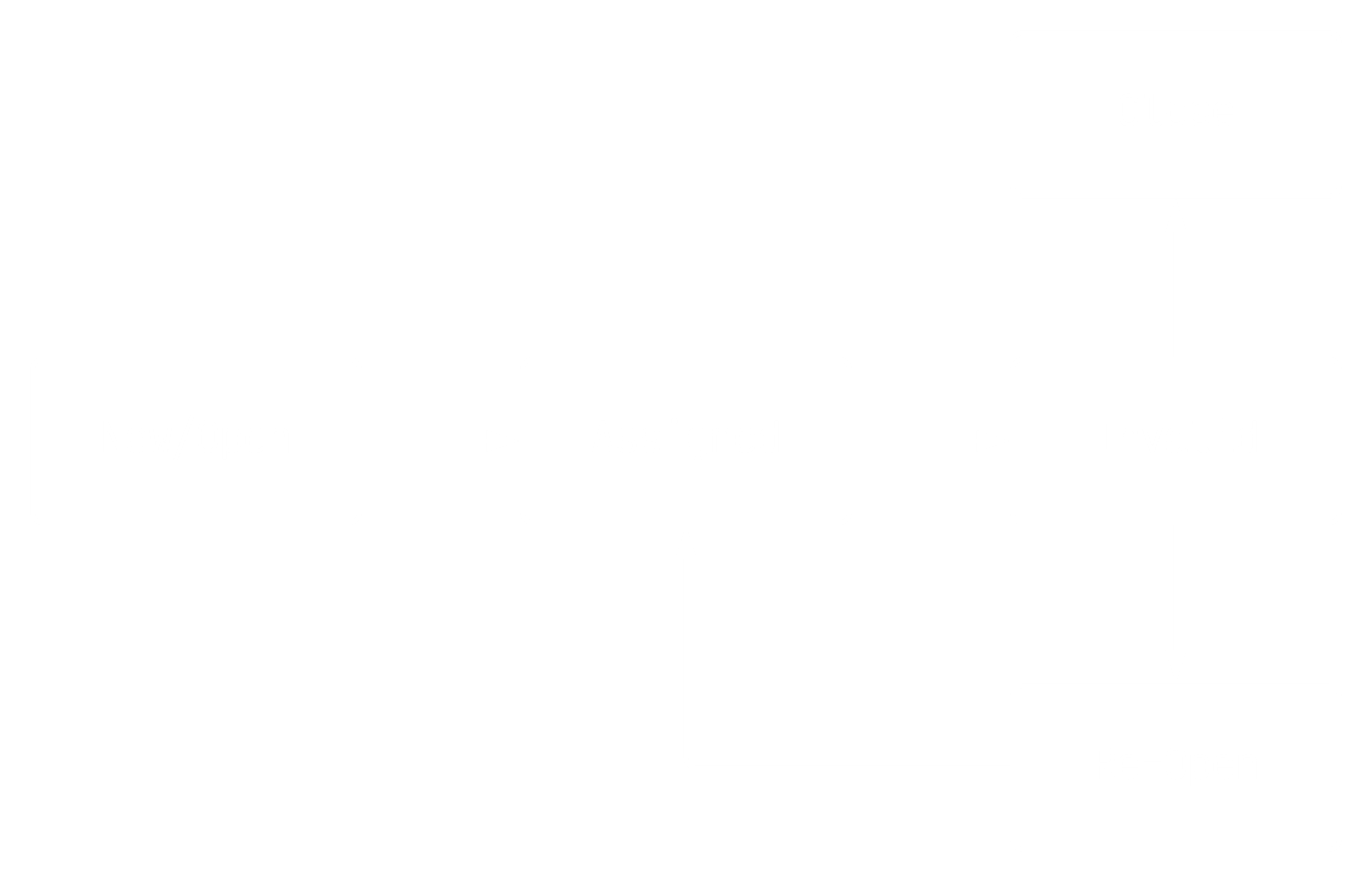


Other than the steps outlined in the diagram above, there are a few more possible statuses a bug can have:



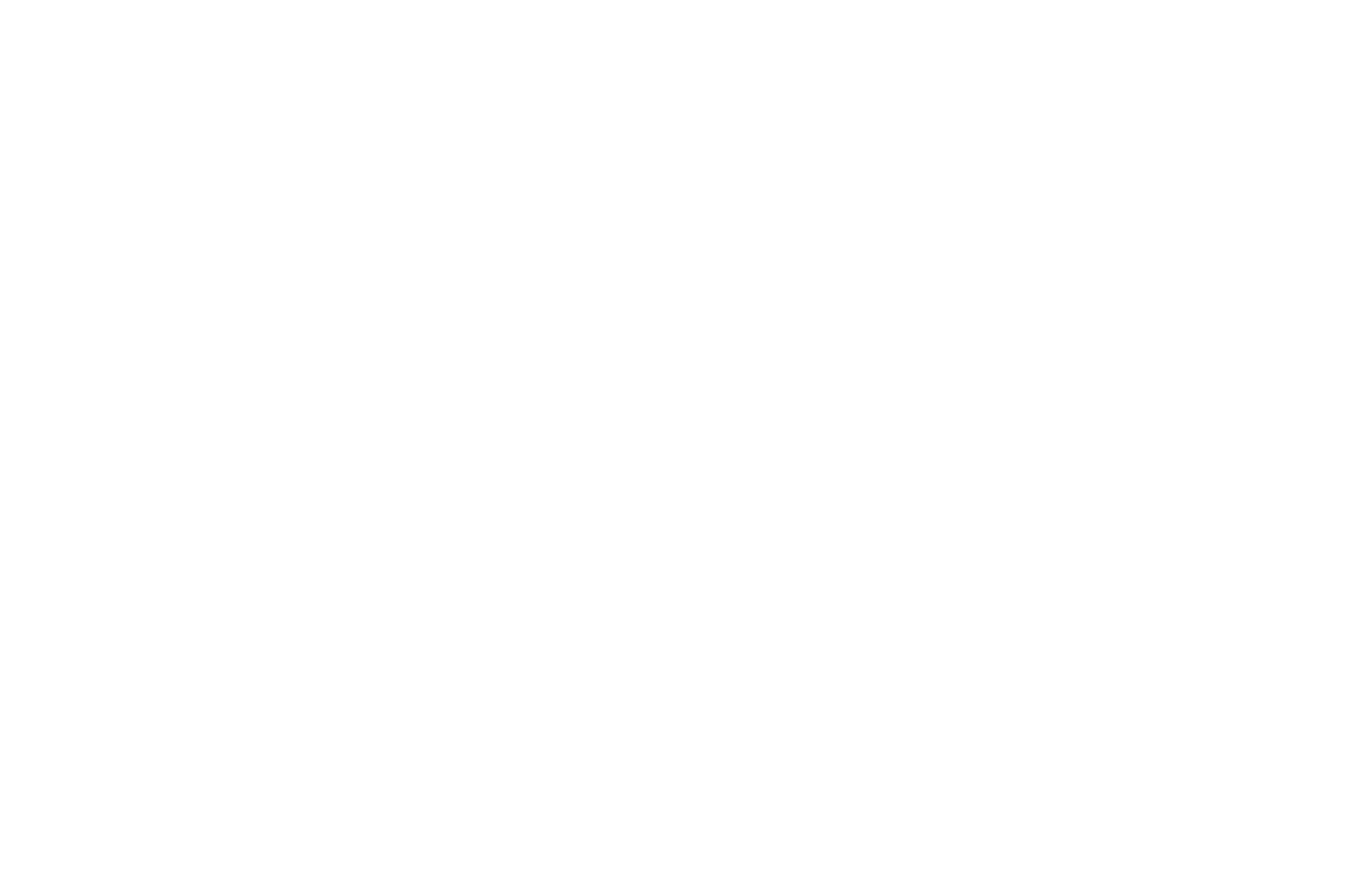
### Invalid Bugs

Any bug that is **not accepted** by the developer is an **invalid bug**. An invalid bug can occur if the **testing engineer** misunderstood requirements or if the **developer** misunderstood requirements. The bug is closed if the testing engineer realizes that they made a mistake and is reopened if they are certain that the developer has made a mistake.



### Duplicate Bug

**Duplicate bugs** occur when the same bug is reported multiple times by different testing engineers. This can happen with **common features** or **dependent modules**.



We can avoid duplicate bugs by checking the **bug repository** before reporting a bug. For new bugs, we should store it in the bug repository for the reference of other testing engineers.

### Not Reproducible Bug

There are cases when, even after going through the **navigation steps** outlined by the testing engineer in the bug report, the developer is **unable to find** the bug. This could be due to several reasons:

1. **Incomplete Bug Report** – The test engineering did not mention all the navigation steps.
2. **Environment Mismatch** – The developer and test engineer are using different servers or platforms.
3. **Data Mismatch** – Different data is being used by the developer and the test engineer.
4. **Build Mismatch** – The developer and test engineer are using different builds. It is possible that a later build does not have the bug.
5. **Inconsistent Bug** – Some bugs are weird and do not occur all the time. There are even bugs that only ever occur once. To deal with accusations of lying, some test engineers even record themselves to prove that a bug really did occur.

### Can’t Fix Bug

A **Can’t Fix** bug is one that has been confirmed to exist, but the **changes to the code** required to fix the bug cannot be made. This could be due to several reasons:

* **No Technology Support** – Perhaps the programming language that has been used does not have the capability to solve the problem.
* **Bug in Framework** – A framework or library being used causes the bug. That code cannot be modified by us, and thus cannot be fixed.
* **Cost** – Sometimes, the cost of fixing a bug is higher than just keeping it.

### Deferred/Postponed

The **deferred/postponed** status is assigned when it is decided that a bug will be fixed in a future release due to time constraints.

### Request for Enhancement

Test engineers may give **suggestions** towards the enhancement of the application in the form of a bug report. The **customer** is consulted about the suggestions, and if they agree, the suggestion is implemented.

## Bug Severity

The **impact** of a bug on the application is known as its **severity**.

* **Blocker** – This is a bug which results in us being unable to proceed to other parts of the application. This causes the test engineer to sit idle.
* **Critical** – This is a bug that causes one of the main functionalities to not work, which means the test engineer cannot continue testing. This applies to a specific module, not the entire application.
* **Major** – This is a bug that causes the supporting components or modules to not work. However, the test engineer can still continue testing.
* **Minor** – This is a bug that causes the UI to not work properly, but testing can proceed without interruption.

## Bug Priority

**Priority** is important to decide which bugs need to be fixed first or which ones need to be fixed quickly.

* **High** – These bugs have major impacts on the customer application. They need to be fixed first.
* **Medium** – These bugs need to be fixed before the release of the current version in development.
* **Low** – These bugs should be fixed if there is time, but they can be deferred to the next release.

Example

|  |  |  |
| --- | --- | --- |
| **Severity** | **Requirement** | **Priority** |
| Critical | Login | P1 |
| Critical | Compose | P1 |
| Critical | Inbox | P1 |
| Major | Send Item | P2 |
| Major | Trash | P3 |
| Minor | Help | P3 |
| Minor | Logout | P4 |

## Bug Reports

Often, bug reports do not have **enough information** or are **inaccurate**. This results in them either being **rejected** or, when reviewed, makes it impossible to tell what the **problem** was.

### Parameters

The following details should be included in a bug report:

* Date of issue, author, approval and status
* Severity and priority
* The associated test case that revealed the problem.
* Expected and actual results.
* Description of the incident, with steps to reproduce
* Status of the incident
* Conclusions, recommendations and approvals

### Characteristics

* **Objective** – Care should be taken to remain objective, non-judgemental and unemotional, e.g. ‘the program crashed’ instead of ‘your program crashed’.
* **Specific** – One report should only log one bug.
* **Concise** – The report should be simple and to-the-point. The report should be reviewed and edited after being written to reduce complexity.
* **Persuasive** – The best reports are ones that make the developer want to fix the bug.
* **Reproducible** – The biggest reasons bug reports are rejected is because the developers could not reproduce it.
* **Explicit** – Reports should state information clearly, or refer to a specific source where the information can be found, e.g. ‘click the Next button’ instead of ‘click the button to continue’.

### Sample

|  |  |
| --- | --- |
| Bug ID |  |
| Module |  |
| Requirements |  |
| Test Case Name |  |
| Release |  |
| Version |  |
| Status |  |
| Reporter |  |
| Date |  |
| Assign To |  |
| CC |  |
| Severity |  |
| Priority |  |
| Server |  |
| Platform |  |
| Build No. |  |
| Test Data |  |
| Attachment |  |
| Brief Description |  |
| Navigation Steps |  |
| Observation |  |
| Expected Result |  |
| Actual Result |  |
| Additional Comments |  |

## Bug Management

Bugs need to be managed in a **methodical** and **systematic** fashion. If we find a bug, we need to make sure that it is fixed at some point. If we fix a bug, we need to make sure this has been recorded. Thus, we should use a **bug tracking system**.

## Test Cases

The **tests** are usually defined in a document called the **Test Plan**, which spells out the details of the testing to be done, schedules and milestones, responsibilities, testing systems and installation and setup, and the list of test cases to be executed.

Test cases can be **manual**, where the tester follows conditions and steps one by one, or **automated**, where a test is written as a program and run against a system.

Tests are usually first defined in a **document** and often have to be **approved** by developers and testers. Automated tests may have all their documentation in the **code** or **comments**.

### Fields

There are no specific rules for test cases, but some common information includes:

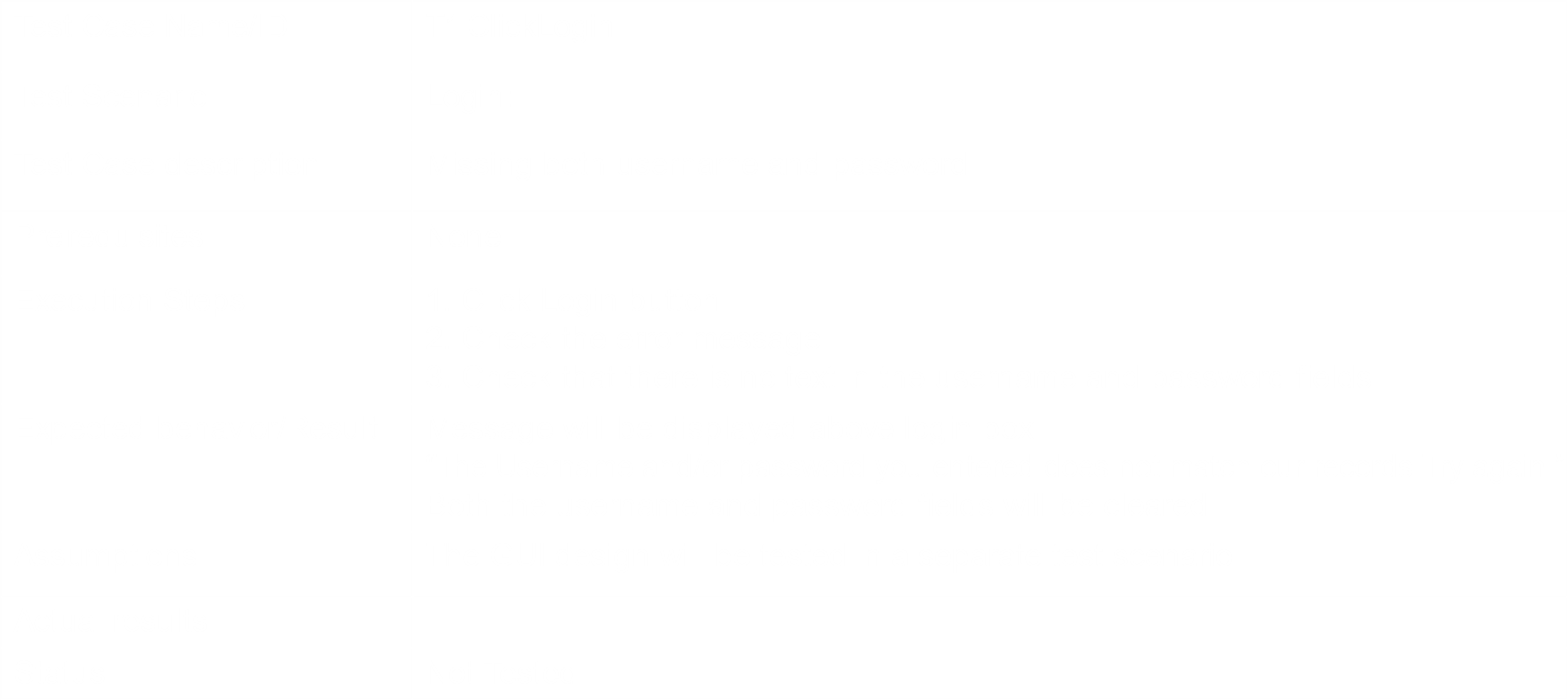
* Unique Test Case Name or ID
* Test Scenario and Summary
* Pre-Requisites and Setup
* Test Data and Input
* Execution Steps
* Expected Behaviour and Results
* Assumptions
* Actual Results
* Status

### Characteristics

* Only test a **single condition** or value per test
* There should be **no confusion** about what the test is supposed to be checking and what the expected behaviour should be.
* The language should be **clear** and **easy to understand**, both for the documentation and the code.
* The test case should be **small**. Long tests indicate that more than one thing is being tested.
* The test case should be **independent**. Test cases should be executable in **any order** and independently from other test cases.
* There should not be **unnecessary steps**.
* Test cases should be **traceable** to requirements.
* Test cases should be **repeatable**.
* Test cases should use **consistent terminology**.

Example

Consider that we have a login page. One possible test case is that the login button is clicked with no username or password provided. The test case would look like this:



## Software Testing Life Cycle

The **Software Testing Life Cycle** (STLC) consists of 6 steps:

* Requirement Analysis
* Test Planning
* Test Case Development
* Environment Setup
* Test Execution
* Test Cycle Closure

### Requirement Analysis

In this step, the testing team studies the **requirements** from a **testing perspective** to identify testable requirements. The QA team may interact with various **stakeholders** to understand requirements in detail.

* Entry Criteria
  + Requirement Specifications
  + Application Architectural View
* Deliverables
  + List of questions with all answers to be resolved from testable requirements
  + Automation feasibility report
* Activities
  + Identify tests to be performed
  + Gather details about testing priorities and focus
  + Prepare Requirement Traceability Matrix (RTM)
  + Identify test environment details where testing is supposed to be carried out
  + Automation feasibility analysis

### Test Planning

In this phase, a senior QA manager determines the **test plan strategy** along with efforts and cost estimates for the project. Moreover, the resources, test environments, test limitations and testing schedule are also determined.

* Entry Criteria
  + Updated requirements document
  + Test feasibility reports
  + Automation feasibility report
* Deliverables
  + Test plan or strategy document
  + Effort estimation document
* Activities
  + Preparation of test plan or strategy document for various types of testing
  + Test tool selection
  + Test effort estimation
  + Resource planning and determining roles and responsibilities
  + Training requirements

### Test Case Development

The **test data** is identified, created and reviewed. The QA team starts **development** process of test cases for individual units.

* Entry Criteria
  + Reviewed and signed test cases
  + Reviewed and signed data
* Deliverables
  + Test cases
  + Test data
* Activities
  + Create test cases and automation scripts
  + Review and baseline test cases and scripts
  + Create test data

### Environment Setup

In this stage, the **software** and **hardware** needed for testing teams to execute test cases is setup. It supports test execution with hardware, software and network configured.

* Entry Criteria
  + Environment setup is working as per plan and checklist
  + Test data setup complete
* Deliverables
  + Environment ready with test data setup
  + Smoke test results
* Activities
  + Understand the required architecture, environment setup and prepare hardware and software requirement list for the test environment
  + Setup test environment and test data
  + Perform smoke tests on the build

### Test Execution

In this stage, code is executed and the expected and actual results are compared. The test analysts start executing the test scripts based on test strategy allowed.

* Entry Criteria
  + Test Cases
  + Test Scripts
* Deliverables
  + Completed RTM with execution status
  + Test cases updated with results
  + Bug reports
* Activities
  + Execute tests as per plan
  + Document test results and log bugs for failed cases
  + Map bugs to test cases in RTM
  + Retest bug fixes
  + Track the bugs to closure

### Test Cycle Closure

This stage involves calling out the testing team member meeting and evaluating cycle completion criteria based on Test coverage, quality, cost, time, critical business objectives and software.

* Entry Criteria
  + Test Closure Conditions
  + Test Summary Report
* Deliverables
  + Test Closure Report
  + Test Metrics
* Activities
  + Evaluate cycle completion criteria based on time, test coverage, cost, software, critical business objectives and quality
  + Prepare test metrics based on the above parameters
  + Document the learning out of the project
  + Prepare the test closure report
  + Test result analysis to find the defect distribution by type and severity

## Testing Categories and Techniques

There are several testing categories, each of which can have multiple testing techniques:

|  |  |
| --- | --- |
| **Testing Category** | **Techniques** |
| Dynamic Testing: Black-Box | Boundary Value Analysis, Equivalence Class Partitioning, Stable Table-Based Testing, Decision Table-Based Testing, Cause-Effect Graphing Technique, Error Guessing |
| Dynamic Testing: White-Box | Basis Path Testing, Graph Matrices, Loop Testing, Data Flow Testing, Mutation Testing |
| Static Testing | Inspection, Walkthrough, Reviews |
| Validation Testing | Unit Testing, Integration Testing, Function Testing, System Testing, Acceptance Testing, Installation Testing |
| Regression Testing | Selective Retest Technique, Test Prioritization |

## Black-Box Testing

A **Black-Box Test** is one where we cannot see the internals of a system. We only give the system some input, and check the output. This technique only considers the **functional requirements** of the system.

An example of a black-box test is a User Acceptance Test, where the tester does not have access to the code and is not concerned about how the system was implemented. They are just testing that the system meets the user requirements and works as expected for typical input.

Black-box testing attempts to find errors in the following categories:

* To test the modules independently
* To test the functional validity of the software so that incorrect or missing functions can be recognized
* To look for interface errors
* To test the system behaviour and check its performance
* To test the maximum load or stress on the system
* To test the system such that the user/customer accepts the system within defined acceptable limits

### Techniques

Some black-box testing techniques are:

* Boundary Value Analysis (BVA)
* Equivalence Class Testing (ECT)
* State Table-Based Testing (STBT)
* Decision Table-Based Testing (DTBT)
* Cause-Effect Graphing Based Testing (CEGBT)
* Error Guessing (EG)

### Boundary Value Analysis

Test cases that check **boundary input values** have a high chance of finding errors, since most errors occur due to boundary values. Boundary values refer to the maximum or minimum values within the input domain.

For example, if the acceptable input for an integer is between 10 and 255, one set of checks should be for the values 9, 10 and 11, while the other set of checks should be for 254, 255 and 256. Notice that we need to check for values at exactly the boundary, just within the boundary and just outside the boundary.

There are several methods that perform boundary value analysis:

* Boundary Value Checking (BVC)
* Robustness Testing Method
* Worst-Case Testing Method

In **Boundary Value Checking**, we place just one variable at one of the boundary values, while the others are at their normal values, also called **nominal** values. Possible combinations for a two variable set could be:

1. ,
2. ,
3. ,
4. ,
5. ,
6. ,
7. ,
8. ,
9. ,

For variables, we will have test cases.

The **Robustness Testing Method** is an extension of Boundary Value Checking. In BVC, we did not check for values just above the maximum and just below the minimum, so here we do that.

1. ,
2. ,
3. ,
4. ,

This will result in a total of test cases.

The **Worst-Case Testing Method** works off of BVC as well, but assumes that multiple variables can be on the boundary at the same time. Thus, on top of the 9 test cases we saw in the BVC example, we also add these test cases:

10. ,

11. ,

12. ,

13. ,

14. ,

15. ,

16. ,

17. ,

18. ,

19. ,

20. ,

21. ,

22. ,

23. ,

24. ,

25. ,

In this case, there are test cases in total.

### Decision Table-Based Testing

The specialty of using a **decision table** is that it can represent **complex combinations** of inputs and results.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Condition Stubs |  | Rule 1 | Rule 2 | Rule 3 | Rule 4 | … |
| C1 | True | True | False | - |  |
| C2 | False | True | False | True |  |
| C3 | True | True | True | - |  |
| Action Stubs | A1 |  | X |  |  |  |
| A2 | X |  |  | X |  |
| A3 |  |  | X |  |  |

The **Condition Stub** section defines the state of each possible condition. A specific combination of states is called a **Rule**. For each rule, we will take a certain action, as defined in the **Action Stub** section. For example, if C1 and C3 are true but C2 is false, this is Rule 1. For Rule 1, the action A2 should be taken.

Essentially, we are checking **if-else statements**. The conditions define which section of the if-else statements should execute, while the actions define the expected resulting behaviour.

For conditions and rules, there will be at least test cases (one for each rule) and at most test cases.

Example

Say we have a program that calculates the salary of employees. If an employee works over 48 hours, the salary is 1.25 times the normal one. If they are working on a holiday or a Sunday, the salary is 2 times the normal one.

In this scenario, there are 2 conditions, which gives us 3 rules in total, since the number of hours an employee works on a holiday or a Sunday is not relevant.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Condition Stubs |  | Rule 1 | Rule 2 | Rule 3 |
| C1: Working Hours > 48 | - | False | True |
| C2: Holidays or Sundays | True | False | False |
| Action Stubs | A1: Normal Salary |  | X |  |
| A2: 1.25x of Salary |  |  | X |
| A3: 2x of Salary | X |  |  |

Based on this, we can create our test cases:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case ID** | **Working Hour** | **Day** | **Expected Result** |
| 1 | 48 | Monday | Normal Salary |
| 2 | 50 | Tuesday | 1.25x of Salary |
| 3 | 52 | Sunday | 2x of Salary |

## White-Box Testing

**White-Box Testing** involves getting into the code instead of just giving it some input and checking for an output. For example, **Unit Testing**, which we saw earlier, is an example of White-Box Testing, since the developer writes the test case based on their own code.

White-Box Testing allows us to test modules right from the start. It can reveal bugs that Black-Box Testing cannot, such as errors from the design phase and typographical errors. All logical paths can be tested.

There are three basic forms of **logic coverage**:

* **Statement Coverage** – It is assumed that if all the statements of a module are executed once, ever bug will be revealed.
* **Decision or Branch Coverage** – Every possible branch must be traversed at least once.
* **Condition Coverage** – Each condition in a decision must take on all possible values at least once.

Consider that we have this code and we want to use **Statement Coverage**.

scanf("%d", &x);  
scanf("%d", &y);  
while (x != y)  
{  
 if (x > y) x = x - y;  
 else y = y - x;  
}

printf("x = ", x);  
printf("y = ", y);

C

Depending on the values of and we provide, it is possible that some branches will remain **unexplored**. Thus, not all the statements are being executed. We need to write test cases in a way so that **all statements** are executed at least once. This will involve running different tests to match the different possible conditions.

If we have two test cases, once where and another where , we will technically only have tested the **while loop**. We need two more test cases, one where and another where to test the **if-else statements**.

Using the same code, but just thinking of it in terms of whether or not all possible **branches of code** are executing, we can use **Branch Coverage**. For example, the above code has **three possible branches**.

Again, if we think of it in terms of the **conditions** and try to ensure that every possible variation of the conditions is being checked, we are using **Condition Coverage**. For example, in the above code, the **while statement** has **two conditions** and the **if statement** has **two conditions**. If the if statement had two parts, there would be four conditions there.

For **Statement Coverage** specifically, we have some calculation to figure out how much of the code is tested.

Statement Coverage Number of executed statements Total number of statements

For each possible test case, we are covering a certain portion of the statements. In the above code, for the test case where , the else statement is not executed. Thus, we have coverage.

Statement Coverage covers:

1. Unused statements
2. Dead code
3. Unused branches
4. Missing statements

## Basis Path Testing

In **Basis Path Testing**, we try to test **all possible logical paths** for an **entire program**. To do this, we use a **flow graph**. Basis Path Testing is a form of **White-Box Testing**.

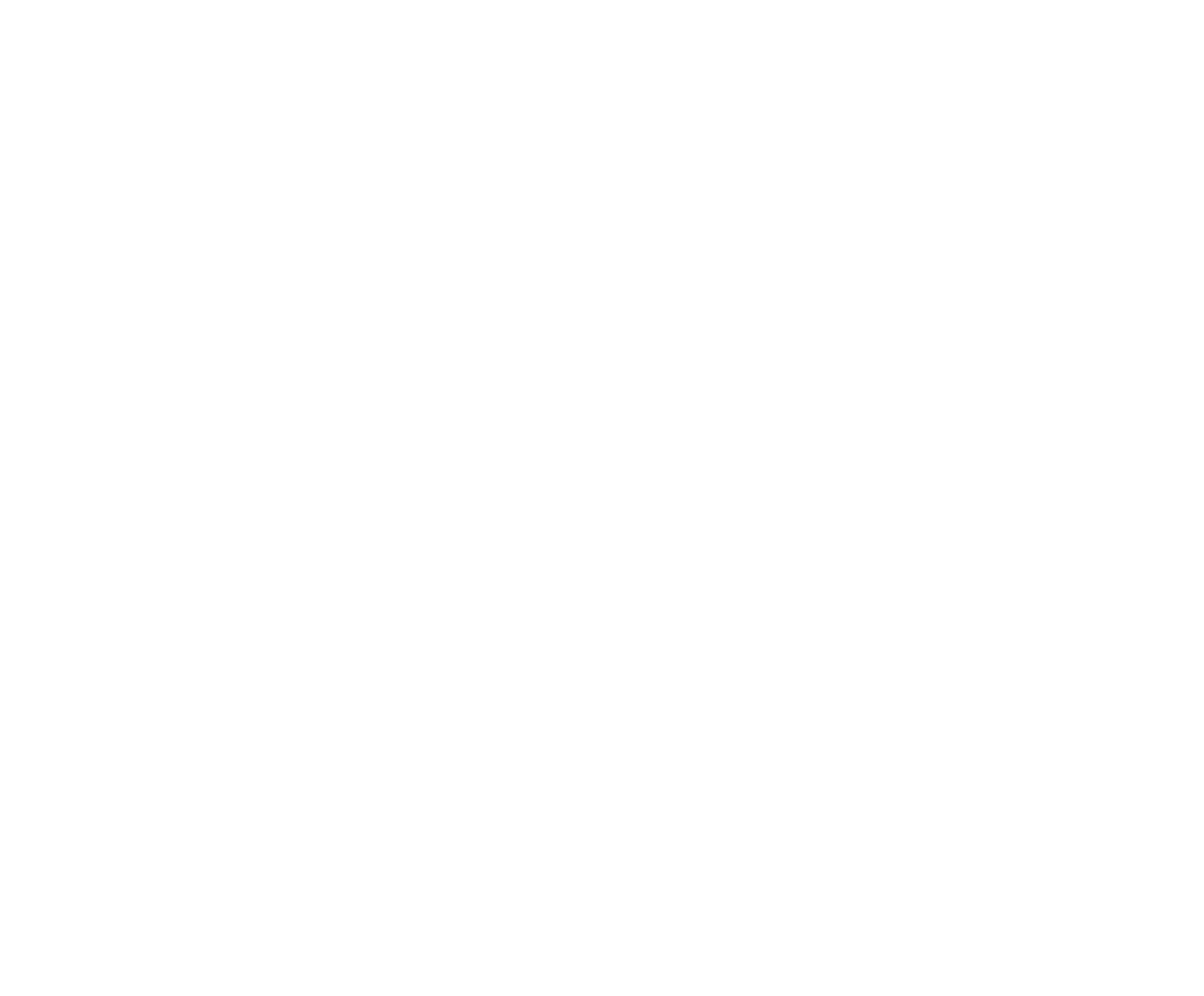
Basis Path Testing is more **general** than the other coverage criteria we have seen and it is useful for detecting more errors. One problem however, is that some programs, such as ones that have loops, can have an **infinite** number of possible paths, which makes it impractical to test all the paths.

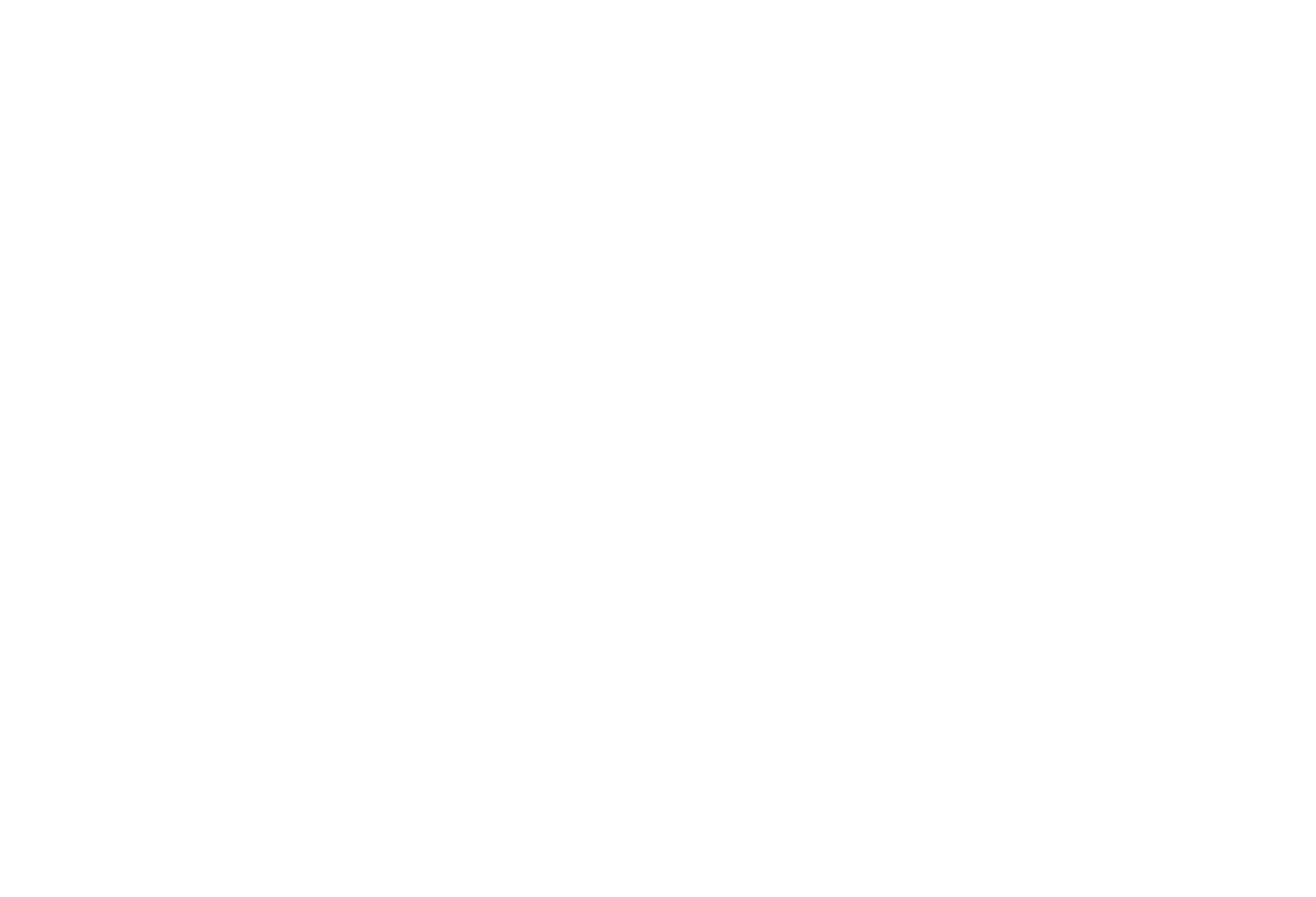
The steps for Basis Path Testing are:

1. Draw a **control flow graph**, which determines the different program paths.
2. Calculate the **Cyclomatic Complexity**, which determines the number of independent paths.
3. Find a **basis set of paths**, which is the set of independent paths.
4. Generate **test cases** to execute each independent path.

### Control Flow Graphs

A **Control Flow Graph** represents the control structure of a program. It has nodes or **vertices** and **edges**. The nodes can be **junction nodes**, which have multiple arrows entering it, or **decision nodes**, which have multiple arrows leaving it. The graph also has **regions**, which are areas bounded by edges and nodes. The area outside the graph is also considered a region.





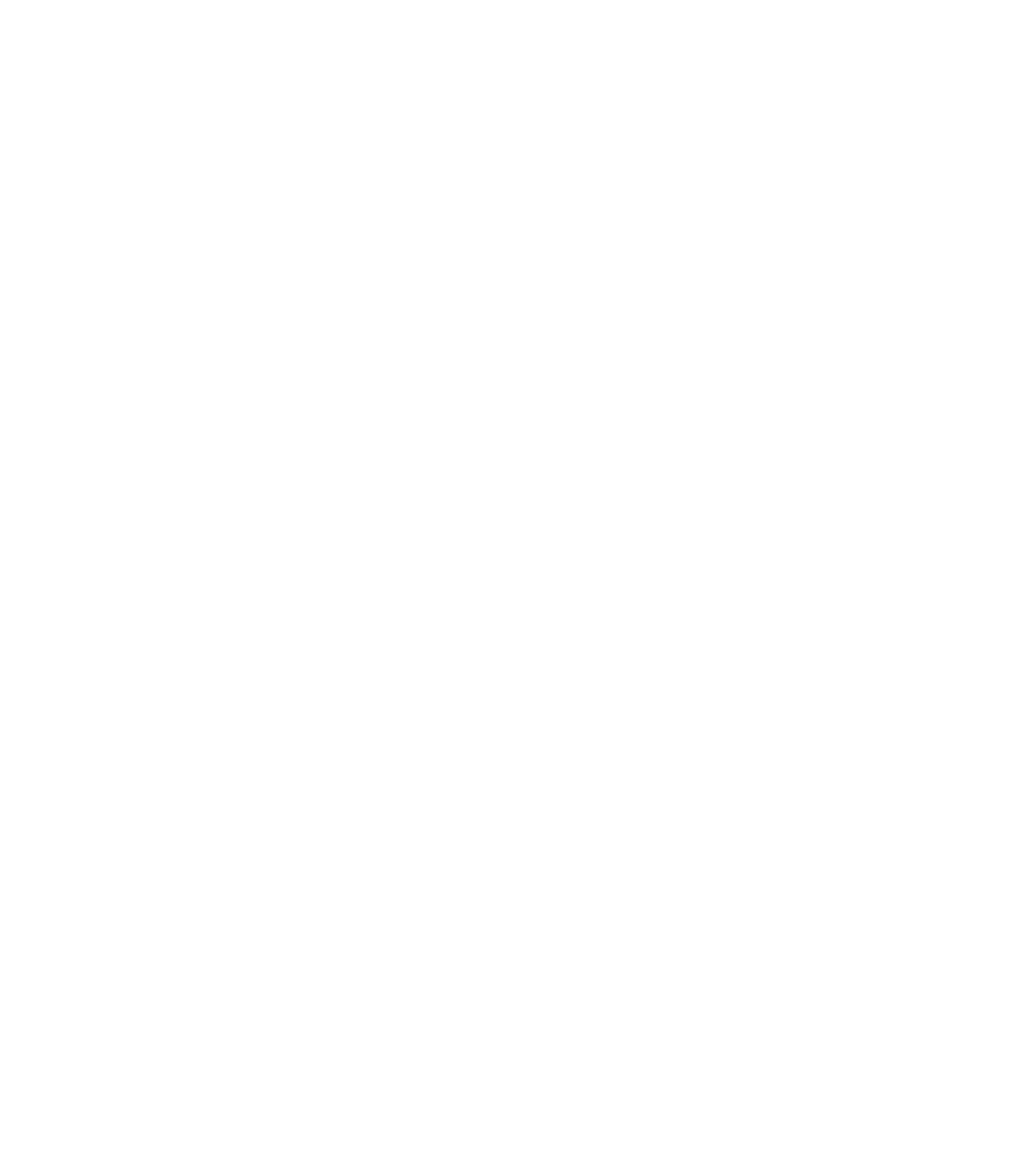
### Cyclomatic Complexity

The **Cyclomatic Complexity**, , measure the **logical complexity** of a program. It can be calculated using one of the following formulae:

1. , where is the number of edges, is the number of nodes and is the number of connected components, which is the number of modules. The value of is if we are testing just one module.
2. , where is the number of decision-nodes. Note that the value of is determined as for each decision node, where is the number of arrows leaving the node. A decision node with arrows leaving is technically two decision nodes combined, which is why .
3. number of regions

### Independent Paths

An **independent path** is one which introduces at lease one **new edge** that is not traversed before the path is defined. The **Cyclomatic Complexity** gives us the number of independent paths in out graph.



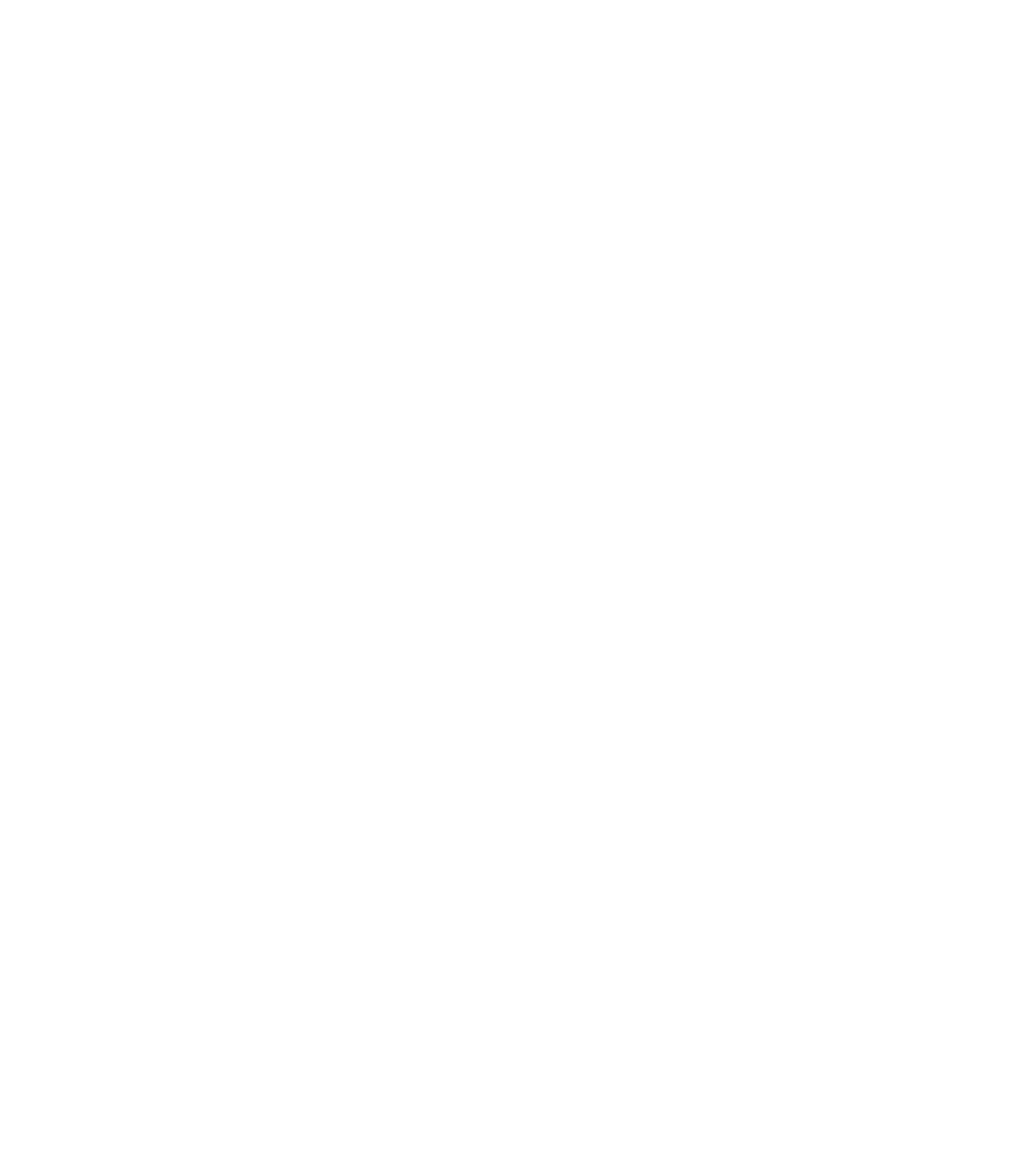
This graph for example, has paths, , , , , and . However, only are independent. This can be a little confusing.

, and are all independent, since each time, we are getting at least **one new edge** in the path that we did not have in any of the paths before. Similarly, can be considered an independent path. However, the other two paths are not independent, since although the path as a whole is a **unique combination of edges**, they do not force us to traverse any edges that we **did not traverse before**. This is one interpretation of it, but have some freedom with choosing exactly which paths we consider. The total number will still be in this case.

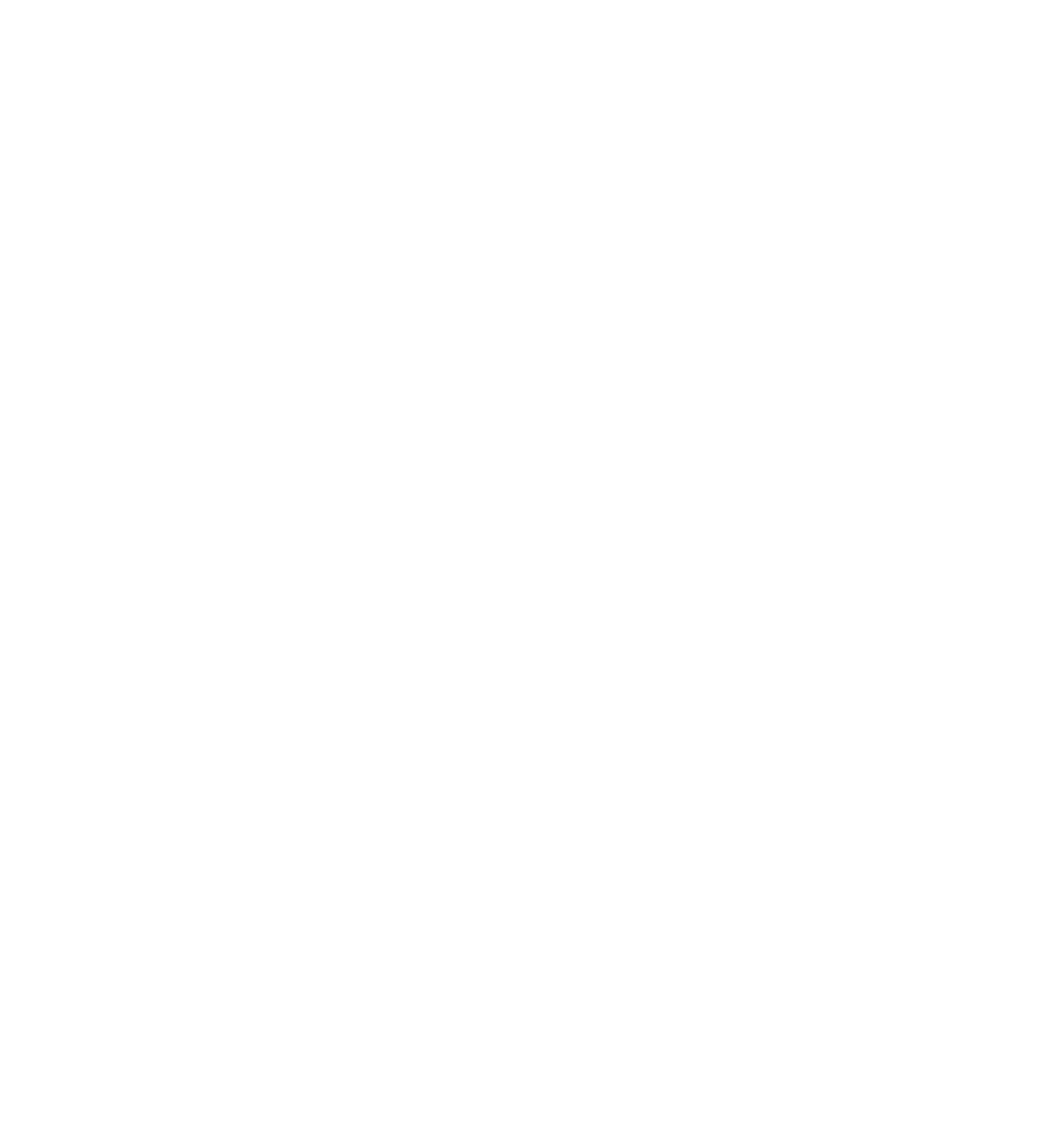
A more complicated example could be the one below, which has **two components**.

int main()  
{  
 int number();  
 int fact();  
 clrscr();  
 printf("Enter the number whose factorial is to be found out");  
  
 scanf("%d", &number);  
 if (number < 0) printf("Factorial cannot be defined for this number.");  
 else printf("Factorial is %d", fact(number));  
}  
  
int fact(int number)  
{  
 int index;  
 int product = 1;  
 for (index = 1; index <= number; index++) product = product \* index;  
 return product;  
}

C



## Validation Activities



The testing methods on the far-right of the diagram above are combinedly called **Validation Tests**.

## Integration Testing

In **Integration Testing**, smaller units are integrated into larger units, and larger units are combined into the overall system. The focus here is not on the individual units, but on the **interaction** between them. The purpose is to verify the **functional performance** and **reliability** between the modules that are integrated.

We perform integration testing because:

* Unit test only test units in **isolation**
* Many failures result from faults in **interaction** between subsystems
* Often, **off-the-shelf** components are used, which cannot be unit tested
* Without integration tests, **system tests** become **time consuming**

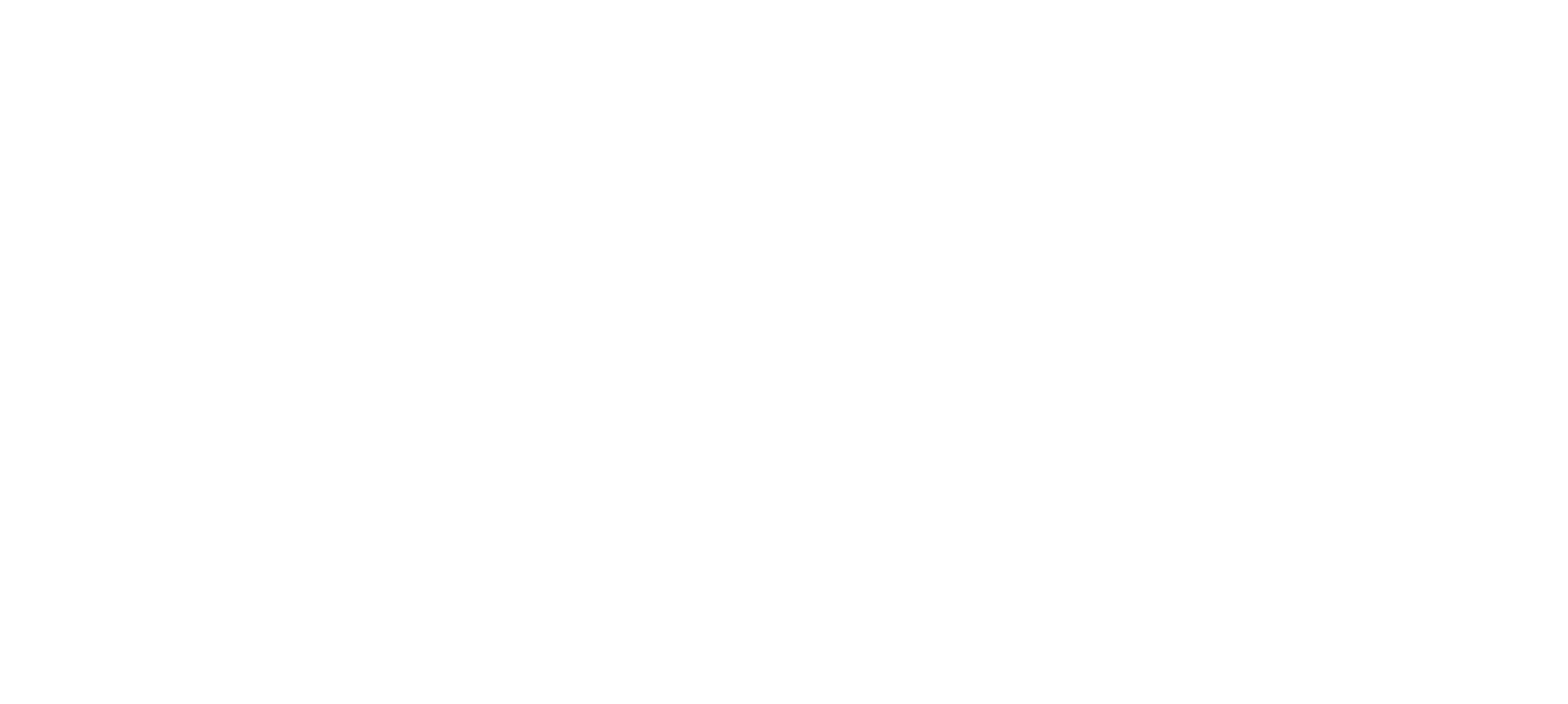
There are three approaches to integration testing:

1. Decomposition-Based Integration Testing
2. Call Graph-Based Integration Testing
3. Path-Based Integration Testing

Each of these have different methods.

### Big Bang Approach

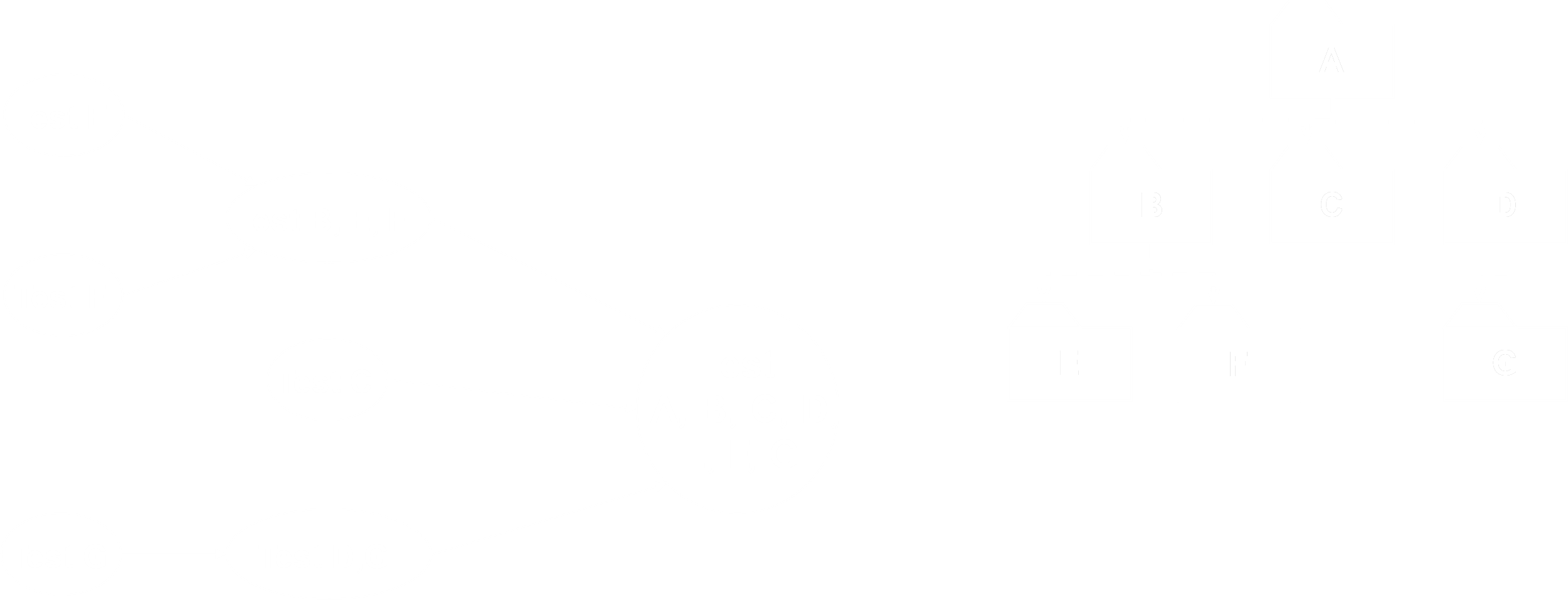
The **Big Bang Approach** is a **Decomposition-Based Integration Testing** approach. Under this approach, each of the subsystems are tested and then one massive test is done where all the subsystems are tested together.



This is a bad idea, since the **interfaces** between the subsystems have not been tested yet.

### Bottom-Up Testing Strategy

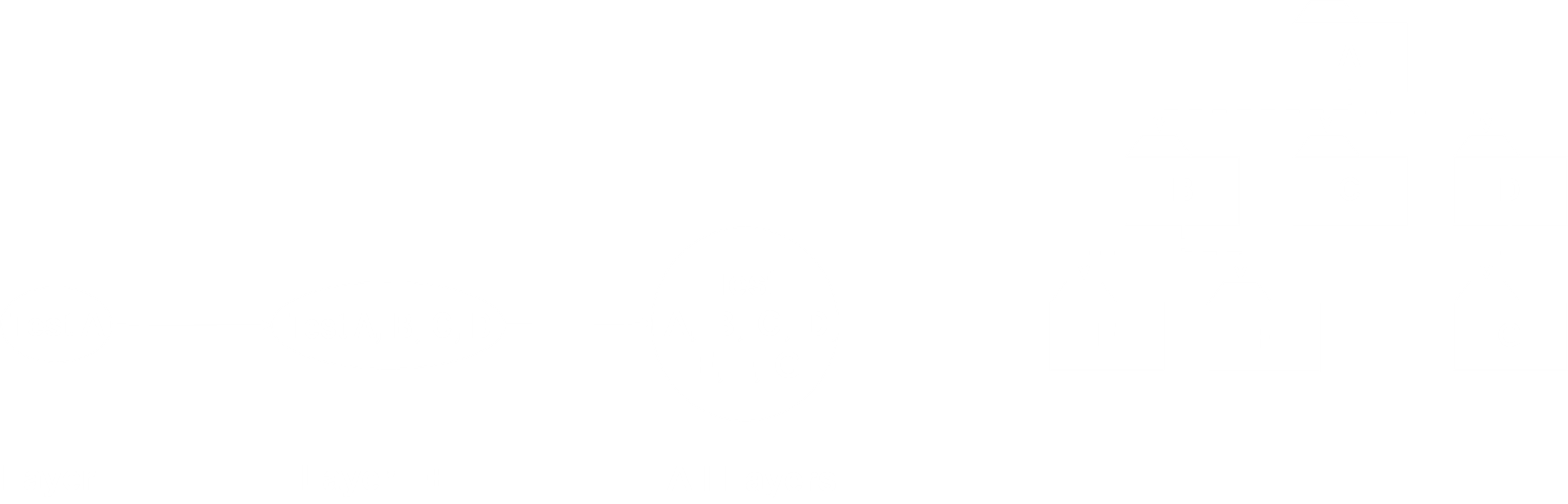
The **Bottom-Up Testing Strategy** is another **Decomposition-Based Integration Test**. Here, we test the subsystems in the lowest layer of the call hierarchy first, and then test the ones above that which call the lowest subsystems and so on.



This process requires **drivers**.

### Top-Down Testing Strategy

The **Top-Down Testing Strategy** is the opposite of the previous testing strategy, so it is also a **Decomposition-Based Integration Test**. We test the top-layer components first and go down the tree, testing increasingly large collections of subsystems together.



This process requires **stubs**.

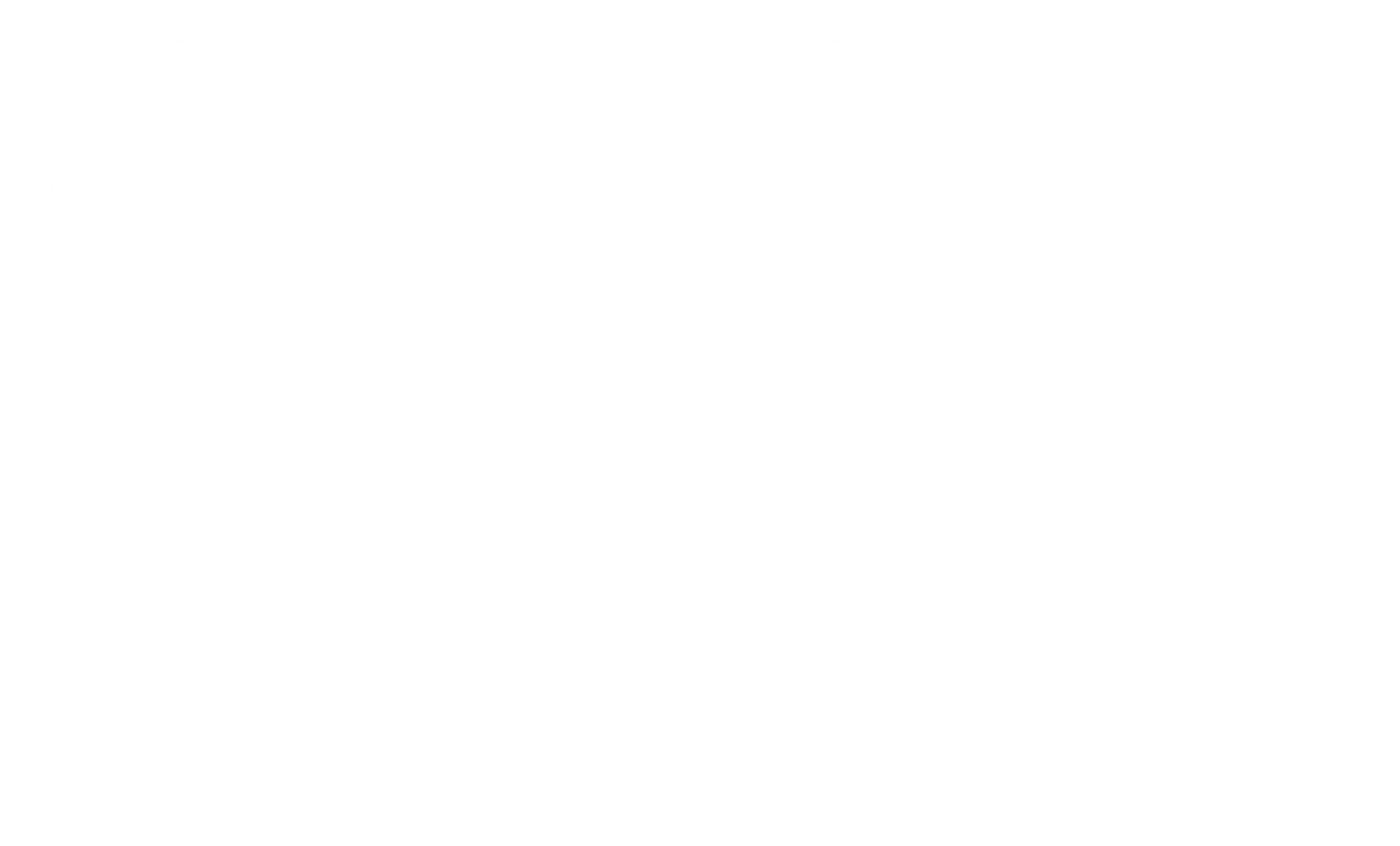
|  |  |  |
| --- | --- | --- |
| **Issue** | **Top-Down Testing** | **Bottom-Up Testing** |
| Architectural Design | It discovers errors in high-level design, thus detecting errors at an early stage. | High-level design is validated at a later stage. |
| System Demonstration | Since we integrate the modules from top to bottom, the high-level design slowly expands as a working system. Therefore, the feasibility of the system can be demonstrated. | It may not be possible to show the feasibility of the design. However, if some modules are already built as reusable components, then it may be possible to produce some kind of demonstration. |
| Test Implementation | () stubs are required for the subordinate modules. | () test drivers are required for subordinate modules to test the lower-level modules. |

### Call Graph Based Integration Testing

A **Call Graph** is a **directed graph** that has **modules** or **units** as **nodes**. A directed edge from one node to another means that one module is **calling** the other.



The call graph can be described using an **adjacency matrix**.



The purpose behind using a call graph is to **avoid developing stubs and drivers**.

There are two types of integration tests we can perform using a Call Graph:

1. **Pair-Wise Integration Testing** – For **each pair of interactions**, e.g. , , etc. in the graph above, we perform a separate integration test. Thus, the number of tests will be equal to the **number of edges** in the graph.
2. **Neighbourhood Integration Testing** – The **neighbourhood** for a node is defined as all the nodes that are **immediate predecessors and successors**. If we create a **neighbourhood table** based on this, for the above graph, it would look like this:

|  |  |  |
| --- | --- | --- |
| Node | Neighbourhoods | |
| Predecessors | Successors |
| 1 | - | 2, 3, 4, 5, 10, 11 |
| 5 | 1 | 6, 7, 8 |
| 8 | 5 | 9, 16 |
| 9 | 8 | 17 |
| 10 | 1 | 13, 14 |
| 11 | 1 | 12 |
| 12 | 11 | 18 |
| 13 | 10 | 19 |
| 14 | 10 | 15 |
| 15 | 14 | 20 |

Notice how we do not include nodes that are already present in other neighbourhoods unless their neighbourhood has some new node. Essentially, this leaves out just the **sink nodes**, nodes that terminate the program. Thus, the number of test cases is for the graph above. This is significantly less than what we had for pair-wise integration testing.

### Path-Based Integration Testing

**Path-Based Integration Testing** is a little complex, so we need to understand a few terms before we dive into it.

A **source node** is one where execution starts or resumes. The node to which control is transferred when the module is called by another module is also a source node.

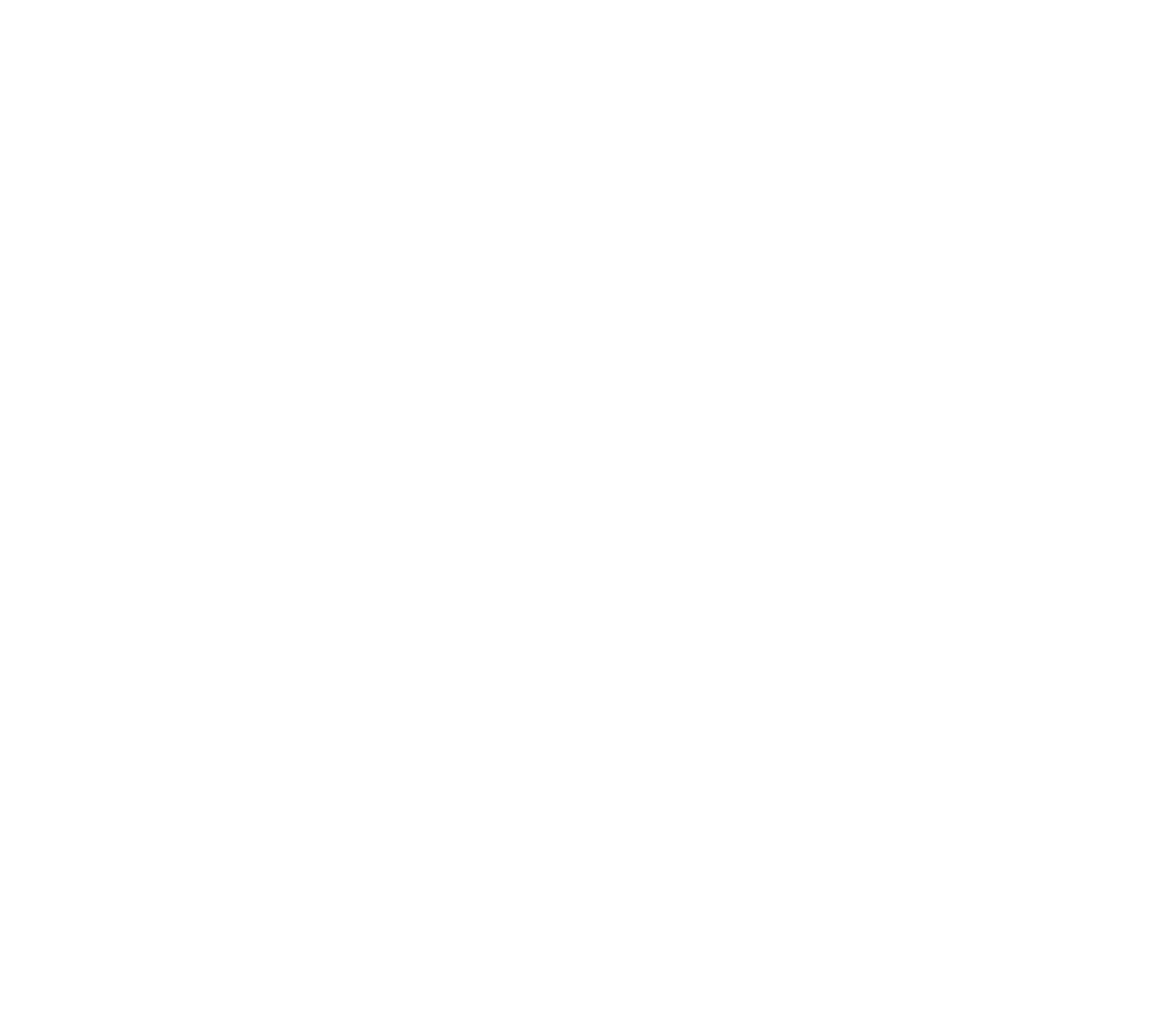
A **sink node** is one where execution terminates. The node from which control is transferred when another module is called is also a sink node.

A **module execution path** (MEP) is a sequence from a source node to a sink node, with no intervening sink nodes.

A **message** is a mechanism by which one unit transfers control to another and acquires a response.

An **MM-Path** is a path consisting of MEPs and messages.

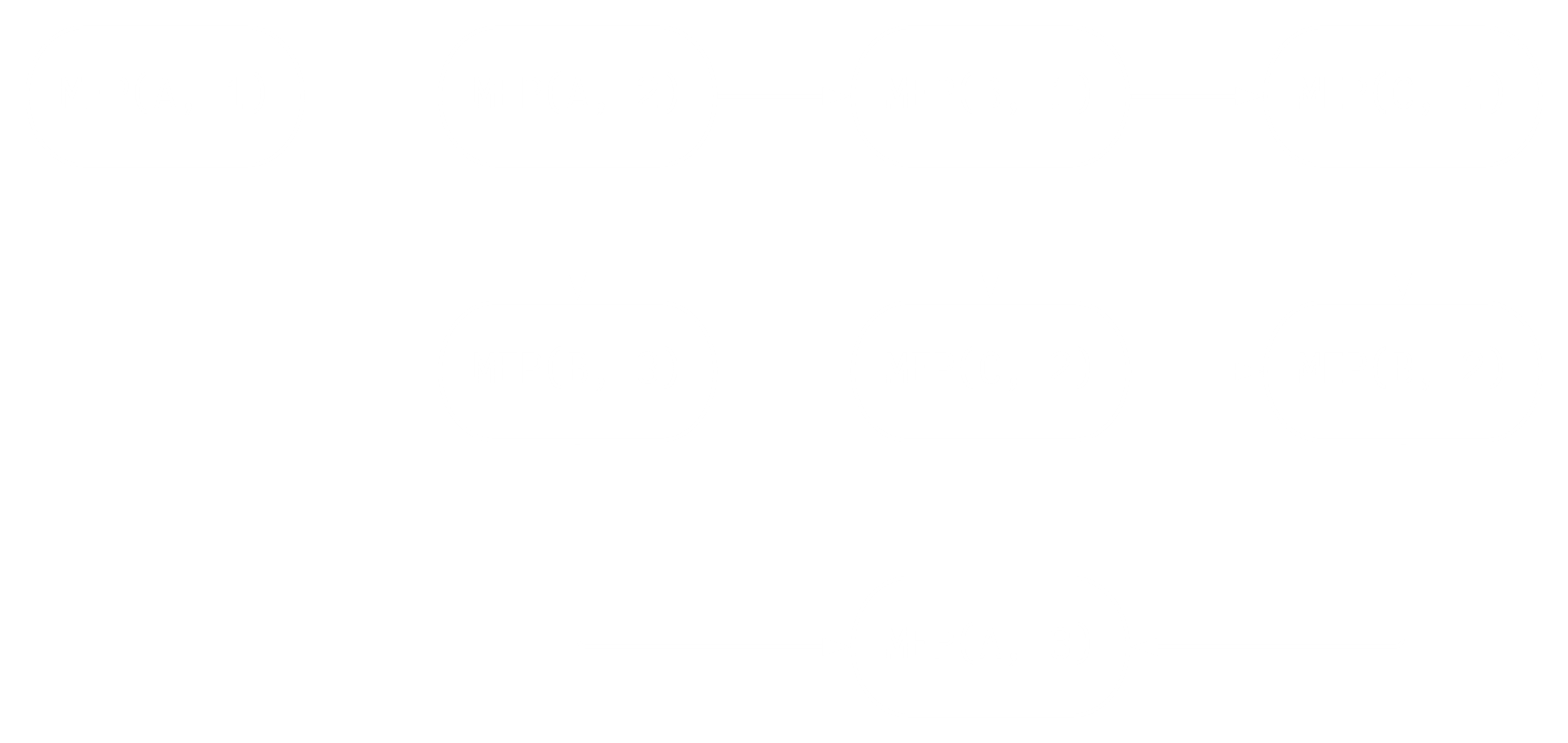
An **MM-Path Graph** is an extended flow graph where nodes are MEPs and edges are messages.



|  |  |  |  |
| --- | --- | --- | --- |
|  | Source Nodes | Sink Nodes | MEPs |
| Unit A | 1, 4 | 3, 5 | MEP |
| MEP |
| MEP |
| Unit B | 1, 5 | 4, 6 | MEP |
| MEP |
| MEP |
| Unit C | 1 | 5 | MEP |
| MEP |

The **source nodes** and **sink nodes** should be easy enough to identify. In Unit A, we start one **execution path** at node 1 and at node 3 we switch to a different module. Thus, the path from node 1 to node 3 forms one **MEP**. Similarly, we have other MEPs.

From this, we can create an **MM-Path Graph**. When we switch from one module to another, we are essentially **ending** one MEP and **starting** another MEP. We can connect these MEPs together to create the graph.



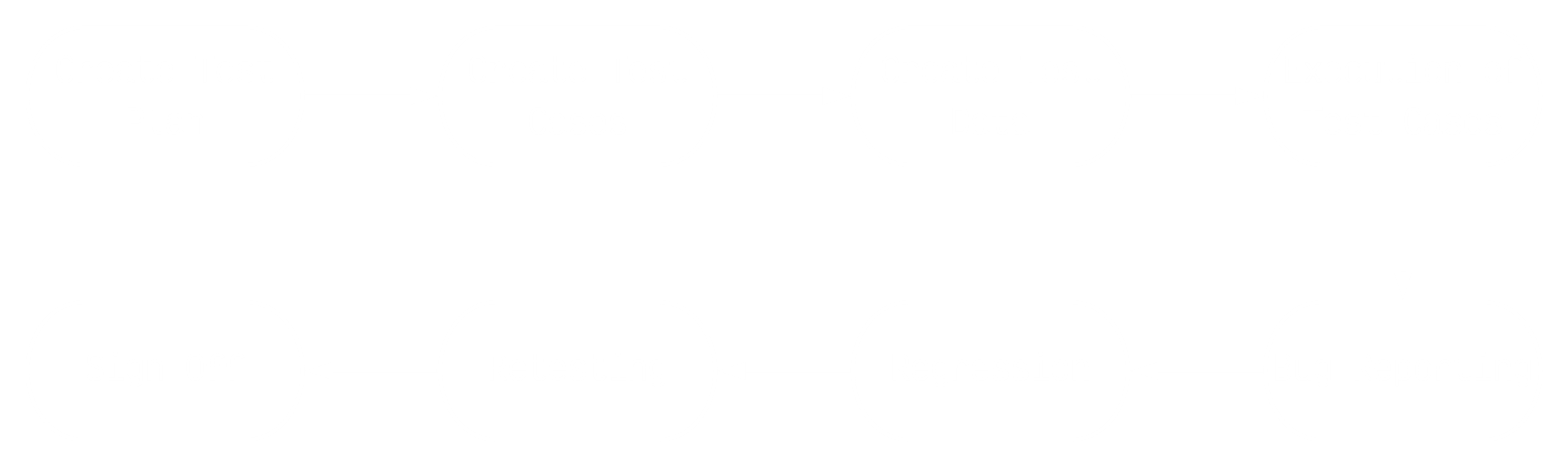
## System Testing

In **System Testing** we test the system as a whole. The tests can be based on functional and non-functional requirements, on the grounds of things like performance, security, maximum load, etc.

In system testing, we provide some input and check if we get the expected output. Thus, this is a form of **black-box testing**.

### Steps

1. Create a **Test Plan**.
2. Create **System Test Cases** and **Test Scripts**.
3. Prepare the **Test Data** required for this testing.
4. **Execute** the system test cases and script.
5. Report the **bugs**. **Re-test** the bugs once fixed.
6. Perform **regression testing** to verify the impact of the change in the code.
7. **Repeat** the testing cycle until the system is ready to be deployed.
8. **Sign off** from the testing team.



### Test Cases

We have already seen how to write **test cases**. System test cases cover **all possible scenarios** and use cases. They cover functional, non-functional, user interface and security related test cases.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Test Case ID | Test Suite Name | Description | Steps | Test Data | Expected Result | Actual Result | Pass/Fail | Remarks |
| TC\_01 | ABC |  |  |  |  |  |  |  |
| TC\_02 |  |  |  |  |  |  |  |  |

## Acceptance Testing

In the **User Acceptance Testing** stage, testing is performed by the **client** to verify that the application developed is per their requirements. This is the formal testing done to verify that **acceptance criteria** have been satisfied.

In UAT, the main purpose is to verify **end-to-end business flow**. It does not focus on cosmetic errors, spelling mistakes or system testing. UAT is carried out in a **separate environment** with **production-like data** setup. It is a type of **black-box testing** involving two or more end-users.

UAT is needed because developers may have **misunderstood** requirements and also because **requirements change** during the project and may not have been effectively communicated with the developers.

### Entry Criteria

* System testing is complete and defects identified are either fixed or documented.
* Acceptance plan is prepared and resources have been identified.
* Test environment for the acceptance testing is available.

### Exit Criteria

* Acceptance decision is made for the software.
* In case of any warning, the development team is notified.

### UAT Process

* Analysis of Business Requirements
* Creation of UAT test plan
* Identify Test Scenarios
* Create UAT Test Cases
* Preparation of Test Data (Production-like Data)
* Run the Test cases
* Record the Results
* Confirm business objectives

There are two types of UAT:

* **Alpha Testing** - Tests are conducted at the development site by the end users. The test environment can be controlled a little in this case.
* **Beta Testing** - Tests are conducted at the customer site and the development team does not have any control over the test environment.

## Alpha Testing

* This test takes place at the **developer’s site**. Developers **observe the users** and **note problems**.
* Alpha testing is testing of an application when **development is about to complete**.
* **Minor design changes** can still be made as a result of alpha testing.
* Alpha testing is typically performed by a group that is **independent of the design team**, but still **within the company**, e.g. in-house software test engineers, or software QA engineers.

### Entry Criteria

* All features are complete/testable (no urgent bugs).
* High bugs on primary platforms are fixed/verified.
* 50% of medium bugs on primary platforms are fixed/verified.
* All features have been tested on primary platforms.
* Performance has been measured/compared with previous releases (user functions).
* Usability testing and feedback (ongoing).
* Alpha sites are ready for installation.

### Exit Criteria

* Get responses/feedbacks from customers.
* Prepare a report of any serious bugs being noticed.
* Notify bug-fixing issues to developers.
* Make sure that **no more additional features** can be included
* Sign off on Alpha testing

### Advantages

* Provides better view about the reliability of the software at an early stage
* Helps simulate real time user behavior and environment.
* Detect many showstopper or serious errors

### Disadvantages

* In depth functionality of the software cannot be tested as it is still under development stage.

## Beta Testing

* Also known as **field testing**.
* Takes place in the **customer’s environment** and involves some extensive testing by a group of customers who use the system in their environment.
* Beta Testing is performed by **real users** of the software application in a **real environment** and can be considered as a form of external User Acceptance Testing.
* Beta version of the software is released to a **limited number of end users** of the product to obtain feedback on the product quality.
* Beta testing **reduces product failure risks** and provides **increased quality** of the product through customer validation.
* Beta Test provides a complete overview of the **true experience** gained by the end users while experiencing the product.

### Entrance Criteria

* Positive responses from alpha sites.
* Customer bugs in alpha testing have been addressed.
* There are no fatal errors which can affect the functionality of the software.
* Secondary platform compatibility testing is complete.
* Regression testing corresponding to bug fixes has been done.
* Beta sites are ready for installation.

### Exit Criteria

* Get responses/feedbacks from the beta testers.
* Prepare a report of all serious bugs.
* Notify bug-fixing issues to developers.

### Guidelines

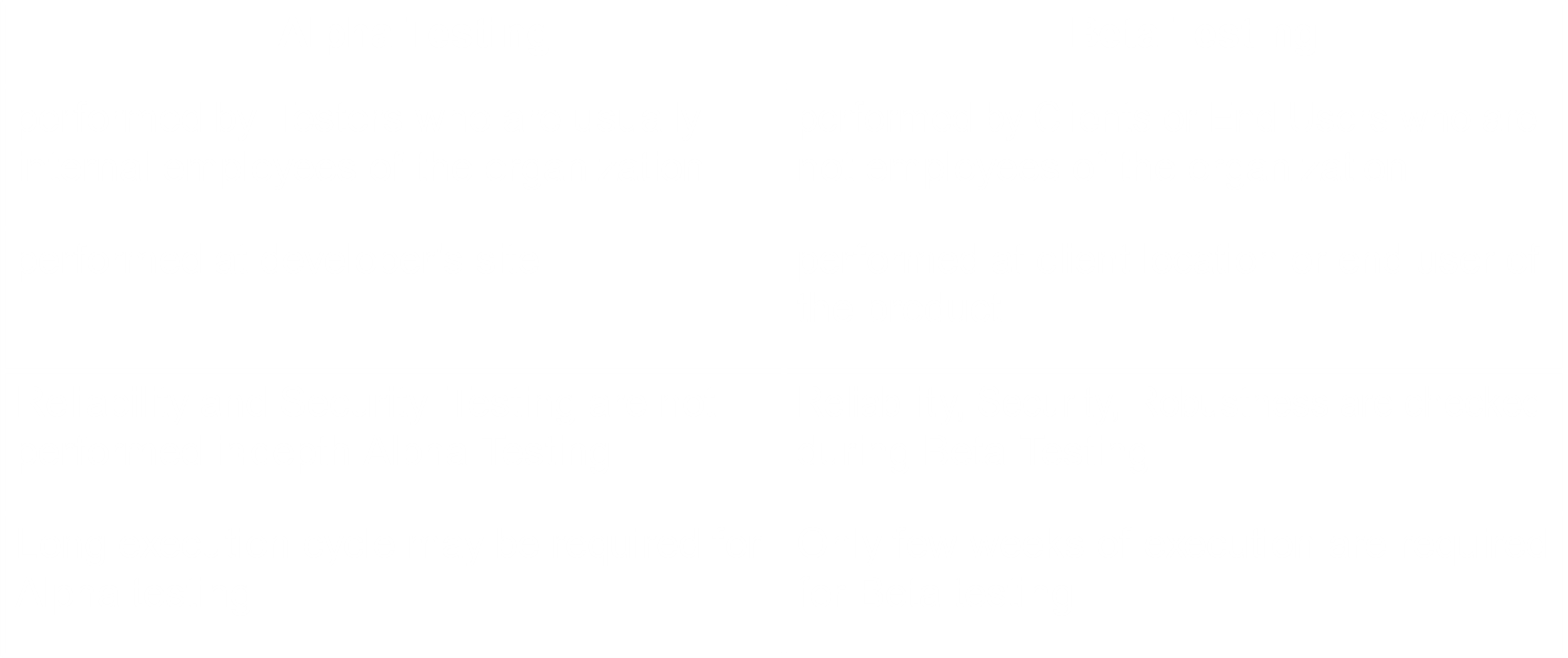
* Don’t expect to release new builds to beta testers more than once every two weeks.
* Don’t plan a beta with fewer than four releases.
* If you add a feature, even a small one, during the beta process, the clock goes back to the beginning of eight weeks and you need another 3–4 releases.

### Advantages

* Reduces product failure risk via customer validation.
* Beta Testing allows a company to test post-launch infrastructure.
* Improves product quality via customer feedback
* Cost effective compared to similar data gathering methods

### Disadvantages

* Doesn’t allow any control over the testing as it is carried out in real environment and not under the lab environment.
* Finding the right beta users and maintaining their participation could be a challenge.



## Smoke Testing

**Smoke testing** is performed on the new build given by developers to the QA team to verify that the **basic functionalities** are working. It is the **first test** that is done on any new build, and if the build fails the smoke test, it is **sent back** without further testing. The test cases chosen cover the most **important functionality** or components of the system. Without smoke testing, we may end up wasting a large amount of time testing a build thoroughly which will have to be changed drastically and tested once again anyways.

Smoke testing normally takes about **60 minutes**. It is even possible to **automate** the tests. Generally, the **QA Lead** performs the smoke test, but the entire **QA team** sits together to discuss the main features of the software and what tests should be performed.

### Scenarios

The following scenarios should be tested in smoke tests:

* **Build Verification** – Check that the correct build is being tested.
* **Account Creation**
* **Login and Logout** – Try both new and old credentials.
* **Business Critical Features** – Test that the most commonly used functionality are not broken.
* **Integration Scenarios** – A simple test should be performed to check integration. For example, if the tester knows that data flows from one module to another, that should be checked, but not thoroughly. This step is more effective the better the tester knows the different system integrations.
* **Add/Edit/Delete** – Check these with the database.
* **Overall Navigation**

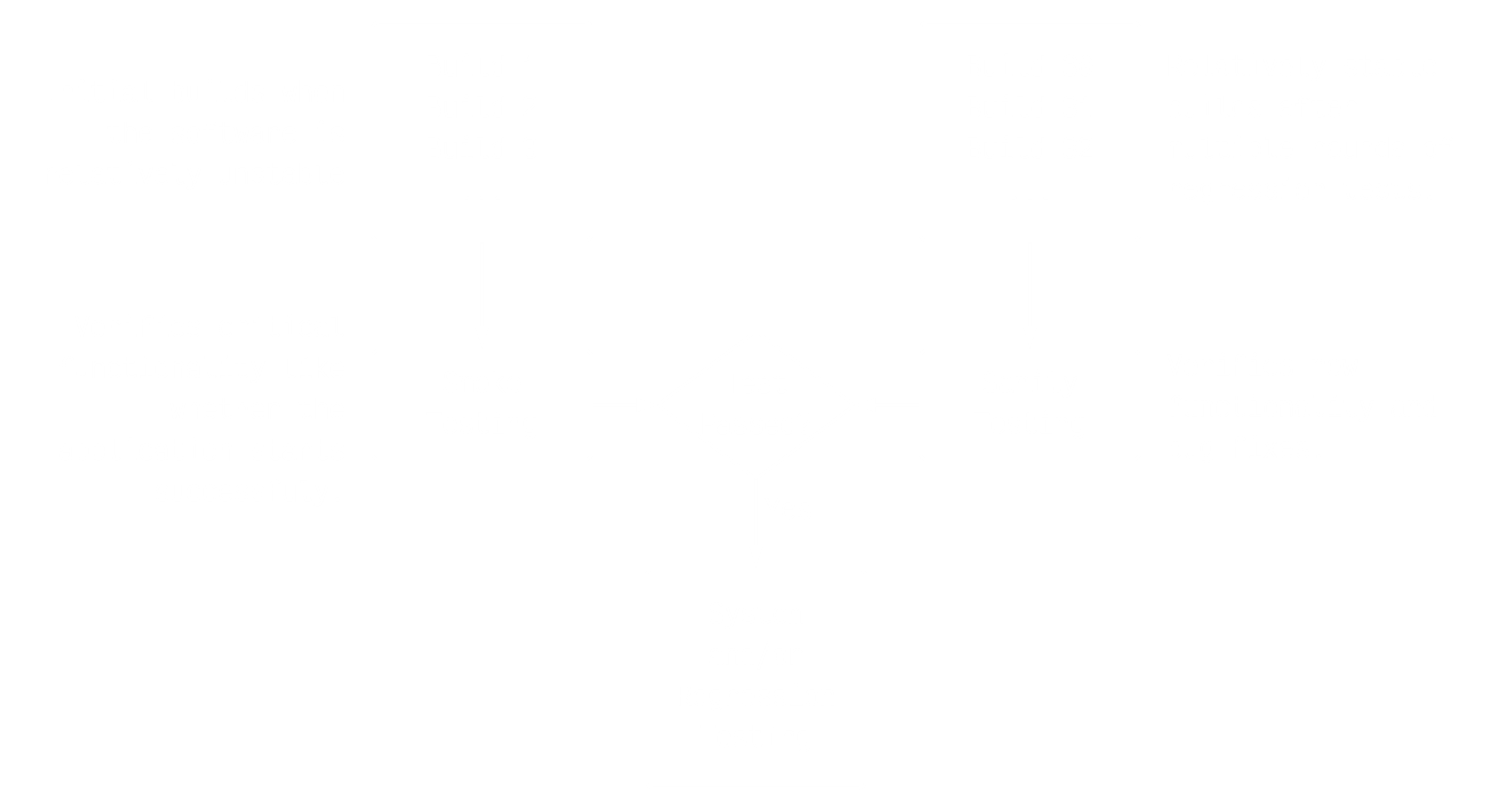
### Advantages

* It helps to find faults earlier in the product lifecycle.
* It saves the testers time by avoiding testing an unstable or wrong build.
* It provides confidence to the tester to proceed with testing.
* It helps to find integration issues faster.
* Major severity defects can be found out.
* Detection and rectification will be an easy process.
* Since execution happens quickly, there will be faster feedback.

## Sanity Testing

When we get a build with only **minor modifications**, we perform a **sanity test** instead of running a thorough regression test. This checks that the modifications **fixed the issues** it was supposed to fix and did not **introduce new issues**. Usually, sanity testing is a subset of regression testing which has a group of test cases related to the changes.

The difference between a sanity test and a smoke test is that **smoke tests** are done for **initial builds** which are unstable, meaning we need to check critical functionality. On the other hand, for **sanity tests**, we are working with later, **stable builds**, so we already know that the critical features will work properly.



## Regression Testing

In **regression testing**, we confirm that new changes did not adversely affect **existing features**. We simply **re-execute** either all or a part of previously executed test cases.

Regression test cases are **carefully selected** to ensure that they cover as much of the system as possible that may have been affected by any change.

Regression testing is crucial during **maintenance**. It is a good idea to **automate** the regression tests so that **all test cases** can be run after each modification.

Whenever we find a **bug**, we should write a **test case** that exhibits the bug. Then, we can use the test case repeatedly in the future to make sure the bug does not **reappear**.

We should perform regression tests when:

* New functionality is added
* Requirements change
* Bugs are fixed
* Performance issues are fixed
* The system environment changes

### Challenges

* Time Consuming
* Expensive
* Complex

### Test Case Selection

* Test cases which have frequent defects
* Functionalities which are more visible to the users
* Test cases which verify core features of the product
* Test cases which have undergone more and recent changes
* All Integration Test Cases
* All Complex Test Cases
* Boundary value test cases
* A sample of Successful test cases
* A sample of Failure test cases

## Non-Functional Testing

Major non-functional testing techniques include:

* Performance testing
  + Load testing
  + Stress testing
  + Scalability testing
  + Stability testing
* Compatibility testing
* Compliance testing
* Recovery testing
* Security testing
* Usability testing

## Performance Testing

**Performance testing** ensures that the application will work well under its expected workload. The goal is not to find bugs, but to eliminate **performance bottlenecks**. It tests the **quality attributes** of the system, such as scalability, reliability and resource usage.

Some factors that become involved are:

* **Throughput** – This is the number of transactions that the software can handle at a time. It is measured in transactions per second (TPS).
* **Response Time** – This is how long a single transaction takes to complete. It is the delay between the point of request and the first response. Response time increases with user load.
* **Tuning** – Performance can be increased by adjusting the parameters of the product, operating system and other components. We can thus improve performance without touching the source code.
* **Benchmarking** – Performance must meet those of competitive products.

Testing that checks the above factors is called performance testing. We keep tuning the system until it reaches the expected level of performance. Performance testing should be done in the design phase, the development phase and the deployment phase.

### Parts to Test

* **High-Frequency Transactions** – The most frequently used transactions should be included in the performance tests since they will affect the performance of other parts.
* **Mission-Critical Transactions** – The most important transactions should be included since they have the greatest impact.
* **Read Transactions** – Read-only transactions are different from complex ones, so we should include at least one of these so that their performance can be judged separately.
* **Update Transactions** – Update transactions are also different from other transactions, so at least one of these should be included.

### Process

1. Identify performance scenarios
2. Plan and design performance test scripts
3. Configure the test environment and distribute the load
4. Execute test scripts
5. Results
6. Result analysis
7. Identify bottlenecks
8. Tuning
9. Retest

### Load Testing

**Load Testing** tests the response of the system under various **load conditions**. We test normal and peak load conditions. This allows us to check the performance of the system with either less than or equal to the desired load level.

### Stress Testing

During **Stress Testing**, we subject the system to **overload** to ensure the system can sustain the stress. Recovery from such a phase is critical since it is highly likely to happen in the production environment.

Stress testing is conducted due to various reasons:

* To monitor system performance during failures/heavy load.
* To verify if the system has saved the data before crashing.
* To verify if the system prints meaningful error messages while crashing.
* To verify if unexpected failures cause security issues.

### Scalability Testing

In **Scalability Testing**, we check the performance of an application by increasing or decreasing the load in particular scales. There are two types of scalability testing:

* **Upward Scalability Testing** – We increase the number of users on a particular scale until the application crashes. This allows us to find the maximum capacity of the application.
* **Downward Scalability Testing** – If Load Testing fails, we use this to decrease users at a particular interval until the goal is achieved. This allows us to find the bottleneck.

### Stability Testing

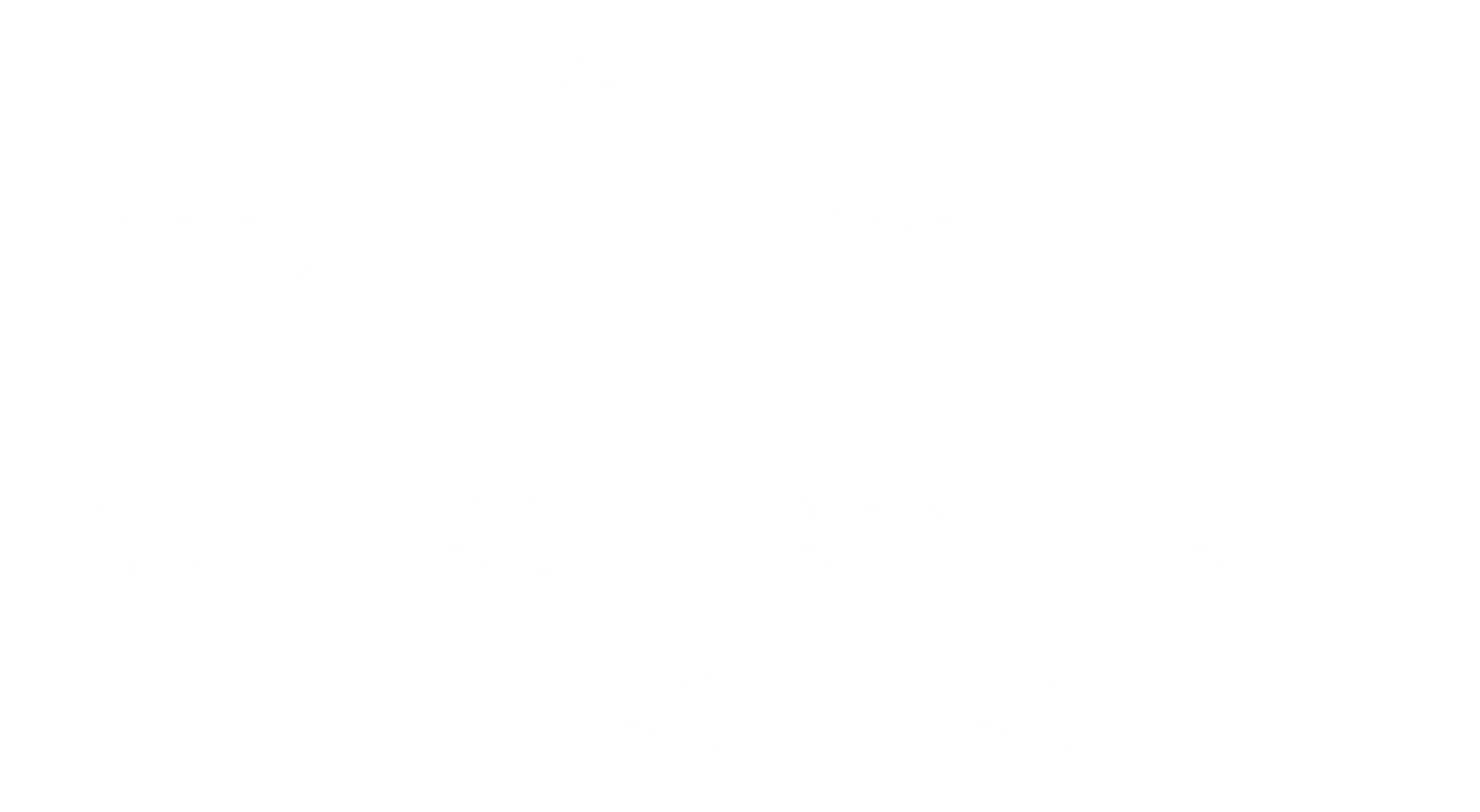
In **Stability Testing**, we check the performance of the system by applying some load for a particular period of time.

### Performance Testing Tools

* Open Source:
  + OpenSTA
  + Diesel Test
  + TestMaker
  + Grinder
  + LoadSim
  + Jmeter
  + Rubis
* Commercial:
  + LoadRunner
  + Silk Performer
  + Qengine
  + Empirix e-Load

## Compatibility Testing

**Compatibility Testing**, also called **Platform Testing**, checks whether the system is capable of running on different hardware, operating systems, applications, network environments and mobile devices.



### Version Checking

In Compatibility Testing, we also perform **Version Checking**, where we are essentially checking that the current version of the software is compatible with other versions. This can be of two types, **Backward** Compatibility Testing and **Forward** Compatibility Testing, both of which do exactly what they sound like.

Backward compatibility testing is easy enough, since we know all the changes to our software from previous versions. Forward compatibility testing is more difficult, since we do not know what changes there will be going forward.

### Benefits

* Enhances Software Development Process
  + Easy to validate application’s stability, usability, and scalability across various platforms and deliver feedback.
  + Helps to analyze defects if any in the development stage itself.
* Identifies Errors before Production
  + ensures to meet the fundamental expectation of users.
  + effectively performs system compatibility tests on different browsers, platforms, configurations, OS, hardware, etc.
  + Thus, as compatibility tests are performed on the app, it ensures to deliver best customer experience.
* Delivers and Meets Customer Expectations
  + capable of timely identification of the defects
  + Since defects are identified earlier before production, the testing practice gives enough time for the developer teams to fix them at the earliest.

### Common Defects

* Defects due to broken tables or frames
* Issues after making variations or modifications to the user interface with respect to look and feel
* Defects with scroll bar
* Alignment issues
* Issues due to changes in colour or style
* Changes or variations of font size and style

## Compliance Testing

Every industry has regulatory and compliance boards that protect end users. In **Compliance Testing**, we ensure that our software meets the rules of such boards.

### Steps

* Professionals, who are knowledgeable and experienced, who understand the compliance must be retained.
* Understanding the risks and impacts of being non-compliant
* Document the processes and follow them
* Perform an internal audit and follow with an action plan to fix the issues

## Recovery Testing

In **recovery testing**, we check how quickly our system can recover from a crash or hardware failure. We forcefully cause a failure to verify that the recovery is successful.

### Steps

1. Determining the feasibility of the recovery process.
2. Verification of the backup facilities.
3. Ensuring proper steps are documented to verify the compatibility of backup facilities.
4. Providing Training within the team.
5. Demonstrating the ability of the organization to recover from all critical failures.
6. Maintaining and updating the recovery plan at regular intervals.

## Security Testing

In **Security Testing**, we ensure that our system protects data and maintains functionality as intended. The process is intended to reveal flaws in the security mechanisms, finding potential loopholes and weaknesses and check whether there is an information leakage.

In Security Testing, we aim to verify the 6 basic principles of security:

1. Confidentiality – Is my secret safe?
2. Integrity – Is my data tampered with?
3. Authentication – Who am I?
4. Authorization – What can I do?
5. Availability – Is the information ready for use when expected?
6. Non-repudiation – Ensure that it cannot later be denied that an action happened, or a communication that took place.

## Usability Testing

**Usability** is the measure of how easy it is to use something, i.e. how easy is it for the typical user to understand, learn and operate our system. Usability is difficult to evaluate and measure. However, it can be done based on the following parameters:

* Level of skill required to learn/use the software. It should maintain the balance for both novice and expert user.
* Time required to get used to in using the software.
* The measure of increase in user productivity if any.
* Assessment of a user's attitude towards using the software.

With usability testing we can:

* Uncover problems in design
* Discover opportunities to improve design
* Learn about user behaviour and preferences

### Process

1. Plan and prepare
2. Select participants
3. Conduct test
4. Analyse results
5. Develop recommendations